responder Documentation

Release 1.3.0

Kenneth Reitz

Mar 10, 2019

Contents

```
import responder

api = responder.API()

@api.route("/{greeting}")
async def greet_world(req, resp, *, greeting):
    resp.text = f"{greeting}, world!"

if __name__ == '__main__':
    api.run()
```

Powered by Starlette. That `async` declaration is optional.

This gets you a ASGI app, with a production static files server (WhiteNoise) pre-installed, jinja2 templating (without additional imports), and a production webserver based on uvloop, serving up requests with gzip compression automatically.

CHAPTER 1

Features

- A pleasant API, with a single import statement.
- Class-based views without inheritance.
- ASGI framework, the future of Python web services.
- WebSocket support!
- The ability to mount any ASGI / WSGI app at a subroute.
- *f-string syntax* route declaration.
- Mutable response object, passed into each view. No need to return anything.
- Background tasks, spawned off in a `ThreadPoolExecutor`.
- GraphQL (with *GraphiQL*) support!
- OpenAPI schema generation, with interactive documentation!
- Single-page webapp support!

CHAPTER 2

Testimonials

"Pleasantly very taken with python-responder. @kennethreitz at his absolute best."

—Rudraksh M.K.

"ASGI is going to enable all sorts of new high-performance web services. It's awesome to see Responder starting to take advantage of that."

—Tom Christie, author of Django REST Framework

"I love that you are exploring new patterns. Go go go!"

— Danny Greenfield, author of Two Scoops of Django

CHAPTER 3

User Guides

3.1 Quick Start!

This section of the documentation exists to provide an introduction to the Responder interface, as well as educate the user on basic functionality.

3.1.1 Declare a Web Service

The first thing you need to do is declare a web service:

```
import responder

api = responder.API()
```

3.1.2 Hello World!

Then, you can add a view / route to it.

Here, we'll make the root URL say "hello world!":

```
@api.route("/")
def hello_world(req, resp):
    resp.text = "hello, world!"
```

3.1.3 Run the Server

Next, we can run our web service easily, with `api.run()`:

```
api.run()
```

This will spin up a production web server on port `5042`, ready for incoming HTTP requests.

Note: you can pass `port=5000` if you want to customize the port. The `PORT` environment variable for established web service providers (e.g. Heroku) will automatically be honored and will set the listening address to `0.0.0.0` automatically (also configurable through the `address` keyword argument).

3.1.4 Accept Route Arguments

If you want dynamic URLs, you can use Python's familiar *f-string syntax* to declare variables in your routes:

```
@api.route("/hello/{who}")
def hello_to(req, resp, *, who):
    resp.text = f"hello, {who}!"
```

A `GET` request to `/hello/brettcannon` will result in a response of `hello, brettcannon!`.

3.1.5 Returning JSON / YAML

If you want your API to send back JSON, simply set the `resp.media` property to a JSON-serializable Python object:

```
@api.route("/hello/{who}/json")
def hello_to(req, resp, *, who):
    resp.media = {"hello": who}
```

A `GET` request to `/hello/guido/json` will result in a response of `{'hello': 'guido'}`.

If the client requests YAML instead (with a header of `Accept: application/x-yaml`), YAML will be sent.

3.1.6 Rendering a Template

If you want to render a template, simply use `api.template`. No need for additional imports:

```
@api.route("/hello/{who}/html")
def hello_html(req, resp, *, who):
    resp.html = api.template('hello.html', who=who)
```

The `api` instance is available as an object during template rendering.

3.1.7 Setting Response Status Code

If you want to set the response status code, simply set `resp.status_code`:

```
@api.route("/416")
def teapot(req, resp):
    resp.status_code = api.status_codes.HTTP_416    # ...or 416
```

3.1.8 Setting Response Headers

If you want to set a response header, like `X-Pizza: 42`, simply modify the `resp.headers` dictionary:

```
@api.route("/pizza")
def pizza_pizza(req, resp):
    resp.headers['X-Pizza'] = '42'
```

That's it!

3.1.9 Receiving Data & Background Tasks

If you're expecting to read any request data, on the server, you need to declare your view as async and await the content.

