PNU Mini Bootcamp 백엔드&클라우드 과정

1일차 - Python 기초 및 FastAPI 개요

송준우

2025년 2월 3일

멋쟁이사자처럼

목차

2. API개발 기초

2.1 HTTP 프로토콜

HTTP(HyperText Transfer Protocol) 프로토콜의 특징

- 인터넷에서 데이터를 주고받을 수 있는 프로토콜
- 클라이언트와 서버 사이에 이루어지는 요청/응답 프로토콜
- TCP/IP 프로토콜 위에서 동작
- Stateless

2.1 HTTP 프로토콜

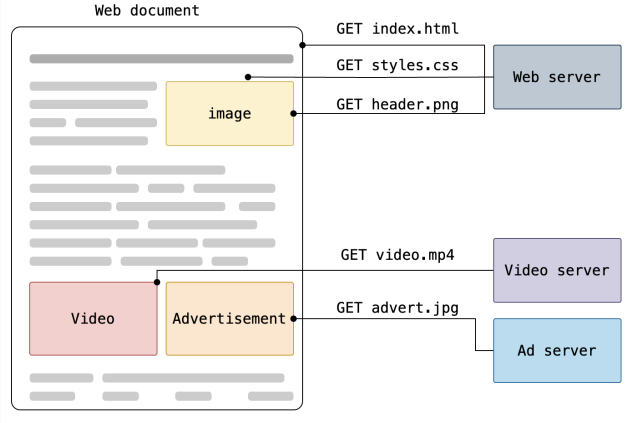


Figure 1: 출처: https://developer.mozilla.org/en-US/docs/Web/HTTP/Overview

2.1 HTTP 프로토콜 - 요청/응답

httpie.io 온라인 터미널로 Google 웹서버와 통신해보기 https://httpie.io/cli/run

```
~ $ http -v GET google.com
GET / HTTP/1.1
Accept: */*
Accept-Encoding: gzip, deflate
Connection: keep-alive
Host: google.com
User-Agent: HTTPie/3.2.1

HTTP/1.1 301 Moved Permanently
Cache-Control: public, max-age=2592000
Content-Length: 219
Content-Security-Policy-Report-Only: object-src 'none';base-uri 'self';script-src ...
Content-Type: text/html; charset=UTF-8
...
X-Frame-Options: SAMEORIGIN
X-XSS-Protection: 0

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-8">
<TITLE>301 Moved</TITLE></HEAD><BODY>
.....
```

2.1 HTTP 프로토콜 - HTTP 요청

Socket 연결을 통해 서버에게 리소스(HTML, 이미지, 동영상 등)를 요청하는 것으로 다음과 같은 구성과 규칙을 가짐

- 요청 라인(Request Line) : 요청 메서드, URL, HTTP 버전
- 요청 헤더(Request Header) : 요청에 대한 정보
- 요청 본문(Request Body) : 요청에 대한 데이터
- 요청 라인과 요청 헤더는 CRLF(Carriage Return Line Feed)로 구분
- 요청 헤더와 요청 본문은 빈 줄로 구분
- 요청 본문은 생략 가능
- 요청 본문이 있는 경우 Content-Length 헤더 필요
- 요청 본문은 주로 POST, PUT 메서드에서 사용
- GET 메서드는 요청 본문 대신 URL에 데이터를 포함(Query String)

2.1 HTTP 프로토콜 - HTTP 요청과 응답 예시

Socket을 이용하여 웹 서버에 연결 후 아래 텍스트를 전송하면 서버는 해당 요청에 대한 응답을 반환한다.

Listing 1: HTTP 요청 예시

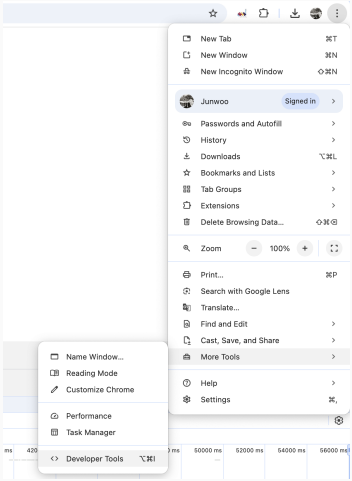
```
GET / HTTP/1.1
Host: google.com
User-Agent: Mozilla/5.0 요청에 사용된 프로그램 이름(주로 브라우저 이름)
```

Listing 2: HTTP 응답 예시

```
HTTP/1.1 200 OK
Date: Sun, 05 Jan 2025 15:37:29 GMT
Cache-Control: private, max-age=0
Content-Type: text/html; charset=ISO-8859-1
...
<!doctype html><html>...
```

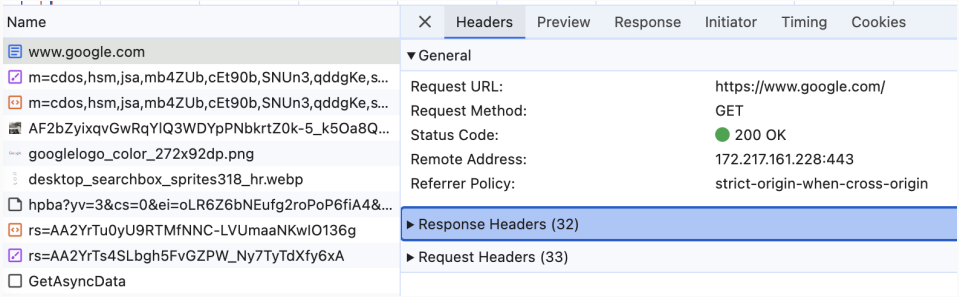
2.1 HTTP 프로토콜 - Chrome 개발자 도구를 이용한 HTTP 요청/응답 분석

