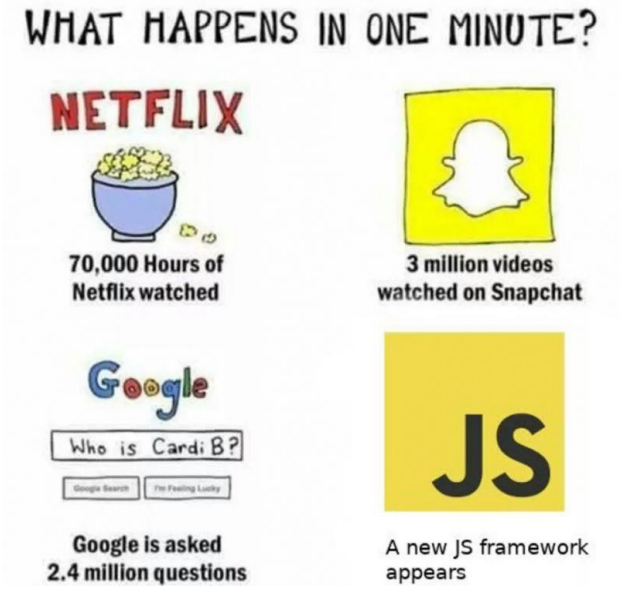
Soal terkait

5. SISTER.Js (6 poin + 9 poin bonus)

Source gambar: https://plainenglish.io/blog/top-10-javascript-frameworks-for-server-side-development-in-2020-6d265016c02



Let's add more !!!

Ketentuan

Buatlah *framework backend* dengan memanfaatkan **socket** dengan menggunakan **NodeJs, Python** , **Java** , **Golang , C , C++**, atau **Rust**. Spesifikasi wajib *framework* adalah sebagai berikut:

- Melakukan *routing request* berdasarkan url dan *request method*. Contoh:

```
app.get('/nilai-akhir', NilaiController.getNilaiAkhir)
```

  Kode ini setiap kali ada request masuk dengan url 'http://[IP]:[PORT]/nilai-akhir' dan request method **GET**, maka akan mengarahkan request ke method **NilaiController.getNilaiAkhir** untuk kemudian diproses oleh method tersebut.

- Menerima request **GET, POST, PUT,** dan **DELETE** dengan spesifikasi minimal setiap *request method* sebagai berikut:
  1. **GET**:
     a. Bisa melakukan routing url biasa(ex: www.google.com)
     b. Bisa melakukan routing url yang mengandung query param beserta menyediakan method untuk meng-ekstrak query paramnya (ex: www.google.com?q=123)
  2. **POST**:
     a. Bisa melakukan routing url biasa (ex: http://recruit.sister20.tech/submit/a)
     b. Bisa memproses request body. Jenis content-type minimal yang bisa diproses antara lain:
        i. text/plain
        ii. application/json
        iii. application/x-www-form-urlencoded
  3. **PUT**:
     a. Bisa memproses url biasa(ex: www.google.com)
     b. Bisa memproses request body. Jenis content-type minimal yang bisa diproses antara lain:

i. text/plain
ii. application/json
iii. application/x-www-form-urlencoded

4. **DELETE**:
   a. Bisa memproses url biasa(ex: www.google.com)

- Dapat memproses HTTP Header. Header minimal yang bisa diproses antara lain:
  - Content-Type
  - Accept

- Mengirimkan response kepada *client* beserta response code yang sesuai dalam bentuk **plaintext** atau **JSON** yang di-*stringify* (Untuk parsing/stringify JSON boleh menggunakan *library* atau membuat sendiri jika ingin mendapatkan nilai bonus)

Bonus

Terdapat beberapa fitur tambahan yang bisa kamu selesaikan jika ingin nilai bonus (+ lagi gabut) sebagai berikut:

1. (Poin 1) Handling *middleware*.
2. (Poin 1) Parsing Content-Type: multipart/form-data
3. (Poin 1) Membuat *parser* JSON tanpa bantuan library (Silahkan implementasikan ilmu TBFO dan Stima)
4. (Poin 2) Handle *file upload* serta dapat mengirimkan *response* berupa *file* (Hint: Implementasikan parser **multipart/form-data**)
5. (Poin 4) Paralelisasi penerimaan dan pemrosesan *request* yang masuk