Here, we'll process our data in the background, while responding immediately to the client:

```
import time

@api.route("/incoming")
async def receive_incoming(req, resp):

    @api.background.task
    def process_data(data):
        """Just sleeps for three seconds, as a demo."""
        time.sleep(3)


    # Parse the incoming data as form-encoded.
    # Note: 'json' and 'yaml' formats are also automatically supported.
    data = await req.media()

    # Process the data (in the background).
    process_data(data)

    # Immediately respond that upload was successful.
    resp.media = {'success': True}
```

A `POST` request to `/incoming` will result in an immediate response of `{'success': true}`.

3.2 Feature Tour

3.2.1 Class-Based Views

Class-based views (and setting some headers and stuff):

```
@api.route("/{greeting}")
class GreetingResource:
    def on_request(self, req, resp, *, greeting):    # or on_get...
        resp.text = f"{greeting}, world!"
        resp.headers.update({'X-Life': '42'})
        resp.status_code = api.status_codes.HTTP_416
```

3.2.2 Background Tasks

Here, you can spawn off a background thread to run any function, out-of-request:

```
@api.route("/")
def hello(req, resp):

    @api.background.task
    def sleep(s=10):
        time.sleep(s)
        print("slept!")

    sleep()
    resp.content = "processing"
```

3.2.3 GraphQL

Serve a GraphQL API:

```
import graphene

class Query(graphene.ObjectType):
    hello = graphene.String(name=graphene.String(default_value="stranger"))

    def resolve_hello(self, info, name):
        return f"Hello {name}"

schema = graphene.Schema(query=Query)
view = responder.ext.GraphQLView(api=api, schema=schema)

api.add_route("/graph", view)
```

Visiting the endpoint will render a *GraphiQL* instance, in the browser.

You can make use of Responder's Request and Response objects in your GraphQL resolvers through `info.context['request']` and `info.context['response']`.

3.2.4 OpenAPI Schema Support

Responder comes with built-in support for OpenAPI / marshmallow:

```
import responder
from marshmallow import Schema, fields

description = "This is a sample server for a pet store."
terms_of_service = "http://example.com/terms/"
contact = {
    "name": "API Support",
    "url": "http://www.example.com/support",
    "email": "support@example.com",
}
license = {
    "name": "Apache 2.0",
    "url": "https://www.apache.org/licenses/LICENSE-2.0.html",
}

api = responder.API(
    title="Web Service",
    version="1.0",
```

(continues on next page)

(continued from previous page)

```
    openapi="3.0.2",
    description=description,
    terms_of_service=terms_of_service,
    contact=contact,
    license=license,
)


@api.schema("Pet")
class PetSchema(Schema):
    name = fields.Str()


@api.route("/")
def route(req, resp):
    """A cute furry animal endpoint.
    ---
    get:
        description: Get a random pet
        responses:
            200:
                description: A pet to be returned
                content:
                    application/json:
                        schema:
                            $ref: '#/components/schemas/Pet'
    """
    resp.media = PetSchema().dump({"name": "little orange"})
```

```
>>> r = api.session().get("http://;/schema.yml")

>>> print(r.text)
components:
  parameters: {}
  responses: {}
  schemas:
    Pet:
      properties:
        name: {type: string}
      type: object
  securitySchemes: {}
info:
  contact: {email: support@example.com, name: API Support, url: 'http://www.example.
↪com/support'}
  description: This is a sample server for a pet store.
  license: {name: Apache 2.0, url: 'https://www.apache.org/licenses/LICENSE-2.0.html'}
  termsOfService: http://example.com/terms/
  title: Web Service
  version: 1.0
openapi: 3.0.2
paths:
  /:
    get:
      description: Get a random pet
      responses:
        200: {description: A pet to be returned, schema: $ref = "#/components/schemas/
↪Pet"}
```

(continues on next page)

(continued from previous page)

```
tags: []
```

3.2.5 Interactive Documentation

Responder can automatically supply API Documentation for you. Using the example above:

```
api = responder.API(
    title="Web Service",
    version="1.0",
    openapi="3.0.2",
    docs_route='/docs',
    description=description,
    terms_of_service=terms_of_service,
    contact=contact,
    license=license,
)
```

This will make `/docs` render interactive documentation for your API.