1. Chrome 메뉴 →More tools
   →Developer tools →Network 탭
2. www.google.com 접속

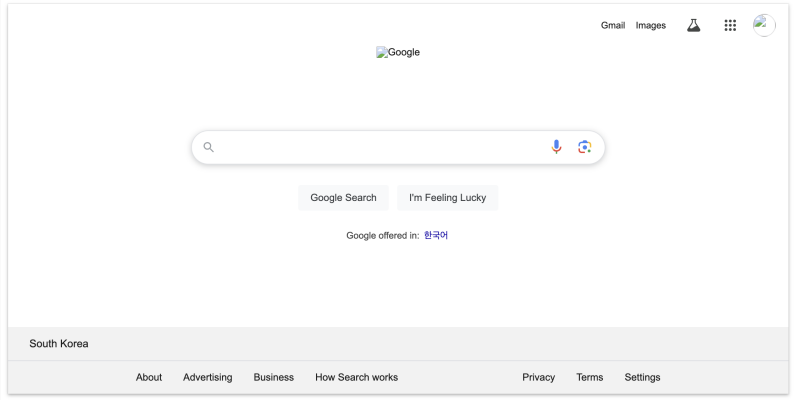


2.1 HTTP 프로토콜 - Chrome 개발자 도구를 이용한 HTTP 요청/응답 분석

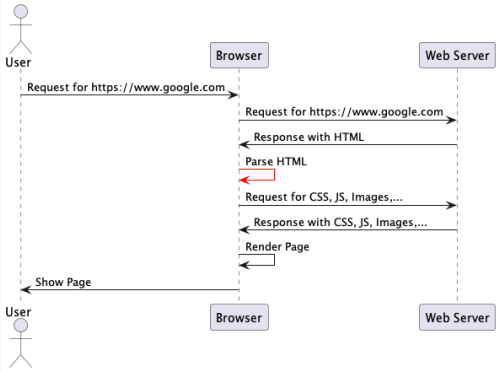
- Network 탭 하단의 요청 목록에서 www.google.com 항목 클릭
- Headers 탭에서 요청/응답 헤더 확인
- Preview 탭에서 응답 리소스의 미리보기 확인(HTML만 렌더링되며 이미지 등의 리소스는 아직 다운로드되지 않은 상태)
- Response 탭에서 응답 본문 확인



Preview 탭에서 일부 이미지가 표시되지 않는 이유



Preview 탭에서 일부 이미지가 표시되지 않는 이유



https://developer.mozilla.org/en-US/docs/Web/Performance/How_browsers_work

2.2 간단한 HTTP 서버 구현하기

Socket 통신을 이용하여 간단한 HTTP서버를 만들어봅시다.

HTTP 서버 프로그램 요구사항

- TCP/IP 소켓을 이용하여 클라이언트의 연결요청 대기
- 클라이언트의 연결 요청을 받으면 연결 수락
- 클라이언트의 요청 정보 읽기
- HTML 텍스트로 응답하기
- 프로그램이 강제종료될 때까지 클라이언트의 요청을 계속 받기

간단한 HTTP 서버 구현하기

Listing 3: 간단한 HTTP 서버 구현하기

```
1  from socket import *
2
3  def createServer():
4      serverSocket = socket(AF_INET, SOCK_STREAM)
5      try:
6          serverSocket.bind(('localhost', 8080))
7          serverSocket.listen()
8          while True:
9              (connectionSocket, addr) = serverSocket.accept() # Blocking
10             print('Connection received from ', addr)
11
12             request = connectionSocket.recv(1024).decode('utf-8')
```

2.2 간단한 HTTP 서버 구현하기 ii

```
13              print(request)
14
15              response = 'HTTP/1.1 200 OK\n'
16              response += 'Content-Type: text/html\n'
17              response += '\n'
18              response += '<html><body>Hello World</body></html>\n'
19              connectionSocket.sendall(response.encode('utf-8'))
20              connectionSocket.shutdown(SHUT_WR)
21
22      except KeyboardInterrupt:
23          print('\nShutting down the server\n')
24          serverSocket.close()
25
26  if __name__ == '__main__':
27      createServer()
```

2.2 간단한 HTTP 서버 구현하기 iii

소스코드 다운로드: https://github.com/song9063/PNU-Bootcamp-2025/blob/main/Day1/Day1-2.2-http-server-main.py

2.2 간단한 HTTP 서버 구현하기

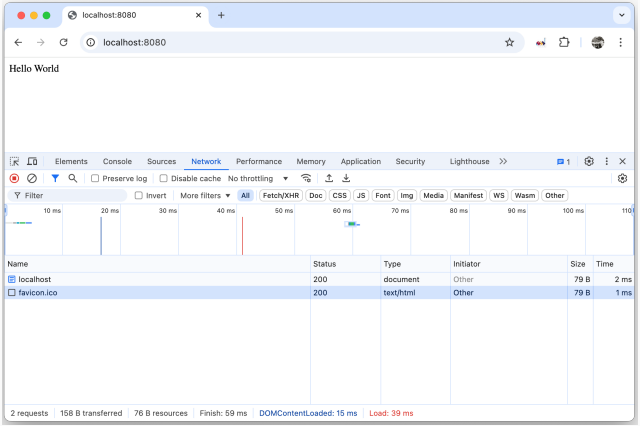
주요 키워드

키워드	설명
socket	네트워크 통신을 위한 소켓 객체
AF_INET	IPv4 주소체계(AF_INET6=IPv6)
SOCK_STREAM	TCP 프로토콜(SOCK_DGRAM=UDP)
bind	소켓에 서버주소와 포트번호를 지정
listen	클라이언트의 연결요청 대기
accept	클라이언트의 연결요청 수락
recv	클라이언트의 요청 정보 읽기
sendall	클라이언트에게 응답 보내기
shutdown	소켓 연결 종료

2.2 간단한 HTTP 서버 구현하기 - 클라이언트 요청 분석

Chrome 브라우저로 http://localhost:8080에 접속하면 Chrome은 아래와 같이 총 2개의 리소스에 대한 요청을 보냅니다.

1. / Web Root의 HTML
2. favicon.ico



2.2 간단한 HTTP 서버 구현하기 - 클라이언트 요청 분석 i

Chrome 브라우저가 보낸 요청 헤더는 웹 브라우저의 기능을 위한 여러 정보가 있어서 복잡합니다.
간단한 HTTP 클라이언트 프로그램을 이용하여 단계적으로 요청 헤더를 분석해봅시다.

Listing 4: HTTP 클라이언트

```
1  import socket
2
3  cliSocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
4  cliSocket.connect(('localhost', 8080))
5
6  strCmd = '''GET / HTTP/1.1
7  Host: localhost:8080
8
9  '''.encode()
10 cliSocket.send(strCmd)
```

2.2 간단한 HTTP 서버 구현하기 - 클라이언트 요청 분석 ii

```
11
12 while True:
13     response = cliSocket.recv(1024)
14     if not response or len(response) < 1:
15         break
16     print(response.decode(), end='')
17
18 cliSocket.close()
```

소스코드 다운로드: https://github.com/song9063/PNU-Bootcamp-2025/blob/main/Day1/Day1-2.2-http-client-main.py

서버 프로그램을 실행한 후 위의 클라이언트 프로그램을 실행해보세요.

