C 언어를 이용한 웹서버 구현

2020.06.05

소프트웨어학부 컴퓨터전공
2016003727
윤주경

목차

제 1 장

이번 프로젝트의 목표는 C 언어만을 이용해서 서버와 클라이언트의 동작 방식에 대해 이해하고 실제로 구현해보는 것이다. 그 동안 몇차례 서버와 클라이언트에 대해 공부하고 실제로 결과물도 만들었지만 이미 존재하는 API 나 프레임워크를 이용해 만들었기 때문에 완벽하게 이해하고 만들었다고 보기에는 무리가 있다. php 와 java script 를 이용해 웹서버를 만들어 보았지만 C 언어로는 처음이기 때문에 기본적인 개념에 대해 알아가는 것이 중요하다.

하나하나 공부하면서 웹서버의 기능을 확장시키는 방식으로 프로젝트를 진행하였기 때문에 github 를 이용해 프로젝트를 업로드 하였다. 카피문제를 피하기 위해 최근에 무료가 된 github 비공개 저장소를 이용했다. 저장소는 과제 제출기간이 끝나고 public 으로 변경한다.

https://github.com/JuKyYoon/ComputerNetwork_Project

제 1 절 C 언어를 이용한 Socket 통신

프로젝트 예제 파일로 서버와 클라이언트를 C 언어로 구현한 코드가 주어졌다. 이 코드를 바탕으로 서버와 클라이언트가 어떤 순서와 방식으로 통신하는지 알아본다. 뒤에 제작할 코드와 겹치는 부분이 많기 때문에 여기서 겹치는 부분을 설명한다. 처음 공부하는 입장에서 코드를 다시 작성하였기 때문에 좋은 코드라고 할 수는 없지만 순서대로 진행하기 위해 코드를 따로 정리하지는 않았다.

제 1 항 서버 측 코드

연결을 위한 서버의 코드는 크게 5 단계로 나눠진다. 소켓 생성, 소켓에 주소 할당, 클라이언트 요청 대기, 통신, 종료.

서버 쪽 코드를 먼저 살펴보면 우선 맨 처음으로 main 함수의 명령인수의 숫자를 검사한다. 명령인수로 포트번호를 알려주기 때문에 명령인수가 적절치 않으면 프로그램을 종료해야 한다.

```
1. if (argc < 2) { // 명령 인수 숫자가 적다
2.         fprintf(stderr, "Argument is not valid.\n"); // 에러문 출력
3.         exit(0); // 서버 종료
4. }
```

위 조건문을 지나면 받은 인자를 atoi 함수를 이용해 입력받은 Port 번호를 정수형 변수로 변환해 준다.

```
1. int port_number = atoi(argv[1]);
2. if( port_number < 0 ){ error_print("Not valid Port"); }
```

그리고 <sys/socket.h> 헤더파일에 명시되어 있는 socket 함수를 이용해서 소켓을 생성한다. 이 때 생성한 소켓은 소켓 디스크립터로 접근 할 수 있다. socket 함수는 3 개의 인자를 받는데 순서대로 통신 영역, 사용 할 전송 방식, 사용되는 프로토콜을 의미한다. 통신 영역은 IPv4 를 사용하기 위해 PF_INET, 웹 서버이기 때문에 TCP 프로토콜을 사용하기 위해 SOCK_STREAM, 마지막으로는 IPPROTO_TCP 을 인자로 넣어 소켓을 생성한다.

```
1. int client_socket_fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
2. if( client_socket_fd < 0 ){ // 소켓을 만들기 실패하면 종료
3.         error_print("Fail to open socket.\n");
4. }
```

만들어진 소켓에 주소를 할당하기 위해 bind 함수를 이용해준다. 그 전에 소켓 주소를 따로 정의해주기 위해 sockaddr_in 구조체를 사용해주었다. 클라이언트의 요청을 받기 위해 클라이언트 주소도 필요하다.

```
1. struct sockaddr_in server_address, client_addr;
2. // 소켓 주소를 담는 구조체, 서버와 클라이언트 2 개 필요하다.
3. memset (&server_address, 0, sizeof(server_address));
4. // 구조체를 0 으로 초기화
5. server_address.sin_family = AF_INET;           // IPv4 로 설정
6. server_address.sin_port = htons(port_number); // htons 함수를 통해 포트
   번호를 네트워크 바이트 순서로 변환해준다.
7. server_address.sin_addr.s_addr = htonl(INADDR_ANY);
8. // 주소를 자동으로 대입해주는 INADDR_ANY 사용, long 형이라 htonl 사용
```

bind 함수를 통해 서버측 소켓에 서버의 주소를 배정해준다.

```
1. if ( bind( server_socket_fd, (struct sockaddr *)&server_address, size
   of(server_address) ) < 0)
2. {
3.         error_print("error to bind");
4. }
5.        // 첫번째 인자 : 서버 소켓 디스크립터
6.        // 두번째 인자 : 서버의 주소 정보
7.        // 세번째 인자 : 서버의 주소 정보의 크기
```

bind 까지 마치고 나면 클라이언트의 요청을 받아줘야 한다. listen 함수를 통해 서버측 소켓의 연결 대기열을 만들어서 서버는 대기상태가 된다.

```
1. listen(server_socket_fd, 7);
```

client 의 요청이 들어오면 클라이언트의 주소를 이용해서 연결을 허용해준다. 이 때 연결된 서버와 클라이언트는 accept 함수의 성공 시 생성된 새로운 소켓을 이용해 통신한다.

```
1. socklen_t req_client = sizeof(client_addr);
```