Tujuan

- Mempelajari bagaimana *framework* yang biasa dipakai di *backend* bekerja
- Menghubungkan materi *low level* yang dipelajari di Jarkom dengan high level di *webdev*
- Kalau ditanya dosen , bisa pamer kalau pernah bikin *framework* BE *from scratch* (fun fact di kelas WBD tahun lalu, Pak Yudis nanya di kelas apakah ada yang pernah pakai node.js buat backend **tanpa** menggunakan Express.js atau framework apapun)

Berkas

- Source code program
- Dokumen (Markdown/PDF) berisi fitur-fitur yang diimplementasikan, cara kerjanya secara singkat, serta petunjuk cara menggunakan fitur tersebut. Untuk tiap fitur, minimal terdapat 1 gambar, baik screenshot kode atau screenshot saat sedang menjalankan fitur tersebut.

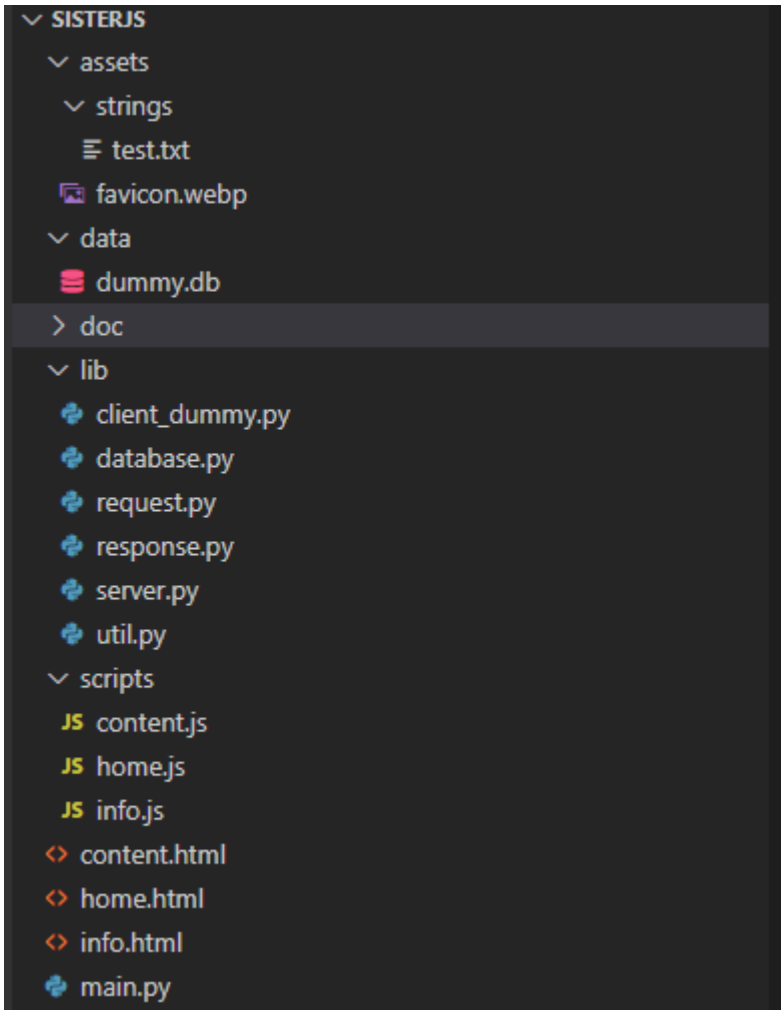
Referensi

- Referensi library socket untuk tiap bahasa:
  - https://nodejs.org/api/net.html#net
  - https://docs.python.org/3/library/socket.html
  - https://docs.oracle.com/javase/8/docs/api/java/net/Socket.html
  - https://doc.rust-lang.org/std/net/index.html
  - https://pkg.go.dev/net
  - https://www.gnu.org/software/libc/manual/html_node/Sockets.html
  - https://man7.org/linux/man-pages/man2/socket.2.html
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/POST

Latar Belakang

Oke jadi buat masalah ini penulis bikin framework make bahasa piton (karena gampang, duh). Untuk desainnya penulis niru banyak banget dari flask. Sebelumnya penulis belom pernah bikin web (ril gw baru belajar bikin web langsung bikin framework), jadi struktur data dan penggunaan dari framework yang penulis buat mungkin cukup aneh, mudah – mudahan karena niru flask jadinya ga begitu aneh yak.