2.2 간단한 HTTP 서버 구현하기 - 클라이언트 요청 분석

Listing 5: 서버 로그

```
1 Connection received from  ('127.0.0.1', 55935)
2 GET / HTTP/1.1
3 Host: localhost:8080
```

Listing 6: 클라이언트 로그

```
1 HTTP/1.1 200 OK
2 Content-Type: text/html
3
4 <html><body>Hello World</body></html>
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/Overviewhttp_flow

2.2 간단한 HTTP 서버 구현하기 - 라우팅 추가하기

http://localhost:8080/user/list로 요청이 들어오면 사용자의 목록을 JSON으로 응답하도록 서버 프로그램을 수정해봅시다.

먼저 클라이언트의 요청 문자열을 분석하여 요청 URL을 추출해야 합니다.
서버 프로그램에 아래와 같이 요청 헤더에서 URL을 추출하는 함수를 추가합니다.

Listing 7: URL 추출함수

```
1  import re
2  def parseRequest(requests: str) -> str | None:
3      if len(requests) < 1:
4          return None
5      arRequests = requests.split('\n')
6      for line in arRequests:
7          match = re.search(r'\b(GET|POST|DELETE|PUT|PATCH)\b\s+(.*?)\s+HTTP/1.1', line)
8          if match:
9              strPath = match.group(2)
10             return strPath
11     return None
```

2.2 간단한 HTTP 서버 구현하기 - 라우팅 추가하기

recv().decode() 함수 다음줄에 직전에 만든 parseRequest() 함수를 호출하고 함수의 반환값을 출력해봅시다.

Listing 8: URL 추출함수

```
1            request = connectionSocket.recv(1024).decode('utf-8')
2            strPath = parseRequest(request)
3            print(strPath)
```

서버 코드 수정 후 서버 프로그램을 재실행하고 클라이언트 프로그램을 실행하여 http://localhost:8080 로 요청을 보내봅시다.

Listing 9: 서버 로그

```
Connection received from  ('127.0.0.1', 57724)
Path: /
```

2.2 간단한 HTTP 서버 구현하기 - 라우팅 추가하기

클라이언트 프로그램의 strCmd 문자열을 아래와 같이 수정하여
http://localhost:8080/user/list로 요청을 보내봅시다.

Listing 10: URL 추출함수

```
1 strCmd = '''GET /user/list HTTP/1.1
2 Host: localhost:8080
3
4 '''.encode()
```

서버 코드 수정 후 서버 프로그램을 재실행하고 클라이언트 프로그램을 실행하여
http://localhost:8080 로 요청을 보내봅시다.

Listing 11: 서버 로그

```
Connection received from  ('127.0.0.1', 58517)
Path: /user/list
```

2.2 간단한 HTTP 서버 구현하기 - 라우팅 추가하기

이제 서버프로그램을 수정하여 클라이언트가 /user/list로 요청을 보내면 JSON 형식으로 사용자 목록을 응답하도록 수정해봅시다.

Listing 12: Dummy 사용자 목록을 리턴하는 함수

```
1  import json
2
3  def getUserList():
4      return [
5          {'id': 1, 'name': 'Trump'},
6          {'id': 2, 'name': 'Biden'},
7          {'id': 3, 'name': 'Obama'},
8      ]
```

2.2 간단한 HTTP 서버 구현하기 - 라우팅 추가하기

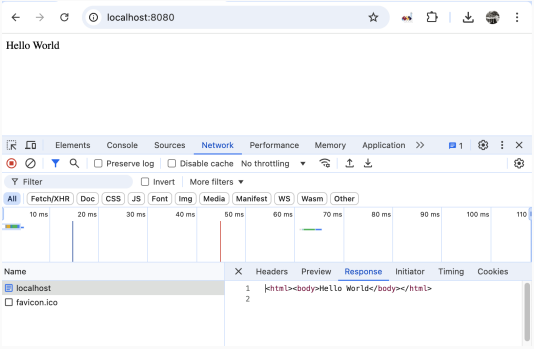
parseRequest() 함수의 반환값을 if문으로 검사하여 /user/list로 요청이 들어오면 getUserList() 함수를 호출하여 JSON으로 응답하고 그 외의 요청이 들어오면 기존처럼 HTML로 응답하도록 수정해봅시다.

Listing 13: Dummy 사용자 목록을 리턴하는 함수

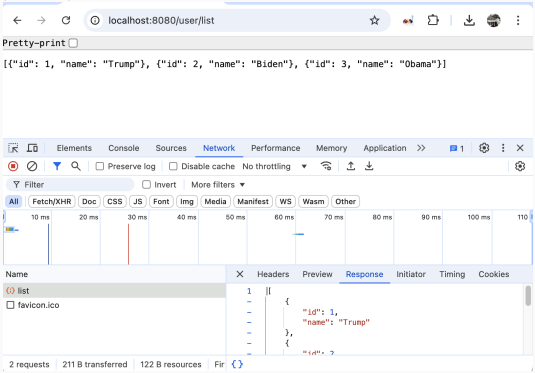
```
response = 'HTTP/1.1 200 OK\n'
if strPath == '/user/list':
    response += 'Content-Type: application/json\n'
    response += '\n'
    response += json.dumps(getUserList())
else:
    response += 'Content-Type: text/html\n'
    response += '\n'
    response += '<html><body>Hello World</body></html>\n'
```

2.2 간단한 HTTP 서버 구현하기 - 라우팅 추가하기

/ 요청에 대한 결과화면



/user/list 요청에 대한 결과화면



소스코드 다운로드: https://github.com/song9063/PNU-Bootcamp-2025/blob/main/Day1/Day1-2.3-http-server-main.py

2.2 간단한 HTTP 서버 구현하기 - HTTP 응답코드

HTTP 응답상태코드 https://developer.mozilla.org/en-US/docs/Web/HTTP/Status

- 1xx : Informational
- 2xx : Success
- 3xx : Redirection
- 4xx : Client Error
- 5xx : Server Error

2.2 간단한 HTTP 서버 구현하기 - HTTP 응답코드

우리 서버는 /user/list 이외의 모든 요청에 대해서 동일한 HTML을 응답하도록 구현되어 있습니다.
웹 루트와 /user/list 이외의 요청에 대해서는 404 Not Found 응답을 보내도록 수정해봅시다.

createServer() 함수 시작 부분에 아래와 같이 응답 가능한 URL 목록을 list형태로 정의합니다.

Listing 14: 응답 가능한 URL 목록

```
1 def createServer():
2     arPath = ['/', '/user/list']
```

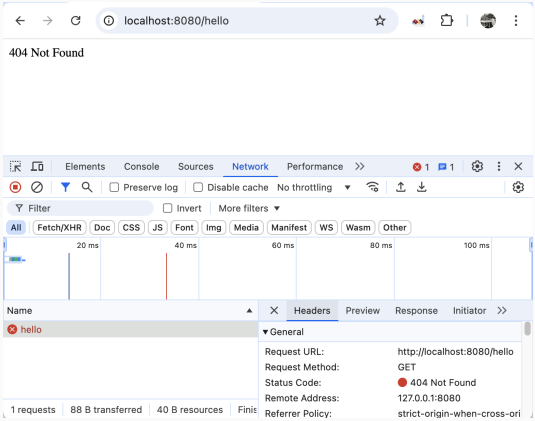
2.2 간단한 HTTP 서버 구현하기 - HTTP 응답코드

createServer() 함수 내용 중 parseRequest() 함수의 반환값을 if문으로 검사하여
지원하지않는 URL로 요청이 들어오면 404 Not Found 응답을 보내도록 수정합니다.