이 때 서버와 클라이언트간 메시지를 주고 받기 위해 두개의 문자열을 생성하였다.

```
1. char *request_msg; // 클라이언트에서 받은 메시지
2. request_msg = malloc( (size_t)MSG_SIZE );
3. char *response_msg; // 클라이언트로 보낼 메시지
4. response_msg = malloc( (size_t)MSG_SIZE );
```

accept(), write(), read()를 이용해서 클라이언트와 통신한다.

```
1. while(1){
2.     int socket_fd = accept( server_socket_fd, (struct sockaddr
   *)&client_addr, &req_client);
3.     if( socket_fd < 0){
4.         error_print("Fail to accpet");
5.     }
6.     if( read(socket_fd, request_msg, MSG_SIZE) < 0 ){
7.           // client 에서 request 를 받는다.
8.           error_print("Fail to read");
9.     }
10.      printf("to server from client : %s", request_msg);
11.      printf("enter the msg : ");
12.      fgets(request_msg, MSG_SIZE, stdin ); // 메시지 입력을 받는다.
13.      if( write(socket_fd, response_msg, strlen(response_msg))< 0 ) {
14.      // write 함수를 이용해 소켓에 메시지를 작성한다.
15.          error_print("Fail to writing to socket.");
16.      }
17.      close(socket_fd); // 연결된 소켓을 닫는다.
18.  }
```

통신이 끝나면 할당된 메모리를 해제하고 열려있는 디스크립터를 닫아준다.

```
1. free(response_msg);
2. free(request_msg);
3. close(server_socket_fd);
4. return 0;
```

* 에러문 출력 함수

```
1. void error_print(char *error_msg){
2.     perror(error_msg);
3.     exit(1);
4. }
```

제 2 항 클라이언트 측 코드

연결을 위해 서버에 요청을 보내는 클라이언트측 코드는 크게 4 가지 단계로 나눠진다. 소켓 생성, 소켓 연결, 통신, 종료.

서버 측과 마찬가지로 맨 처음에는 main 함수의 명령인수를 검사한다. 서버와 달리 machine name 과 포트번호 2 가지 인수를 받는다. 이 때 편의를 위해 machine name 은 localhost 로 정의한다. 포트번호는 서버와 동일한 번호를 사용한다.

```
1. if (argc < 3) { // 인자 개수가 부족하다.
2.     fprintf(stderr, "Argument is out of quntity.\n");
3.     // 에러문 출력 후 클라이언트 종료
4.     exit(0);
5. }
```

위 조건문을 지나면 받은 인자를 atoi 함수를 이용해 입력 받은 포트 번호를 정수형 변수로 변환해 준다.

```
1. int port_number = atoi(argv[2]);
2. if( port_number < 0 ){ error_print("Not valid Port"); }
```

서버 측과 동일한 프로토콜을 사용하는 소켓을 생성한다.

```
1. int client_socket_fd = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
2. if( client_socket_fd < 0 ){
3.     error_print("Fail to open socket.\n");
4. }
```

그 다음에는 서버 접속을 위해 hostent 구조체를 정의해 클라이언트의 정보를 저장한다. 도메인 주소로 IP 주소를 얻기 위해 gethostbyname()을 사용하였다.

```
1. struct hostent *server;
2. if ( (server = gethostbyname(argv[1])) == NULL) {
3.     fprintf(stderr, "Host error\n");
4.     exit(0);
5. }
```

서버와 마찬가지로 socketaddr_in 구조체를 사용해서 소켓 주소를 저장한다. 이 때 bind() 는 따로 사용하지 않는다. 왜냐하면 서버는 어떤 클라이언트가 접근해 올 지 몰라 클라이언트의 정보를 미리 알 필요가 없다. 그래서 클라이언트에서는 고정된 포트 번호를 배정하는 bind 과정을 따로 할 필요가 없다. 이와 달리 서버는 항상 고정된 포트 번호가 필요하기 때문에 bind 가 필요하다.

```
1. struct sockaddr_in server_address; // 소켓 주소를 담는 구조체
2. memset (&server_address, 0, sizeof(server_address));
3. // 구조체를 0 으로 초기화
4. server_address.sin_family = AF_INET; // IPv4 로 설정
5. server_address.sin_port = htons(port_number);
6. // htons 함수를 통해 포트번호를 네트워크 바이트 순서로 변환해준다.
7. server_address.sin_addr.s_addr = htonl(INADDR_ANY);
8. // 서버주소에 IP 저장
```