3.2.6 Mount a WSGI App (e.g. Flask)

Responder gives you the ability to mount another ASGI / WSGI app at a subroute:

```
import responder
from flask import Flask

api = responder.API()
flask = Flask(__name__)

@flask.route('/')
def hello():
    return 'hello'

api.mount('/flask', flask)
```

That's it!

3.2.7 Single-Page Web Apps

If you have a single-page webapp, you can tell Responder to serve up your `static/index.html` at a route, like so:

```
api.add_route("/", static=True)
```

This will make `index.html` the default response to all undefined routes.

3.2.8 Reading / Writing Cookies

Responder makes it very easy to interact with cookies from a Request, or add some to a Response:

```
>>> resp.cookies["hello"] = "world"

>>> req.cookies
{"hello": "world"}
```

To set cookies directives, you should use *resp.set_cookie*:

```
>>> resp.set_cookie("hello", value="world", max_age=60)
```

Supported directives:

- `key` - **Required**
- `value` - [OPTIONAL] - Defaults to `""`.
- `expires` - Defaults to `None`.
- `max_age` - Defaults to `None`.
- `domain` - Defaults to `None`.
- `path` - Defaults to `"/"`.
- `secure` - Defaults to `False`.
- `httponly` - Defaults to `True`.

For more information see directives

3.2.9 Using Cookie-Based Sessions

Responder has built-in support for cookie-based sessions. To enable cookie-based sessions, simply add something to the `resp.session` dictionary:

```
>>> resp.session['username'] = 'kennethreitz'
```

A cookie called `Responder-Session` will be set, which contains all the data in `resp.session`. It is signed, for verification purposes.

You can easily read a Request's session data, that can be trusted to have originated from the API:

```
>>> req.session
{'username': 'kennethreitz'}
```

Note: if you are using this in production, you should pass the `secret_key` argument to `API(...)`:

```
api = responder.API(secret_key=os.environ['SECRET_KEY'])
```

3.2.10 Using `before_request`

If you'd like a view to be executed before every request, simply do the following:

```
@api.route(before_request=True)
def prepare_response(req, resp):
    resp.headers["X-Pizza"] = "42"
```

Now all requests to your HTTP Service will include an `X-Pizza` header.

For `websockets`:

```
@api.route(before_request=True, websocket=True)
def prepare_response(ws):
    await ws.accept()
```

3.2.11 WebSocket Support

Responder supports WebSockets:

```
@api.route('/ws', websocket=True)
async def websocket(ws):
    await ws.accept()
    while True:
        name = await ws.receive_text()
        await ws.send_text(f"Hello {name}!")
    await ws.close()
```

Accepting the connection:

```
await websocket.accept()
```

Sending and receiving data:

```
await websocket.send_{format}(data)
await websocket.receive_{format}(data)
```

Supported formats: `text`, `json`, `bytes`.

Closing the connection:

```
await websocket.close()
```

3.2.12 Using Requests Test Client

Responder comes with a first-class, well supported test client for your ASGI web services: **Requests**.

Here's an example of a test (written with pytest):

```
import myapi

@pytest.fixture
def api():
    return myapi.api

def test_response(api):
    hello = "hello, world!"

    @api.route('/some-url')
    def some_view(req, resp):
        resp.text = hello

    r = api.requests.get(url=api.url_for(some_view))
    assert r.text == hello
```

3.2.13 HSTS (Redirect to HTTPS)

Want HSTS (to redirect all traffic to HTTPS)?

```
api = responder.API(enable_hsts=True)
```

Boom.

3.2.14 CORS

Want CORS ?

```
api = responder.API(cors=True)
```

The default parameters used by **Responder** are restrictive by default, so you'll need to explicitly enable particular origins, methods, or headers, in order for browsers to be permitted to use them in a Cross-Domain context.

In order to set custom parameters, you need to set the `cors_params` argument of `api`, a dictionary containing the following entries:

- `allow_origins` - A list of origins that should be permitted to make cross-origin requests. eg. `['https://example.org', 'https://www.example.org']`. You can use `['*']` to allow any origin.
- `allow_origin_regex` - A regex string to match against origins that should be permitted to make cross-origin requests. eg. `'https://.*\.example\.org'`.
- `allow_methods` - A list of HTTP methods that should be allowed for cross-origin requests. Defaults to *['GET']*. You can use `['*']` to allow all standard methods.
- `allow_headers` - A list of HTTP request headers that should be supported for cross-origin requests. Defaults to `[]`. You can use `['*']` to allow all headers. The `Accept`, `Accept-Language`, `Content-Language` and `Content-Type` headers are always allowed for CORS requests.
- `allow_credentials` - Indicate that cookies should be supported for cross-origin requests. Defaults to `False`.
- `expose_headers` - Indicate any response headers that should be made accessible to the browser. Defaults to `[]`.
- `max_age` - Sets a maximum time in seconds for browsers to cache CORS responses. Defaults to `60`.