```
1  strPath = parseRequest(request)
2  print(f'Path: {strPath}')
3  if strPath is None:
4      connectionSocket.shutdown(SHUT_WR)
5          continue
6  if strPath not in arPath:
7      response = 'HTTP/1.1 404 Not Found\n'
8      response += 'Content-Type: text/html\n'
9      response += '\n'
10     response += '<html><body>404 Not Found</body></html>\n'
11     connectionSocket.sendall(response.encode('utf-8'))
12     connectionSocket.shutdown(SHUT_WR)
13     continue
```

2.2 간단한 HTTP 서버 구현하기 - HTTP 응답코드

웹 브라우저로 지원하지않는 주소로 요청을 보내면 404 Not Found 응답을 받는 것을 확인할 수 있습니다.



2.2 간단한 HTTP 서버 구현하기 - 파일응답 보내기

파일 요청에 대한 응답을 만들어봅시다.
www.google.com에서 google 로고 이미지를 다운로드 받아서 서버 프로그램이 실행되는 디렉토리에 저장합니다.
파일명: google.png

서버 프로그램에 아래와 같이 google.png 요청을 위한 주소를 추가합니다.

```
def createServer():
    arPath = ['/', '/user/list', '/google.png']
```

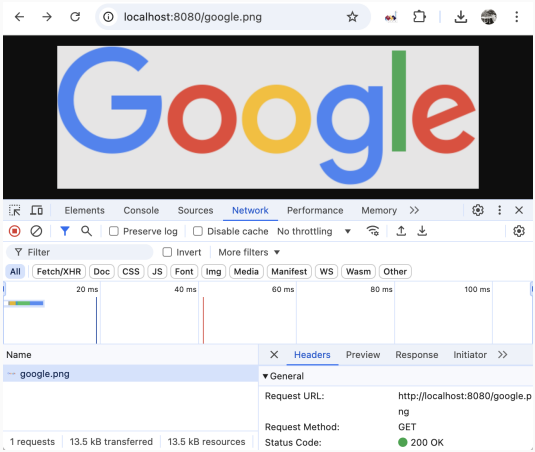
2.2 간단한 HTTP 서버 구현하기 - 파일응답 보내기

strPath를 검사하는 if문에 google.png 요청에 대한 응답을 추가합니다.

```
1  elif strPath == '/google.png':
2      response += 'Content-Type: image/png\n'
3      response += '\n'
4      connectionSocket.sendall(response.encode('utf-8'))
5      with open('google.png', 'rb') as f:
6          while chunk := f.read(1024):
7              connectionSocket.sendall(chunk)
8      connectionSocket.shutdown(SHUT_WR)
9      continue
```

2.2 간단한 HTTP 서버 구현하기 - 파일응답 보내기

웹 브라우저로 http://localhost:8080/google.png로 요청을 보내 이미지가 정상적으로 표시되는 것을 확인할 수 있습니다.



응답 헤더의 Content-Type

서버는 클라이언트에게 응답 본문의 데이터 타입을 알려주기 위해 Content-Type 헤더를 사용합니다. Content-Type 헤더는 MIME 타입을 사용하며, MIME 타입은 데이터의 형식을 나타내는 문자열입니다.

- text/html : HTML 문서
- image/png : PNG 이미지
- image/jpeg : JPEG 이미지
- application/json : JSON 데이터 등

https://developer.mozilla.org/en-US/docs/Web/HTTP/MIME_types

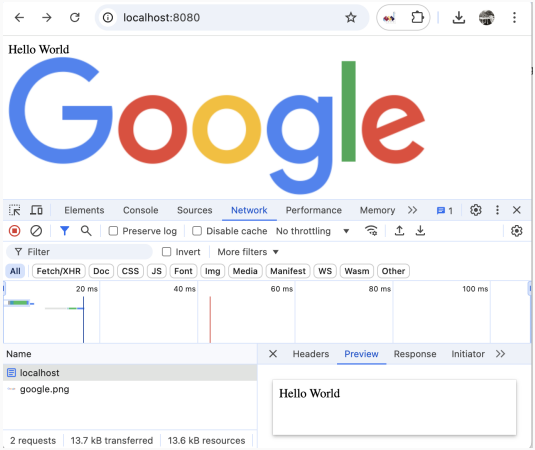
2.2 간단한 HTTP 서버 구현하기 - 이미지 태그가 포함된 HTML 응답 보내기

웹 루트 요청에 대한 HTML에 이미지 태그를 추가해봅시다.
아래 코드처럼 웹 루트 요청에 대한 응답에 이미지 태그를 추가합니다.

```
1 response += '<html><body>Hello World<br /><img src="/google.png"
      /></body></html>\n'
```

2.2 간단한 HTTP 서버 구현하기 - 이미지 태그가 포함된 HTML 응답 보내기

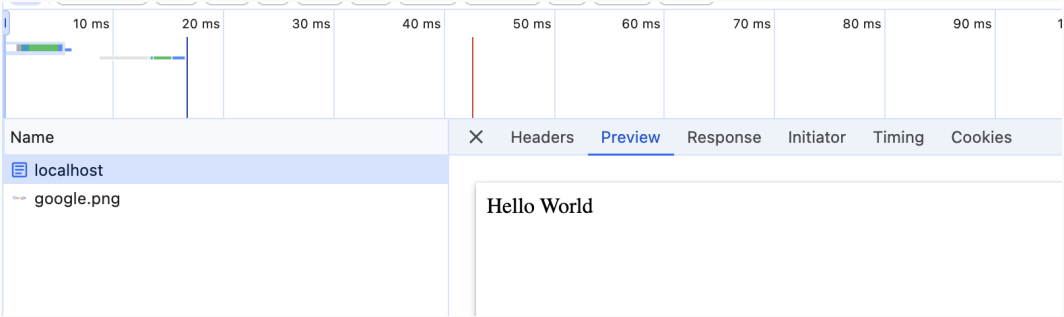
Chrome 브라우저로 http://localhost:8080로 요청을 보내고 개발자 도구의 Network탭 내용을 확인합니다.



2.2 간단한 HTTP 서버 구현하기 - 이미지 태그가 포함된 HTML 응답 보내기

앞에서 보았던 google.com의 응답과 마찬가지로 웹 페이지에 대한 응답에서는 이미지를 볼 수가 없습니다. 웹 브라우저는 HTML 문서를 파싱하면서 이미지 등의 리소스 URL이 확인되면 해당 리소스를 다운로드하기 위해 이어서 요청을 보냅니다.

서버 로그를 확인해보면 / 요청과 /google.png 이렇게 총 2번의 요청이 들어온 것을 확인할 수 있습니다.



2.2 간단한 HTTP 서버 구현하기 - 다른 주소로 이동 시키기

웹 브라우저에게 3** 응답코드와 함께 다른 URL을 보내면 브라우저는 해당 주소로 자동으로 이동합니다.(Redirection)

웹 브라우저로 http://localhost:8080/google로 접속하면 www.google.com으로 이동하도록 서버 프로그램을 수정해봅시다.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Redirectionstemporary_redirections

2.2 간단한 HTTP 서버 구현하기 - 다른 주소로 이동 시키기

서버 프로그램에 아래와 같이 /google 요청을 위한 주소를 추가합니다.