Sekilas dulu ini struktur file dari sisterjs



Buat asset, script, data, sama file - file html itu sebenernya dummy semua kwkwkw. Yang penting semuanya di lib (kecuali client_dummy sama database itu buat testing juga).

Untuk demonya bisa coba jalanin main.py. Di main.py juga ada comment tentang fitur – fitur yang ada di sini (siapa tau males baca). terus client dummy untuk testing kiriman request HTTP atau buka localhost:10000/ di browser, ntar ada web dummy yang isinya crud database.

Untuk implementasinya penulis juga buat dengan object oriented, dengan kelas Server sebagai kelas utamanya dan Server_Handler itu worker thread buat servernya karena bikin multithreading, kelas Request untuk representasi request dari user dan kelas Response untuk membuat response HTTP dari bentuk yang diinput. Util isinya fungsi statik tambahan atau konstanta.

Fitur yang diimplementasikan

1. Routing request

   Untuk pertama yang jelas bikin routing, buat routingnya pertama buka socket dulu di port yang diinginkan, ini ada di kelas server. Port sama address bisa dipilih tapi defaultnya kek di bawah

```python
# Main Server Class
class Server():
    def __init__(self, port:int=10000, addr:str='127.0.0.1', limit:int=5):
        self.default_icon = ''.encode('utf-8')
        self.server_name = "pyster/0.1.0"
        self.socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        self.port = port
        self.addr = addr
        self.socket.bind((addr, self.port))
        self.socket.listen(limit)
        self.running = False
        self.routes = {}
        self.routes_vars = {}
        self.config = {}
        self.before_middlewares = []
        self.after_middlewares = []

        @self.route('/favicon.ico')
        def handle_favicon_route(request):
            return Response(content_type='image/webp', content=self.default_icon)

    # Base
    def run(self):
        print(f"Server running on {self.addr}:{self.port}\n\n")
        self.running = True
        while self.running:
            client_socket, client_address = self.socket.accept()
            handler = Server_Handler(client_socket, client_address, self)
            handler.start()
```

   Terus untuk bikin rutenya, penulis niru flask dengan bikin decorated function, penggunaannya kek gini

```python
# Returnnya harus dalam bentuk Response, cuman string sama html aja yang dikhususin bisa dihandle tanpa bentuk Server_Response
@server.route('/', methods=["GET"])
def handle_home_route(request: Request):
    with open('home.html', 'r') as f:
        content = f.read()
    return Response(content_type='text/html', content=content)
```

   Kalo diliat fungsi route di sana banyak yang bakal ngebingungin soalnya penulis rada ngide buat bisa ngeroute dengan argument di urlnya, bukan kaya query, tapi kaya gini

```python
# Application/json, application/x-www-form-urlencoded bakal diubah langsung ke bentuk dictionary
# NOTE: Yang bisa diterima: Application/json, application/x-www-form-urlencoded, text/plain
@server.route('/api/dummydata/<int>/uhh/<str>', methods=["PUT"])
def handle_home_route(request: Request, *args):
    print("ARGS: ", args)
    print("ACCEPT: ", request.acc_type)
    print("CONTENT-TYPE: ", request.content_type)
    print("CONTENT-LENGTH: ", request.content_length)
    print("DATA: ", request.contents)
    print("QUERY: ", request.query)
    return "Put response with an int and a str, uhh in between"
```

