Сверху вниз

Начнем наше исследование сверху вниз — пройдем весь путь, который проходит пользовательский Request, чтобы, наконец, стать прекрасным Response.

Request

Объект Request включает в себя множество полезных данных, которые были собраны по пути от пользовательского браузера до бизнес-логики сервера. Например, путь запроса, заголовки, куки, тело, адрес источника запроса, добавленные атрибуты...

Пример — содержание типового Aiohttp Request:

```
body_exists = {bool} True
can_read_body = {bool} True
charset = {NoneType} None
config_dict = {ChainMapProxy: 1} ChainMapProxy(<Application 0x7fc8b0f21820>)
content = {StreamReader} <StreamReader 98304 bytes>
content_length = {int} 156450
content_type = {str} 'multipart/form-data'
cookies = {mappingproxy: 0} {}
forwarded = {tuple: 0} ()
has_body = {bool} True
headers = {CIMultiDictProxy: 19} <CIMultiDictProxy('Host': '127.0.0.1:8080', 'Connection': 'keep-alive', 'Content-Length': '156450', 'sec-ch-ua': '"Google Chrome";v="95", '...
host = {str} '127.0.0.1:8080'
http_range = {slice} slice(None, None, 1)
if_modified_since = {NoneType} None
if_range = {NoneType} None
if_unmodified_since = {NoneType} None
keep_alive = {bool} True
loop = {_UnixSelectorEventLoop} <_UnixSelectorEventLoop running=True closed=False debug=False>
match_info = {UrlMappingMatchInfo: 0} <MatchInfo {}: <ResourceRoute [POST] <PlainResource  /files.upload> -> <class 'app.api.files.views.upload.FilesUploadView'>>
message = {RawRequestMessage} <RawRequestMessage(method='POST', path='/files.upload', version=HttpVersion(major=1, minor=1), headers=<CIMultiDictProxy('Host': ...
method = {str} 'POST'
path = {str} '/files.upload'
path_qs = {str} '/files.upload'
protocol = {RequestHandler} <RequestHandler connected>
query = {MultiDictProxy: 0} <MultiDictProxy()>
query_string = {str} ''
raw_headers = {tuple: 19} ((b'Host', b'127.0.0.1:8080'), (b'Connection', b'keep-alive'), (b'Content-Length', b'156450'), (b'sec-ch-ua', b'"Google Chrome";v="95", "Chromiu...
raw_path = {str} '/files.upload'
rel_url = {URL} /files.upload
remote = {str} '127.0.0.1'
scheme = {str} 'http'
secure = {bool} False
task = {Task} <Task pending name='Task-7' coro=<RequestHandler.start() running at /Users/artembakulev/Work/kts_drive/venv/lib/python3.9/site-packages/aiohttp/web_pr...
transport = {_SelectorSocketTransport} <_SelectorSocketTransport fd=14 read=polling write=<idle, bufsize=0>>
url = {URL} http://127.0.0.1:8080/files.upload
user = {User} User(trace_id='0af10e62-a887-49af-b49c-03fdb7418b03')
version = {HttpVersion: 2} HttpVersion(major=1, minor=1)
writer = {StreamWriter} <aiohttp.http_writer.StreamWriter object at 0x7fc8b531f3a0>
```

Из объекта Request можно получить данные, переданные клиентом, и определить, что же он хотел сделать. Как понять, какую часть бизнес-логики приложения клиент хотел выполнить этим запросом?

Можно взглянуть на такие параметры, как path ('/files.upload') и method ('POST'). По этой паре чаще всего можно однозначно определить «намерения» запроса. Скорее всего, запрос передает какие-то данные для загрузки куда-то, и для этих действий на веб-сервере написана специальная функция. Но как найти эту связанную функцию? Это для нас может сделать *Router.*

Router

Это объект, который хранит в себе все связки «путь-логика» в приложении и умеет по пути и методу входящего запроса определить: какая часть функционала его «интересует». В aiohttp в роли Router выступает класс *UrlDispatcher,* который обладает двумя ключевыми методами, о предназначении которых можно догадаться по их немного упрощенным сигнатурам:

```
def add_route(
    self,
    method: str,
    path: str,
    handler: Функционал,
) -> Route:
```

```
async def resolve(self, request: Request) -> Функционал:
```

Говоря упрощенно, метод *add_route* добавляет какой-то функционал по ключу (method, path), а метод *resolve* достает эти параметры из *Request* и находит нужный функционал, если он был когда-то добавлен.

Если внимательно посмотреть на сигнатуру метода *add_route,* то можно увидеть, что он возвращает Route. Давайте рассмотрим его внимательнее.

Route

Это чуть более сложный объект, чем просто ключ (method, path). С помощью Route можно, например, задать такие параметры для совпадения пары «запрос-функционал»:

- только строгое соответствие /user/get
- route с именованным параметром /user/{user_id}/get
- route с регулярным выражением внутри /user/{\d+}/get

Также можно обернуть Route в слой дополнительной логики. Например, сделать так, чтобы Route поддерживал CORS:

```
app_.cors.add(app_.router.add_route('POST', '/files.upload', upload_function))
```

В коде выше видно, что на место *handler: Функционал* в сигнатуре метода *add_route* мы передаем какую-то upload_function. Это так называемый function-based view.

Начнем с того, что такое *View* вообще. За ответом переходите к следующей карточке.