```
def createServer():
    arPath = ['/', '/user/list', '/google.png', '/google']
```

```
if strPath == '/google':
    response = 'HTTP/1.1 303 See Other\n'
    response += 'Location: https://www.google.com\n'
    response += 'Content-Type: text/html\n'
    response += '\n'
    connectionSocket.sendall(response.encode('utf-8'))
    connectionSocket.shutdown(SHUT_WR)
    continue
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링

지금까지 작성한 코드는 createServer() 함수 내부에 모든 요청들에 대한 처리 로직이 들어 있습니다.

이런 형태의 코드는 유지보수가 매우 어렵습니다.
이 예제를 리팩토링하여 FastAPI와 유사한 형태로 코드를 구조화해봅시다.

현재까지 작성한 코드의 완성파일
https://github.com/song9063/PNU-Bootcamp-2025/blob/main/Day1/Day1-2.5-http-server-main.py

2.2 간단한 HTTP 서버 구현하기 - 리팩토링

리팩토링 목표

1. HTTP 응답헤더를 상수로 정의
2. HTTP 요청헤더와 응답헤더를 dataclass로 구조화하기
3. HTTP 요청에 대한 처리 로직을 함수로 분리
4. HTML 파일 렌더링

HTTP 응답헤더 상수로 정의

Enum을 이용하여 HTTP 응답헤더에 사용할 문자열들을 상수로 정의합니다.

```
from enum import Enum
class HttpContentType(Enum):
    TEXT_HTML = 'text/html'
    APPLICATION_JSON = 'application/json'
    IMAGE_PNG = 'image/png'
```

HTTP 요청헤더를 위한 dataclass 만들기

@dataclass를 사용하여 요청헤더를 구조화하고 parseRequest() 함수의 반환형으로 사용합니다.

```
from dataclasses import dataclass

class HTTPMethod(Enum):
    GET = 'GET'
    POST = 'POST'

@dataclass
class HTTPRequest:
    method: HTTPMethod
    url: str
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 2

```
def parseRequest(requests: str) -> HTTPRequest | None:
    if len(requests) < 1:
        return None
    arRequests = requests.split('\n')
    for line in arRequests:
        match =
            re.search(r'\b(GET|POST|DELETE|PUT|PATCH)\b\s+(.*?)\s+HTTP/1.1',
            line)
        if match:
            method = HTTPMethod(match.group(1))
            url = match.group(2)
            try:
                return HTTPRequest(method, url)
            except ValueError:
                return None
    return None
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 2

parseRequest() 함수의 반환값 검사 코드를 아래와 같이 수정합니다.

```
req = parseRequest(request)
if req is None or req.url is None:
    connectionSocket.shutdown(SHUT_WR)
    continue
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 2

strPath를 검사하는 if문을 오른쪽과 같이 수정합니다.

```
if strPath not in arPath:
if strPath == '/google':
if strPath == '/user/list':
elif strPath == '/':
elif strPath == '/google.png':
```

```
if req.url not in arPath:
if req.url == '/google':
if req.url == '/user/list':
elif req.url == '/':
elif req.url == '/google.png':
```

HTTP 응답헤더를 위한 Enum 만들기

응답헤더를 **Enum**으로 정의한 후 응답헤더를 생성하는 함수를 만들어봅시다.
상태코드는 숫자와 문자열을 가지고 있어야 하므로 touple로 Enum 클래스를 만듭니다.

```
1  class HTTPStatusCode(Enum):
2      OK = (200, 'OK')
3      NOT_FOUND = (404, 'Not Found')
4      SEE_OTHER = (303, 'See Other')
5      SERVER_ERROR = (500, 'Internal Server Error')
6
7  def makeResponseHeader(status: HTTPStatusCode, contentType:
       HttpContentType, extra: dict|None = None) -> str:
8      strResp = f'HTTP/1.1 {status.value[0]} {status.value[1]}\n'
9      strResp += f'Content-Type: {contentType.value}\n'
```

```
10      if extra:
11          for key, value in extra.items():
12              strResp += f'{key}: {value}\n'
13      strResp += '\n'
14      return strResp
```

makeResponseHeader() 함수로 응답헤더 출력코드 간결하게 만들기

변경전:

```
1 if req.url not in arPath:
2     print('Resource not found')
3     response = 'HTTP/1.1 404 Not Found\n'
4     response += 'Content-Type: text/html\n'
5     response += '\n'
```

변경후:

```
1 if req.url not in arPath:
2     print('Resource not found')
3     response = makeResponseHeader(HTTPStatusCode.NOT_FOUND,
          HttpContentType.TEXT_HTML)
```

makeResponseHeader() 함수로 응답헤더 출력코드 간결하게 만들기

변경전:

```
1 if req.url == '/google':
2     response = 'HTTP/1.1 303 See Other\n'
3     response += 'Location: https://www.google.com\n'
4     response += 'Content-Type: text/html\n'
5     response += '\n'
```

변경후:

```
1 if req.url == '/google':
2     response = makeResponseHeader(HTTPStatusCode.SEE_OTHER,
          HttpContentType.TEXT_HTML, {'Location': 'https://www.google.com'})
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 2

makeResponseHeader() 함수로 응답헤더 출력코드 간결하게 만들기

아래 코드들을 다음 페이의의 코드들로 변경해봅시다.

```
            response = 'HTTP/1.1 200 OK\n'
            if req.url == '/user/list':
                response += 'Content-Type: application/json\n'
                response += '\n'
                response += json.dumps(getUserList())
            elif req.url == '/':
                response += 'Content-Type: text/html\n'
                response += '\n'
                response += '<html><body>Hello World<br /><img src="/google.png" /></body></html>\n'
            elif req.url == '/google.png':
                response += 'Content-Type: image/png\n'
                response += '\n'
                connectionSocket.sendall(response.encode('utf-8'))
                with open('google.png', 'rb') as f:
                    while chunk := f.read(1024):
                        connectionSocket.sendall(chunk)
                connectionSocket.shutdown(SHUT_WR)
                continue
```

makeResponseHeader() 함수로 응답헤더 출력코드 간결하게 만들기

```
1            response = ''
2            if req.url == '/user/list':
3                response = makeResponseHeader(HTTPStatusCode.OK,
                     HttpContentType.APPLICATION_JSON)
4                response += json.dumps(getUserList())
5            elif req.url == '/':
6                response = makeResponseHeader(HTTPStatusCode.OK,
                     HttpContentType.TEXT_HTML)
7                response += '<html><body>Hello World<br /><img
                     src="/google.png" /></body></html>\n'
8            elif req.url == '/google.png':
9                response = makeResponseHeader(HTTPStatusCode.OK,
                     HttpContentType.IMAGE_PNG)
10               connectionSocket.sendall(response.encode('utf-8'))
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 2 ii