Untuk implementasinya sendiri itu berupa dictionary of string to function, di Server ada di self.routes. Kalo buat yang argument dari url, itu ada routes_vars, isinya nyimpen variabel ada di input ke berapa (ngitung '/') terus tipenya apa. Buat yang disimpen di self.routes untuk yang variabel dia tetep Namanya literal '/api/dummydata/<int>/uhh/<str>'. Nanti waktu Nerima respons aja diparse buat integer sama stringnya diterima sebagai argument terus diganti jadi <int> sama <str>

Buat nyari responsenya itu ada di fungsi server.response(request_data: str). Request_data diproses dulu jadi kelas Request yang lebih rapih dan ngambil ngambilin info kek tipe request, address, extract querynya, sama extract content yang dikirim. Request juga udah ngambil header Accept, Content-Type, sama Content-Length, kek gini doang kelasnya:

```python

# Request Class
class Request():
    def __init__(self, reqstr:str=''):
        reqstr = reqstr
        area = reqstr.split('\r\n\r\n')
        httplines = area[0].split('\r\n')
        request_line = httplines[0].split(' ')
        addr = request_line[1]
        addr_cnt = addr.split('?')

        self.type: str = request_line[0]
        self.addr: str = addr_cnt[0]
        self.query: dict = {}
        self.contents: dict = {}
        self.acc_type: str = ''
        self.content_length: int = 0

        for line in httplines:
            accloc = line.find('Accept:')
            contentloc = line.find('Content-Type:')
            lengthloc = line.find('Content-Length:')

            if(accloc != -1):
                self.acc_type: str = line.split(': ')[1]

            if(contentloc != -1):
                self.content_type: str = line.split(': ')[1]
                if(self.content_type == 'application/x-www-form-urlencoded'):
                    self.contents = extract_wwwquery(area[1])
                elif(self.content_type == 'text/plain'):
                    self.contents = extract_plaintext(area[1])
                elif(self.content_type == 'application/json'):
                    self.contents = extract_json(area[1])

            if(lengthloc != -1):
                self.content_length: int = int(line.split(': ')[1])

        if(len(addr_cnt) > 1 and addr_cnt[1] != ''):
            self.query = extract_wwwquery(addr_cnt[1])
```

Query bakal dipisahin dari address route, spasi dipake buat parsing sama, tanda tanya harus ada di terakhir. Karena itu gaboleh ada rute yang make tanda tanya sama gaboleh ada rute yang make spasi. Parsing query make ini:

```
def extract_wwwquery(query:str):
    query_dict = {}
    query_cnt = query.split('&')
    for query in query_cnt:
        query_pair = query.split('=')
        query_dict[query_pair[0]] = query_pair[1]
    return query_dict
```

Fungsi ini juga dipake buat proses data yang bentuknya Application/x-www-form-urlencoded

Habis itu dicek ada routenya atau engga, awalnya cek polosan, kalo gaada bakal dicek bisa dari route yang punya variabel atau engga. Beres itu dicek lagi di rute itu ada metode yang dibolehin atau engga. Kalo ada baru diproses responsenya sesuai fungsi dekorator yang ditambahin.

Buat return fungsi dekoratornya harus berupa Response, kecuali string, return berupa string bisa otomatis diubah jadi Response. Untuk html sama json ada fungsi wrappernya buat response yaitu json_response(data: dict) sama html_response(path: str). Di kelas response, dari input yang dibuat bisa digenerate data yang perlu disend, kelasnya kek gini:

```
# Response Class
class Response():
    def __init__(self, status_code:int=200, content_type:str='text/plain', content=''):
        self.status_code = status_code
        self.content_type = content_type
        self.content = content

    def override_response(self, server_name:str ='unnamed server', status_code:int=200, content_type:str='text/plain', content='', keep_connection:bool=False):
        self.status_code = status_code
        self.content_type = content_type
        self.content = content
        return self.generate(server_name, keep_connection)

    def generate(self, server_name:str ='unnamed server', keep_connection:bool=False):
        connection = 'Closed'
        if keep_connection:
            connection = 'Keep-Alive'

        response = (
f"HTTP/1.1 {self.status_code} {HTML_ERROR_MESSAGES[self.status_code]}\r\n"
f"Server: {server_name}\r\n"
f"Content-Length: {len(self.content)}\r\n"
f"Content-Type: {self.content_type}\r\n"
f"Connection: {connection}\r\n\r\n"
        )

        if isinstance(self.content, bytes):
            response = response.encode('utf-8') + self.content
        else:
            response += self.content
            response = response.encode('utf-8')

        return response
```