3.2.15 Trusted Hosts

Make sure that all the incoming requests headers have a valid `host`, that matches one of the provided patterns in the `allowed_hosts` attribute, in order to prevent HTTP Host Header attacks.

A 400 response will be raised, if a request does not match any of the provided patterns in the `allowed_hosts` attribute.

```
api = responder.API(allowed_hosts=[example.com, tenant.example.com])
```

- `allowed_hosts` - A list of allowed hostnames.

Note:

- By default, all hostnames are allowed.
- Wildcard domains such as `*.example.com` are supported.
- To allow any hostname use `allowed_hosts=["*"]`.

3.3 Deploying Responder

You can deploy Responder anywhere you can deploy a basic Python application.

3.3.1 Docker Deployment

Assuming existing `api.py` and `Pipfile.lock` containing `responder`.

`Dockerfile`:

```
FROM kennethreitz/pipenv
ENV PORT '80'
COPY . /app
CMD python3 api.py
EXPOSE 80
```

That's it!

3.3.2 Heroku Deployment

The basics:

```
$ mkdir my-api
$ cd my-api
$ git init
$ heroku create
...
```

Install Responder:

```
$ pipenv install responder --pre
...
```

Write out an `api.py`:

```
import responder

api = responder.API()

@api.route("/")
async def hello(req, resp):
    resp.text = "hello, world!"

if __name__ == "__main__":
    api.run()
```

Write out a `Procfile`:

```
web: python api.py
```

That's it! Next, we commit and push to Heroku:

```
$ git add -A
$ git commit -m 'initial commit'
$ git push heroku master
```

3.4 Building and Testing with Responder

Responder comes with a first-class, well supported test client for your ASGI web services: **Requests**.

Here, we'll go over the basics of setting up a proper Python package and adding testing to it.

3.4.1 The Basics

Your repository should look like this:

```
Pipfile   Pipfile.lock   api.py  test_api.py
```

`$ cat api.py`:

```
import responder

api = responder.API()

@api.route("/")
def hello_world(req, resp):
    resp.text = "hello, world!"

if __name__ == "__main__":
    api.run()
```

`$ cat Pipfile`:

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
responder = "*"

[dev-packages]
pytest = "*"

[requires]
python_version = "3.7"

[pipenv]
allow_prereleases = true
```

3.4.2 Writing Tests

`$ cat test_api.py`:

```
import pytest
import api as service

@pytest.fixture
def api():
    return service.api
```

(continues on next page)

(continued from previous page)

```
def test_hello_world(api):
    r = api.requests.get("/")
    assert r.text == "hello, world!"
```