```
11                  with open('google.png', 'rb') as f:
12                      while chunk := f.read(1024):
13                          connectionSocket.sendall(chunk)
14                  connectionSocket.shutdown(SHUT_WR)
15                  continue
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 i

HTTP 요청에 대한 처리를 함수로 분리해봅시다.
현재까지 작성한 코드는 처리해야할 URL이 많아지면 createServer() 함수가 매우 길어져 소스코드 관리가 어려워집니다.
각 if문에 있는 처리 로직을 함수로 분리하고 createServer() 함수에서는 함수를 호출하도록 수정해봅시다.

```
1 response = ''
2 if req.url == '/user/list':
3     response = makeResponseHeader(HTTPStatusCode.OK,
          HttpContentType.APPLICATION_JSON)
4     ...
5 elif req.url == '/':
6     response = makeResponseHeader(HTTPStatusCode.OK,
          HttpContentType.TEXT_HTML)
7     ...
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 ii

```
8  elif req.url == '/google.png':
9      response = makeResponseHeader(HTTPStatusCode.OK,
           HttpContentType.IMAGE_PNG)
10     ...
11     #  너무 길어짐
```

각 함수들은 요청 URL에 대한 처리 로직을 수행하고 클라이언트로 보낼 응답을 bytes 형태로 반환하도록 구현합니다.

createServer() 함수에서는 URL마다 지정된 함수를 호출하여 반환받은 bytes값을 socket으로 전송하도록 수정합니다.

함수의 형태

```
1 def function_name(request: HTTPRequest) -> bytes:
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 i

URL	함수명
/	handler_home(request: HTTPRequest) →bytes:
/user/list	handler_user_list(request: HTTPRequest) →bytes:
/google.png	handler_google_png(request: HTTPRequest) →bytes:
/google	handler_google(request: HTTPRequest) →bytes:
404	handler_404(request: HTTPRequest) →bytes:

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 i

```
1  # /
2  def handler_home(request: HTTPRequest) -> bytes:
3      response = makeResponseHeader(HTTPStatusCode.OK,
           HttpContentType.TEXT_HTML)
4      response += '<html><body>Hello World<br /><img src="/google.png"
           /></body></html>\n'
5      return response.encode('utf-8')
6
7  # /user/list
8  def handler_user_list(request: HTTPRequest) -> bytes:
9      response = makeResponseHeader(HTTPStatusCode.OK,
           HttpContentType.APPLICATION_JSON)
10     response += json.dumps(getUserList())
11     return response.encode('utf-8')
12
13 # /google
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 ii

```
14 def handler_google(request: HTTPRequest) -> bytes:
15     response = makeResponseHeader(HTTPStatusCode.SEE_OTHER,
           HttpContentType.TEXT_HTML, {'Location': 'https://www.google.com'})
16     return response.encode('utf-8')
17
18 # /google.png
19 def handler_google_png(request: HTTPRequest) -> bytes:
20     response = makeResponseHeader(HTTPStatusCode.OK,
           HttpContentType.IMAGE_PNG).encode('utf-8')
21     with open('google.png', 'rb') as f:
22         response += f.read()
23     return response
24
25 def hander_404(request: HTTPRequest) -> bytes:
26     response = makeResponseHeader(HTTPStatusCode.NOT_FOUND,
           HttpContentType.TEXT_HTML)
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 iii

```
27      response += '<html><body>404 Not Found</body></html>\n'
28      return response.encode('utf-8')
```

while문 내의 코드를 아래와 같이 수정합니다.

```
while True:
    (connectionSocket, addr) = serverSocket.accept() # Blocking
    print('Connection received from ', addr)
    request = connectionSocket.recv(4096).decode('utf-8')
    print(request)
    req = parseRequest(request)
    if req is None or req.url is None:
        connectionSocket.shutdown(SHUT_WR)
        continue

    resp = None
    if req.url not in arPath:
        resp = hander_404(req)
    elif req.url == '/google':
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 ii

```
15          resp = handler_google(req)
16      elif req.url == '/user/list':
17          resp = handler_user_list(req)
18      elif req.url == '/':
19          resp = handler_home(req)
20      elif req.url == '/google.png':
21          resp = handler_google_png(req)
22
23      if resp is not None:
24          chunk_size = 1024
25          arChunks = [resp[i:i+chunk_size] for i in range(0, len(resp),
                chunk_size)]
26          for chunk in arChunks:
27              connectionSocket.sendall(chunk)
28      connectionSocket.shutdown(SHUT_WR)
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3

지금까지 수정한 소스코드는 다음 링크에서 확인할 수 있습니다.
https://github.com/song9063/PNU-Bootcamp-2025/blob/main/Day1/Day1-2.7-http-server-main.py

지금까지의 리팩토링을 통해 로직을 분리하여 while문을 간결하게 만들었습니다.
지금부터는 GET요청 주소로 전달되는 쿼리스트링을 파싱하는 방법에 대해 알아보겠습니다.
쿼리스트링은 URL 뒤에 ?를 붙이고 key=value 형태로 전달되는 문자열입니다.

```
http://localhost:8080/user/list?page=1&size=10
```

https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods/GET

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3

지금 작성한 코드는 요청 URL 문자열을 equal 조건(==)으로 검사하고 있기때문에
만약 URL 뒤에 다른 문자열이 포함되면 정상적으로 처리되지 않습니다.
HTTPRequest 클래스를 아래와 같이 수정합니다.

```
@dataclass
class HTTPRequest:
    method: HTTPMethod
    url: str
    path: str
    query: dict = None
```

url에서 path와 쿼리파라미터를 분리하는 함수를 만듭니다.

```
1  def parseQuery(url: str) -> tuple[str, dict | None]:
2      path = url
3      query = {}
4      match = re.search(r'(\/[\w\-.\/]*)+([/#]*)([0-9a-zA-Z\-]*)\?*(.*)',
           url)
5      if match:
6          print(len(match.groups()))
7          path = match.group(1)
8          queryStr = match.group(4)
9          if path[-1] == '/':
10             path = path[:-1]
11         if path == '':
12             path = '/'
13         for q in queryStr.split('&'):
```

```
14              arQs = q.split('=')
15              if len(arQs) == 2:
16                  query[arQs[0]] = arQs[1]
17      return path, query
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 i

요청이 들어오면 URL에서 path와 query를 분리하여 HTTPRequest 객체를 생성하도록 parseRequest() 함수를 수정합니다.