Generate itu buat bikin data yang perlu disend, override itu buat ngeganti beberapa data sebelum degenerate, awal kepikiran use casenya kalo error biar ga bikin kelas baru tapi gatau deng guna atau engga

Habis responsenya di generate, bakal disend sama Server_Handler ke klien. Kurang lebih intinya gitu sih

2. GET request
   Basic aja, GET bakal routing sama ngirim data balik di Response kaya yang udah dijelasin sebelumnya. Paling tambahan, kalo ga dispesifikasi di dekoratornya method apa, dia bakal default ke ‘GET’ doang. Maksudnya kek gini:

```python
# Kalo method di omit, defaultnya dia nambah method GET doang
@server.route('/home')
def handle_home_route(request: Request):
    return html_response('home.html')
```

3. POST request
   Buat POST, routing sama seperti yang udah dijelasin sebelumnya juga. Buat proses body itu ada dari ctor-nya request. Karena penulis ga bikin yang kirim kiriman data, yang bisa diterima cuman Application/json, Application/x-www-form-urlencoded, sama text/plain. Application/json sama Application/x-www-form-urlencoded selalu diubah ke bentuk dictionary. text/plain sebagai string. Ini buat fungsi wrappernya:

```python
def extract_wwwquery(query:str):
    query_dict = {}
    query_cnt = query.split('&')
    for query in query_cnt:
        query_pair = query.split('=')
        query_dict[query_pair[0]] = query_pair[1]
    return query_dict

def extract_plaintext(query:str):
    return query

def extract_json(query:str):
    query_dict = loads(query)
    return query_dict
```

   extract_plaintext ada biar keren aja

4. PUT request
   Buat PUT, routing sama seperti yang udah dijelasin sebelumnya juga. Buat ekstraksi content body juga sama persis kayak di POST request.

5. DELETE request
   Buat DELETE, routing sama seperti yang udah dijelasin sebelumnya juga. Gaada khusus ngapa - ngapain, bikin fungsi decoratornya aja sendiri.

6. HTTP Header
   Extract HTTP Header diproses sama kelas request dengan cara yang jelek banget sebenernya, cuman jalan jadi yaudah dah males mikir dan skil isu soalnya penulis

```
for line in httplines:
    accloc = line.find('Accept:')
    contentloc = line.find('Content-Type:')
    lengthloc = line.find('Content-Length:')

    if(accloc != -1):
        self.acc_type: str = line.split(': ')[1]

    if(contentloc != -1):
        self.content_type: str = line.split(': ')[1]
        if(self.content_type == 'application/x-www-form-urlencoded'):
            self.contents = extract_wwwquery(area[1])
        elif(self.content_type == 'text/plain'):
            self.contents = extract_plaintext(area[1])
        elif(self.content_type == 'application/json'):
            self.contents = extract_json(area[1])

    if(lengthloc != -1):
        self.content_length: int = int(line.split(': ')[1])

if(len(addr_cnt) > 1 and addr_cnt[1] != ''):
    self.query = extract_wwwquery(addr_cnt[1])
```

Yah gitu caranya, linear terus pake str.find dari piton

7. Response

   Untuk response sebenernya kurang lebih cuman bisa text sama json (ada webp juga sih yang ngirim byte). Penulis ga bikin parser sendiri dan make library json. Kaya udah dijelasin pas awal, ngirimnya make kelas Response buat ngeencode jadi paket HTTP yang sesuai. Cuman buat html sama json wrapper di response

```
def html_response(html_page:str, status_code:int=200):
    with open(html_page, 'r') as f:
        content = f.read()
    return Response(status_code, content_type='text/html', content=content)

def json_response(json:str, status_code:int=200):
    content = create_json(json)
    return Response(status_code, content_type='application/json', content=content)
```