```
$ pytest:
```

```
...
========================== 1 passed in 0.10 seconds ==========================
```

3.4.3 (Optional) Proper Python Package

Optionally, you can not rely on relative imports, and instead install your api as a proper package. This requires:

1. A proper setup.py file.
2. `$ pipenv install -e . --dev`

This will allow you to only specify your dependencies once: in `setup.py`. `$ pipenv lock` will automatically lock your transitive dependencies (e.g. Responder), even if it's not specified in the `Pipfile`.

This will ensure that your application gets installed in every developer's environment, using Pipenv.

3.5 API Documentation

3.5.1 Web Service (API) Class

class responder.**API**(*, *debug=False*, *title=None*, *version=None*, *description=None*, *terms_of_service=None*, *contact=None*, *license=None*, *openapi=None*, *openapi_route='/schema.yml'*, *static_dir='static'*, *static_route='/static'*, *templates_dir='templates'*, *auto_escape=True*, *secret_key='NOTASECRET'*, *enable_hsts=False*, *docs_route=None*, *cors=False*, *cors_params={'allow_credentials': False, 'allow_headers': (), 'allow_methods': ('GET',), 'allow_origin_regex': None, 'allow_origins': (), 'expose_headers': (), 'max_age': 600}*, *allowed_hosts=None*)

The primary web-service class.

Parameters

- **static_dir** – The directory to use for static files. Will be created for you if it doesn't already exist.
- **templates_dir** – The directory to use for templates. Will be created for you if it doesn't already exist.
- **auto_escape** – If `True`, HTML and XML templates will automatically be escaped.
- **enable_hsts** – If `True`, send all responses to HTTPS URLs.
- **title** – The title of the application (OpenAPI Info Object)
- **version** – The version of the OpenAPI document (OpenAPI Info Object)
- **description** – The description of the OpenAPI document (OpenAPI Info Object)

- **terms_of_service** – A URL to the Terms of Service for the API (OpenAPI Info Object)
- **contact** – The contact dictionary of the application (OpenAPI Contact Object)
- **license** – The license information of the exposed API (OpenAPI License Object)

add_event_handler(*event_type*, *handler*)
Adds an event handler to the API.

Parameters

- **event_type** – A string in ("startup", "shutdown")
- **handler** – The function to run. Can be either a function or a coroutine.

add_route(*route=None*, *endpoint=None*, ***, *default=False*, *static=False*, *check_existing=True*, *websocket=False*, *before_request=False*)
Adds a route to the API.

Parameters

- **route** – A string representation of the route.
- **endpoint** – The endpoint for the route – can be a callable, or a class.
- **default** – If `True`, all unknown requests will route to this view.
- **static** – If `True`, and no endpoint was passed, render "static/index.html", and it will become a default route.
- **check_existing** – If `True`, an AssertionError will be raised, if the route is already defined.

add_schema(*name*, *schema*, *check_existing=True*)
Adds a mashmallow schema to the API specification.

mount(*route*, *app*)
Mounts an WSGI / ASGI application at a given route.

Parameters

- **route** – String representation of the route to be used (shouldn't be parameterized).
- **app** – The other WSGI / ASGI app.

on_event(*event_type: str*, ***args*)
Decorator for registering functions or coroutines to run at certain events Supported events: startup, cleanup, shutdown, tick

Usage:

```
@api.on_event('startup')
async def open_database_connection_pool():
    ...

@api.on_event('tick', seconds=10)
async def do_stuff():
    ...

@api.on_event('cleanup')
async def close_database_connection_pool():
    ...
```

path_matches_route(*path*)
Given a path portion of a URL, tests that it matches against any registered route.

Parameters **path** – The path portion of a URL, to test all known routes against.

redirect(*resp*, *location*, *, *set_text=True*, *status_code=301*)
Redirects a given response to a given location.

Parameters

- **resp** – The Response to mutate.
- **location** – The location of the redirect.
- **set_text** – If `True`, sets the Redirect body content automatically.
- **status_code** – an *API.status_codes* attribute, or an integer, representing the HTTP status code of the redirect.

requests = None
A Requests session that is connected to the ASGI app.

route(*route=None*, **options*)
Decorator for creating new routes around function and class definitions.

Usage:

```
@api.route("/hello")
def hello(req, resp):
    resp.text = "hello, world!"
```

schema(*name*, **options*)
Decorator for creating new routes around function and class definitions.

Usage:

```
from marshmallow import Schema, fields

@api.schema("Pet")
class PetSchema(Schema):
    name = fields.Str()
```

serve(*, *address=None*, *port=None*, *debug=False*, **options*)
Runs the application with uvicorn. If the `PORT` environment variable is set, requests will be served on that port automatically to all known hosts.

Parameters

- **address** – The address to bind to.
- **port** – The port to bind to. If none is provided, one will be selected at random.
- **debug** – Run uvicorn server in debug mode.
- **options** – Additional keyword arguments to send to `uvicorn.run()`.

session(*base_url='http://;'*)
Testing HTTP client. Returns a Requests session object, able to send HTTP requests to the Responder application.

Parameters **base_url** – The URL to mount the connection adaptor to.

static_url(*asset*)
Given a static asset, return its URL path.

template(*name_*, ***values*)
Renders the given jinja2 template, with provided values supplied.

Note: The current `api` instance is by default passed into the view. This is set in the dict `api.jinja_values_base`.

Parameters

- **name** – The filename of the jinja2 template, in `templates_dir`.
- **values** – Data to pass into the template.

template_string(*s_*, ***values*)
Renders the given jinja2 template string, with provided values supplied.

Note: The current `api` instance is by default passed into the view. This is set in the dict `api.jinja_values_base`.

Parameters

- **s** – The template to use.
- **values** – Data to pass into the template.

url_for(*endpoint*, ***params*)
Given an endpoint, returns a rendered URL for its route.

Parameters

- **endpoint** – The route endpoint you're searching for.
- **params** – Data to pass into the URL generator (for parameterized URLs).

3.5.2 Requests & Responses

class responder.**Request**(*scope*, *receive*, *api=None*)

accepts(*content_type*)
Returns `True` if the incoming Request accepts the given `content_type`.

apparent_encoding
The apparent encoding, provided by the chardet library. Must be awaited.

content
The Request body, as bytes. Must be awaited.

cookies
The cookies sent in the Request, as a dictionary.

encoding
The encoding of the Request's body. Can be set, manually. Must be awaited.

full_url
The full URL of the Request, query parameters and all.

headers
A case-insensitive dictionary, containing all headers sent in the Request.

media(*format=None*)
Renders incoming json/yaml/form data as Python objects. Must be awaited.

Parameters **format** – The name of the format being used. Alternatively accepts a custom callable for the format type.

method
The incoming HTTP method used for the request, lower-cased.

params
A dictionary of the parsed query parameters used for the Request.

session
The session data, in dict form, from the Request.

text
The Request body, as unicode. Must be awaited.

url
The parsed URL of the Request.

class responder.**Response**(*req*, ***, *formats*)

content
A bytes representation of the response body.

cookies
The cookies set in the Response

headers
A Python dictionary of `{key: value}`, representing the headers of the response.

media
A Python object that will be content-negotiated and sent back to the client. Typically, in JSON formatting.

session
The cookie-based session data, in dict form, to add to the Response.

status_code
The HTTP Status Code to use for the Response.

3.5.3 Utility Functions

responder.API.status_codes.**is_100**(*status_code*)

responder.API.status_codes.**is_200**(*status_code*)

responder.API.status_codes.**is_300**(*status_code*)

responder.API.status_codes.**is_400**(*status_code*)

responder.API.status_codes.**is_500**(*status_code*)

CHAPTER 4

Installing Responder

```
$ pipenv install responder --pre
```

Only **Python 3.6+** is supported.

CHAPTER 5

The Basic Idea

The primary concept here is to bring the niceties that are brought forth from both Flask and Falcon and unify them into a single framework, along with some new ideas I have. I also wanted to take some of the API primitives that are instilled in the Requests library and put them into a web framework. So, you'll find a lot of parallels here with Requests.

- Setting `resp.content` sends back bytes.
- Setting `resp.text` sends back unicode, while setting `resp.html` sends back HTML.
- Setting `resp.media` sends back JSON/YAML (`.text`/`.html`/`.content` override this).
- Case-insensitive `req.headers` dict (from Requests directly).
- `resp.status_code`, `req.method`, `req.url`, and other familiar friends.

CHAPTER 6

Ideas

- Flask-style route expression, with new capabilities – all while using Python 3.6+'s new f-string syntax.
- I love Falcon's "every request and response is passed into each view and mutated" methodology, especially `response.media`, and have used it here. In addition to supporting JSON, I have decided to support YAML as well, as Kubernetes is slowly taking over the world, and it uses YAML for all the things. Content-negotiation and all that.
- **A built in testing client that uses the actual Requests you know and love**.
- The ability to mount other WSGI apps easily.
- Automatic gzipped-responses.
- In addition to Falcon's `on_get`, `on_post`, etc methods, Responder features an `on_request` method, which gets called on every type of request, much like Requests.
- A production static files server is built-in.
- Uvicorn built-in as a production web server. I would have chosen Gunicorn, but it doesn't run on Windows. Plus, Uvicorn serves well to protect against slowloris attacks, making nginx unnecessary in production.
- GraphQL support, via Graphene. The goal here is to have any GraphQL query exposable at any route, magically.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search

Python Module Index

r

Index