```
1  def parseRequest(requests: str) -> HTTPRequest | None:
2      if len(requests) < 1:
3          return None
4      arRequests = requests.split('\n')
5      for line in arRequests:
6          match =
               re.search(r'\b(GET|POST|DELETE|PUT|PATCH)\b\s+(.*?)\s+HTTP/1.1',
               line)
7          if match:
8              method = HTTPMethod(match.group(1))
9              url = match.group(2)
10             path, query = parseQuery(url)
11             try:
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 ii

```
12                 return HTTPRequest(method, url, path, query)
13             except ValueError:
14                 return None
15     return None
```

while 문 내의 코드를 아래와 같이 수정합니다.

```
while True:
    (connectionSocket, addr) = serverSocket.accept() # Blocking
    print('Connection received from ', addr)

    request = connectionSocket.recv(4096).decode('utf-8')
    print(request)
    req = parseRequest(request)
    if req is None or req.url is None:
        connectionSocket.shutdown(SHUT_WR)
        continue
    print(req)
    resp = None

    if not req.path.startswith(tuple(arPath)):
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 ii

```
15          resp = hander_404(req)
16      elif req.path == '/google':
17          resp = handler_google(req)
18      elif req.path == '/user/list':
19          resp = handler_user_list(req)
20      elif req.path == '/':
21          resp = handler_home(req)
22      elif req.path == '/google.png':
23          resp = handler_google_png(req)
24
25      if resp is not None:
26          chunk_size = 1024
27          arChunks = [resp[i:i+chunk_size] for i in range(0, len(resp),
                chunk_size)]
28          for chunk in arChunks:
29              connectionSocket.sendall(chunk)
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 iii

```
30      connectionSocket.shutdown(SHUT_WR)
```

https://github.com/song9063/PNU-Bootcamp-2025/blob/main/Day1/Day1-2.8-http-server-main.py

웹 브라우저 주소창에 아래와 같이 쿼리스트링을 추가하여 요청을 보내봅시다.
http://localhost:8080/user/list?page=1category=usa

서버 로그를 확인해보면 아래와 같이 쿼리파라메터가 dict에 들어가있는 것을 확인할 수 있습니다.

```
HTTPRequest(method=<HTTPMethod.GET: 'GET'>,
    url='/user/list?page=1&category=usa', path='/user/list',
    query={'page': '1', 'category': 'usa'})
```

while문 내의 코드를 더 간결하게 만들기 위해 URL에 대한 처리로직을 별도의 함수로 분리합니다.

```
def handle_request(request: HTTPRequest) -> bytes:
    resp = None
    print(f'Handle request: {request.path}')
    if request.path == '/google':
        resp = handler_google(request)
    elif request.path == '/user/list':
        resp = handler_user_list(request)
    elif request.path == '/':
        resp = handler_home(request)
    elif request.path == '/google.png':
        resp = handler_google_png(request)
    else:
        resp = hander_404(request)
    return resp
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 3 ii

while문 내의 if, elif 코드들을 handle_request() 함수로 변경합니다.
(createServer 함수 코드)

```
def createServer():
    serverSocket = socket(AF_INET, SOCK_STREAM)
    try:
        serverSocket.bind(('localhost', 8080))
        serverSocket.listen()
        while True:
            (connectionSocket, addr) = serverSocket.accept() # Blocking
            print('Connection received from ', addr)

            request = connectionSocket.recv(4096).decode('utf-8')
            # print(request)
            req = parseRequest(request)
            if req is None or req.url is None:
                connectionSocket.shutdown(SHUT_WR)
                continue
```

```
            resp = handle_request(req)

            if resp is not None:
                chunk_size = 1024
                arChunks = [resp[i:i+chunk_size] for i in range(0, len(resp),
                    chunk_size)]
                for chunk in arChunks:
                    connectionSocket.sendall(chunk)
            connectionSocket.shutdown(SHUT_WR)

    except KeyboardInterrupt:
        print('\nShutting down the server\n')
        serverSocket.close()
    except Exception as e:
        print('Unexpected error:', e)
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4 i

HTML 코드가 서버 프로그램 내에 포함되면 코드 관리가 어려워지며 콘텐츠 내용 수정시마다 서버 프로그램을 재시작해야하는 문제가 있습니다.
HTML파일을 별도의 폴더로 관리하고 요청에 맞는 파일을 읽어서 응답으로 보내도록 서버 프로그램을 수정해봅시다.
아래와 같이 폴더를 생성합니다.

```
/
├── main.py
└── html
    ├── google.png 다로로드
    └── index.html
```

index.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <h1>Hello World</h1>
        <img src="/google.png" />
    </body>
</html>
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4

handler_home() 함수에서 index.html 파일을 읽어서 응답으로 보내도록 수정합니다. google.png 이미지 파일의 위치도 이동되었으므로 경로를 수정합니다.

```
def handler_home(request: HTTPRequest) -> bytes:
    response = makeResponseHeader(HTTPStatusCode.OK,
        HttpContentType.TEXT_HTML)
    with open('html/index.html', 'r') as f:
        response += f.read()
    return response.encode('utf-8')

def handler_google_png(request: HTTPRequest) -> bytes:
    response = makeResponseHeader(HTTPStatusCode.OK,
        HttpContentType.IMAGE_PNG).encode('utf-8')
    with open('html/google.png', 'rb') as f:
        response += f.read()
    return response
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4 i

html, png 등 파일을 불러오는 코드는 파일명만 제학하고 모두 동일한 로직을 가집니다. 아래와 같이 파일을 불러오는 함수를 만들어서 사용하면 코드 중복을 줄일 수 있습니다.

```
import os
def read_file(file_path: str) -> str:
    pathToRead = os.path.join('html', file_path)
    with open(pathToRead, 'rb') as f:
        return f.read()

def handler_home(request: HTTPRequest) -> bytes:
    response = makeResponseHeader(HTTPStatusCode.OK,
        HttpContentType.TEXT_HTML).encode('utf-8')
    response += read_file('index.html')
    return response
```

```
def handler_google_png(request: HTTPRequest) -> bytes:
    response = makeResponseHeader(HTTPStatusCode.OK,
        HttpContentType.IMAGE_PNG).encode('utf-8')
    response += read_file('google.png')
    return response
```

주의사항: read_file 함수는 파일을 바이너리 모드('rb')로 읽어서 bytes로 반환합니다. makeReponseHeader 함수는 str 형태로 반학하기 때문에 response 변수에 각각 다른 형식의 문자열을 이어서 넣을 수 없습니다.
따라서 makeReponseHeader 함수의 리턴값을 encode함수로 bytes로 변환 후 response 변수에 넣도록 수정해야합니다.

mimetypes 모듈을 사용하여 불러올 파일의 Content-Type을 자동으로 설정하도록 수정해봅시다.