   Ini create_json di util

```
def create_json(data:dict):
    json = dumps(data, indent=4)
    return json
```

   Gitu doang, gak begitu kompleks soalnya penulis ga bikin yang kirim kiriman file

8. Middleware

   Penulis bikin support middleware, lagi lagi kaya flask, make decorator dan ada dua versi middleware, yaitu before_request sama after_request

```python
# Middleware! ada 2 middle ware, before request sama after request (niru flask banget emang ini ehe)
@server.before_request()
def handle_before_request(request: Request):
    print("Middleware before request")

@server.after_request()
def handle_before_request(request: Request):
    print("Middleware after request")
```

Buat cara kerjanya sederhana banget, setiap fungsi yang ditambah ke decorator bakal diappend ke array of functions di server

```python
self.config = {}
self.before_middlewares = []
self.after_middlewares = []
```

Terus semua fungsi yang ada di dalem array itu selalu dijalanin setiap sebelum atau sesudah request sesuai dekorator yang dipake

```python
def run_before_middlewares(self, request):
    for middleware in self.before_middlewares:
        middleware(request)

def run_after_middlewares(self, request):
    for middleware in self.after_middlewares:
        middleware(request)
```

Ntar yang ngejalanin bukan server tapi worker thread, ada di poin fitur selanjutnya

9. Multithreading

   Kaya udah penulis singgung di awal, penulis make multithreading buat handle client make library Threading dari python. Strukturnya berupa worker thread gitu di kelas Server_Handler. Ini dia make dependency injection buat ngejalanin fungsi yang ada sama akses data yang perlu dari servernya

```python
# Server Worker Thread
class Server_Handler(threading.Thread):
    def __init__(self, client_socket, client_address, server):
        super().__init__()
        self.server = server
        self.client_socket = client_socket
        self.client_address = client_address
        self.buffer_size = 4096

    def run(self):
        print(f"Worker Thread is handling connection from {self.client_address}") # LOG
        with self.client_socket:
            request_data = self.client_socket.recv(4096).decode('utf-8')

            self.server.run_before_middlewares(request_data)
            response_data = self.server.response(request_data)
            self.server.run_after_middlewares(request_data)

            self.client_socket.send(response_data)
            self.client_socket.close()

        print("Worker Thread Closed Successfully.\n\n") # LOG
```

Jadi kalo ada koneksi, server cuman perlu bikin worker thread ini terus jalanin

```python
def run(self):
    print(f"Server running on {self.addr}:{self.port}\n\n")
    self.running = True
    while self.running:
        client_socket, client_address = self.socket.accept()
        handler = Server_Handler(client_socket, client_address, self)
        handler.start()
```

10. Static Loading
    Isi folder bisa diload secara custom buat masukin folder apa aja yang bakal bisa di fetch pake method GET, jadi ga cuman di folder static.

    ```python
    # Folder bisa langsung diload semuanya dan rekursif ke dalem buat method GET
    server.load_static_folder('scripts')
    server.load_static_folder('assets')
    ```

    Jadi kalo mau masukin script .js atau asset foto foto bisa langsung terus bisa distrukturin sendiri karena udah rekursif ke folder dalem foldernya juga.

    Cuman kebetulan karena penulis skill isu, jadi file yang bisa ada di folder – folder itu cuman .html, .css, .js, .webp, .json, .txt (Penulis ga implementasiin yang kirim – kiriman file hehe)

11. Unconventional Requests
    Ya karena ga di spesifikasiin methodnya apa aja ya di Server sama di Responses, sebenernya secara teori kalo mau bikin method request yang ngarang gitu misal HTTP requestnya mau handle make request “WOKWOKWOKWOK”, itu juga bisa. Asal dari frontend atau dari yang ngirim requestnya mau ngirim request itu. Kebetulan penulis belom nyoba yang ini sih (penulis males), cuman secara teori harusnya bisa.

Penutupan

Yah jadi ini soal yang lumayan menarik penulis ngerjainnya. Kapan lagi belajar web dengan bikin framework ya ga ya ga. Tengkyu udah baca pseudo-laporan ini.