AIOService Documentation

Release 0.0.1

Denis Makogon

Aug 21, 2017

Contents

Contents:

CHAPTER 1

AIOService overview

AIOService is

Set of routines to build web services.

AIOService is not

A web service, it is a framework that helps to organize web services based on aiohttp.

AIOService: HTTP

AIOService.HTTP contains set of routines to build HTTP web services with nested versioned API controllers. It has:

```
Root HTTP service
Versioned sub-services
Request controller handlers routines
```

CHAPTER 2

How to build a HTTP web service

AIOHTTP

Web server:

```
from aiohttp import web
async def handle(request):
    name = request.match_info.get('name', "Anonymous")
    text = "Hello, " + name
    return web.Response(text=text)

app = web.Application()
app.router.add_get('/', handle)
app.router.add_get('/{name}', handle)

web.run_app(app)
```

Also, you can collect handlers into classes:

```
class Collector(object):

    async def handle(self, request):
        name = request.match_info.get('name', "Anonymous")
        text = "Hello, " + name
        return web.Response(text=text)

app = web.Application()
coll_handlers = Collector()
app.router.add_get('/', coll_handlers.handle)
app.router.add_get('/{name}', coll_handlers.handle)
```

Pros:

```
* simple
```

Cons:

```
* additional custom logic to version handlers
```

AIOService basics

Using aioservice you can:

- define controllers in classes with capability to define its API version on class level
- map versioned controllers to versioned sub-service
- map any number of versioned sub-services to root service
- optionally generate swagger docs from root service

AIOService HTTP controllers

Controller is a class that consists of HTTP handlers where each handler pinned to specific route and HTTP method

Rules for writing controllers:

- each controller should define its version (we enforce developer to build sustainable API)
- each handler must be a coroutine

Controller example:

```
class TestController(controller.ServiceController):

    controller_name = "test_controllers"
    version = "v1"

    @requests.api_action(method='GET', route="{some_key}")
    async def get_some_key(self, request):
        return web.json_response(data={
            "some_key": request.match_info["some_key"]
        }, status=200)
```

So, when this controller will be pinned to versioned sub-service, *get_some_key* coroutine will handle requests coming to route */v1/{some_key}*

AIOService controller action wrapper

Each handler in controller class should be wrapped with *requests.api_action* decorator. With *requests.api_action* it is possible to define which HTTP method, HTTP route are being covered by decorated handler. Also it help in cases when exception somehow is not handled properly to send correctly formatter response to the server side:

```
@requests.api_action(method='GET', route="{some_key}")
async def get_some_key(self, request):
    return web.json_response(data={
        "some_key": request.match_info["some_key"]
    }, status=200)
```

AIOService versioned services

Version service (or sub-service) is a set of controller classes and middleware(s) that are grouped by specific criteria. Such criteria is - controller version. So, using this criteria it is possible to define versioned service to handle API:

```
sub_app = service.VersionedService(
    versioned_controllers,
    middleware=[content_type_validator])
```

Using versioned services it is possible to assign different middleware to different routes grouped by controller(s) version:

```
async def auth_through_token(app: web.Application, handler):
    async def middleware_handler(request: web.Request):
        headers = request.headers
        x_auth_token = headers.get("X-Auth-Token")
        project_id = request.match_info.get('project_id')
        c = config.Config.config_instance()
        try:
            auth = identity.Token(c.auth_url,
                                  token=x_auth_token,
                                  project_id=project_id)
            sess = session.Session(auth=auth)
            ks = client.Client(session=sess,
                               project_id=project_id)
            ks.authenticate(token=x_auth_token)
        except Exception as ex:
            return web.json_response(status=401, data={
                "error": {
                    "message": ("Not authorized. Reason: {}"
                                .format(str(ex)))
                }
            })
        return await handler(request)
    return middleware_handler

async def content_type_validator(app: web.Application, handler):
    async def middleware_handler(request: web.Request):
        headers = request.headers
        content_type = headers.get("Content-Type")
        if request.has_body:
            if "application/json" != content_type:
                return web.json_response(
                    data={
                        "error": {
                            "message": "Invalid content type"
                        }
                    }, status=400)
        return await handler(request)
    return middleware_handler

sub_app_v1 = service.VersionedService(
    versioned_controllers_v1,
    middleware=[content_type_validator])

sub_app_v2 = service.VersionedService(
    versioned_controllers_v2,
    middleware=[content_type_validator, auth_through_token])
```

Note, that you can define two versioned services with the same set of controllers, but they would override each other. Such case is similar to:

```
app.router.add_get('/', handle_v1)
app.router.add_get('/', handle_v2)
```

AIOService root service

Once versioned services are defined:

```
sub_app_v1 = service.VersionedService(
    versioned_controllers_v1,
    middleware=[content_type_validator])

sub_app_v2 = service.VersionedService(
    versioned_controllers_v2,
    middleware=[content_type_validator, auth_through_token])
```

it is possible to bind them to root service:

```
main_app = service.HTTPService(
    subservice_definitions=[sub_app_v1, sub_app_v2],
    event_loop=event_loop
)
```

or, as an alternative, bind root service to versioned service:

```
sub_app_v1.bind_to_service(main_app)
sub_app_v2.bind_to_service(main_app)
```

Swagger docs

Optionally it is possible to use swagger doc generator. It is requires to install *aiohttp_swagger* in first place:

```
pip install aiohttp_swagger==1.0.2
```

Now only one step left:

```
main_app.apply_swagger()
```

It is recommended avoid use of default settings for swagger doc, so *apply_swagger* allows to pass following parameters:

```
swagger_url (default: "/api/doc") - defines at which HTTP route swagger doc will be␣
↪available
description (default:"Swagger API definition") - Swagger doc description
api_version (default:"1.0.0") - Swagger doc version
title (default: "Swagger API") - Swagger doc title
contact (default: "") - developer(s) contacts
```

CHAPTER 3

Indices and tables

- genindex
- modindex
- search