```
import mimetypes

def read_file(file_path: str) -> tuple[str, str] | None:
    pathToRead = os.path.join('html', file_path)
    mime_type, _ = mimetypes.guess_type(pathToRead)
    with open(pathToRead, 'rb') as f:
        return f.read(), HttpContentType(mime_type)
    return None

def handler_home(request: HTTPRequest) -> bytes:
    data, mimeType = read_file('index.html')
    response = makeResponseHeader(HTTPStatusCode.OK,
        mimeType).encode('utf-8')
```

```
    response += data
    return response

def handler_google_png(request: HTTPRequest) -> bytes:
    data, mimeType = read_file('google.png')
    response = makeResponseHeader(HTTPStatusCode.OK,
        mimeType).encode('utf-8')
    response += data
    return response
```

서버 오류 응답하기

앞서 찾을 수 없는 URL에 대한 응답을 404 Not Found로 응답하도록 했습니다.
404 오류 응답 HTML 내용을 파일로 만들어 python 코드와 분리합니다.
그리고 서버 프로그램에서 오류가 발생하면 500 응답을 보내고 오류 내용을 담은 HTML 파일을 응답으로 보내도록 수정합니다.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/500

```
/
├── main.py
└── html
    ├── google.png
    ├── index.html
    ├── 404.html
    └── 500.html
```

1. 404 응답 콘텐츠를 HTML 파일로 분리하기

Listing 15: html/404.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>404 Not Found</title>
    </head>
    <body>
        404 Not Found
    </body>
</html>
```

2. 500 응답 HTML 만들기

Listing 16: html/500.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>500 Internal Server Error</title>
    </head>
    <body>
        500 Internal Server Error
    </body>
</html>
```

3. 404 핸들러 함수 수정하기, 500 핸들러 함수 작성하기

```
def hander_404(request: HTTPRequest) -> bytes:
    data, mimeType = read_file('404.html')
    response = makeResponseHeader(HTTPStatusCode.NOT_FOUND,
        mimeType).encode('utf-8')
    response += data
    return response

def hander_500(request: HTTPRequest) -> bytes:
    data, mimeType = read_file('500.html')
    response = makeResponseHeader(HTTPStatusCode.SERVER_ERROR,
        mimeType).encode('utf-8')
    response += data
    return response
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4

read_file 함수에서 파일을 찾을 수 없는 경우 None을 반환하도록 수정합니다.

```
def read_file(file_path: str) -> tuple[str, HttpContentType]:
    if len(file_path) > 0 and file_path[0] == '/':
        file_path = file_path[1:]
    pathToRead = os.path.join('html', file_path)
    mime_type, _ = mimetypes.guess_type(pathToRead)
    try:
        with open(pathToRead, 'rb') as f:
            return f.read(), HttpContentType(mime_type)
    except FileNotFoundError:
        pass
    return None, None
```

handler_* 함수 내에서 read_file 함수의 반환값이 None인 경우 None을 응답돌도록 수정합니다.

```
def handler_home(request: HTTPRequest) -> bytes | None:
    data, mimeType = read_file('index.html')
    if data is None:
        return None
    response = makeResponseHeader(HTTPStatusCode.OK,
        mimeType).encode('utf-8')
    response += data
    return response

def handler_google_png(request: HTTPRequest) -> bytes:
    data, mimeType = read_file('google.png')
    if data is None:
        return None
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4 ii

```
    response = makeResponseHeader(HTTPStatusCode.OK,
        mimeType).encode('utf-8')
    response += data
    return response

def hander_404(request: HTTPRequest) -> bytes | None:
    data, mimeType = read_file('404.html')
    if data is None:
        return None
    response = makeResponseHeader(HTTPStatusCode.NOT_FOUND,
        mimeType).encode('utf-8')
    response += data
    return response

def hander_500(request: HTTPRequest) -> bytes:
    data, mimeType = read_file('500.html')
```

```
    if data is None:
        mimeType = HttpContentType.TEXT_HTML
        data = '<html><body>500 Internal Server
            Error</body></html>\n'.encode('utf-8')
    response = makeResponseHeader(HTTPStatusCode.SERVER_ERROR,
        mimeType).encode('utf-8')
    response += data
    return response
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4 i

handler_* 함수들의 응답값이 None인 경우 500 응답을 보내도록 createServer 함수의 while 문을 수정합니다.

```
resp = handle_request(req)
if resp is None:
    resp = hander_500(req)
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4

jpg, png와 같은 이미지 파일 요청이 들어오면 현재는 파일 1개마다 1개의 핸들러 함수를 만들어야합니다.
이러한 방식은 파일이 많아질수록 코드가 길어지고 관리가 어려워집니다.
만약 요청 URL의 끝이 .png 또는 .jpg로 끝난다면 해당 파일을 읽어서 응답으로 보내도록 수정해봅시다.

```
/
├── main.py
└── html
    ├── google.png 다로로드
    ├── Linux.jpg 다로로드
    └── index.html
```

아래와 같이 handler_image 함수를 추가하고
handle_request 함수를 수정합니다.

```
class HttpContentType(Enum):
    TEXT_HTML = 'text/html'
    APPLICATION_JSON = 'application/json'
    IMAGE_PNG = 'image/png'
    IMAGE_JPEG = 'image/jpeg'
    IMAGE_JPG = 'image/jpg'

def handler_image(request: HTTPRequest) -> bytes:
    data, mimeType = read_file(request.path)
    if data is None:
        return None
    response = makeResponseHeader(HTTPStatusCode.OK,
        mimeType).encode('utf-8')
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4 ii

```
    response += data
    return response

def handle_request(request: HTTPRequest) -> bytes:
    resp = None
    print(f'Handle request: {request.path}')

    if request.path.endswith('.png') or request.path.endswith('.jpg'):
        return handler_image(request)

    if request.path == '/google':
        resp = handler_google(request)
    elif request.path == '/user/list':
        resp = handler_user_list(request)
    elif request.path == '/':
        resp = handler_home(request)
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4 iii

```
    elif request.path == '/google.png':
        resp = handler_google_png(request)
    else:
        resp = hander_404(request)
    return resp
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4

웹 브라우저 주소창에 http://localhost:8080/Linux.jpg를 입력하여 이미지 파일을 요청해봅시다.
주소창에 http://localhost:8080/google.png도 입력해봅시다.

이제 handler_google_png 함수는 사용하지 않으므로 삭제해도됩니다.

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4

html/index.html 파일을 아래와 같이 수정하여 두개의 이미지를 출력하도록 합니다.

```
<!DOCTYPE html>
<html>
    <head>
        <title>Home</title>
    </head>
    <body>
        <h1>Hello World</h1>
        <img src="/google.png" />
        <img src="/Linux.jpg" />
    </body>
</html>
```

2.2 간단한 HTTP 서버 구현하기 - 리팩토링 4