위 과정이 끝나면 클라이언트는 서버와 통신할 준비가 되었다. Connect 함수를 통해 서버에 연결 요청을 보낸다. 이 때 connect 함수의 인자는 순서대로 클라이언트 소켓 디스크립터, 서버의 주소 정보, 서버 주소 정보의 크기이다.

```
1. if( connect( client_socket_fd, (struct sockaddr *)&server_address,
   sizeof(server_address)) < 0){
2.     error_print("Fail to connect");
3. }
```

연결이 되었다면 통신하기 위한 두 개의 문자열을 생성한다.

```
1. char *request_msg; // 서버로 보낼 메시지
2. request_msg = malloc( (size_t)MSG_SIZE );
3. char *response_msg; // 서버에서 받은 메시지
4. response_msg = malloc( (size_t)MSG_SIZE );
```

write()와 read()를 이용해 서버와 통신 할 수 있다.

```
1. while(1){
2.     printf("enter the msg : ");
3.     fgets(request_msg, MSG_SIZE, stdin ); // 메시지 입력을 받는다.
4.     if ( write(client_socket_fd, request_msg, strlen(request_msg)) <
   0 ) {
5.     // write 함수를 이용해 소켓에 메시지를 작성한다.
6.         error_print("Fail to writing to socket.");
7.     }
8.
9.
10.
11.      if ( read( client_socket_fd, response_msg, MSG_SIZE) <
   0 ){//소켓 디스크립터에서 읽기
12.          error_print("Fail to reading from socket.");
13.      }
14.      printf("to client from server : %s", response_msg);
15.  }
```

통신이 끝나면 메모리 해제와 디스크립터를 닫아 프로그램을 종료한다.

```
1. free(response_msg);
2. free(request_msg);
3. close(client_socket_fd);
4. return 0;
```

제 2 절 사용 도구

위 코드를 작성하면서 여러 개의 구조체와 함수, 헤더파일을 사용하였다. 해당 내용에 대해 간략하게 요약해본다.

#헤더파일

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <sys/types.h>
#include <netdb.h>
#include <string.h>
#include <unistd.h>
```

#hostent 구조체

```
struct hostent {
        char *h_name; //도메인 이름
        char **h_aliases; //다른 도메인 이름들
        int h_addrtype; //주소정보 체계(IPv4: AF_INET, IPv6: AF_INET6)
        int h_length; // 주소의 길이
        char **h_addr_list; //도메인 이름에 대한 IP 주소가 정수 형태로 반환될 때 이
멤버 변수를 이용
};
```

#sockaddr_in 구조체

```
struct sockaddr_in {
short sin_family; // 주소 체계: 항상 AF_INET 으로 설정 (IPv4)
AF_INET u_short sin_port; // 16 비트 포트 번호, network byte order
struct in_addr sin_addr; // 32 비트 IP 주소
char sin_zero[8]; // 전체 크기를 16 비트로 맞추기 위한 dummy
};

struct in_addr {
```

```
        u_long s_addr; // 32 비트 IP 주소를 저장 할 구조체, network byte order
};
```

CPU 에서는 데이터를 저장하고 해석하는 방식이 두 가지가 있다. 그래서 네트워크에 대한 데이터를 저장하거나 가져올 때는 해당 형식에 맞춰야 한다. 네트워크에서는 Big endian 형식을 사용한다. 형식을 맞추기 위해 short, long 형에 맞춰 4 가지 함수가 있다.

```
unsigned short htons(unsigned short);
unsigned short ntohs(unsigned short);
unsigned long htonl(unsigned long);
unsigned long ntohl(unsigned long);
```

INADDR_ANY

서버의 IP 주소를 자동으로 할당해주는 API 이다. 컴퓨터 NIC 가 여러 개일 경우에도 지원해준다.

IPv4 IPv6

네트워크 계층의 프로토콜이다. IPv4 프로토콜의 경우 주소 수가 전세계 컴퓨터에 비해 적기 때문에 최근에는 IPv6 을 사용한다. 이번 프로젝트에서는 127.0.0.1 로 자기 자신 IP 를 서버로 이용했기 때문에 IPv4 로 소켓을 설정해 주었다.

#브라우저

해당 프로젝트에 사용되는 브라우저는 Google Chrome 이다.

제 3 절 HTTP response 와 HTTP request

프로젝트의 요구사항을 만족시키기 위해 클라이언트를 브라우저로 변경하였다. Shell 에서 이루어졌던 통신과 달리 브라우저와 서버간 통신이기 때문에 브라우저를 통해 서버를 접속하기만 해도 HTTP request 를 보낸다. 이 request 는 브라우저에서 보내오는 것이기 때문에 위에서 작성한 코드를 바꾸지 않고도 메시지를 볼 수 있다. 내용은 아래와 같다.

```
GET / HTTP/1.1
Host: localhost:8080
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/83.0.4103.61 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng
,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Sec-Fetch-Site: none
Sec-Fetch-Mode: navigate
Sec-Fetch-User: ?1
```

```
Sec-Fetch-Dest: document
Accept-Encoding: gzip, deflate, br
Accept-Language: ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7
```

HTTP/1.1 은 HTTP/1.0 보다 많은 필드가 추가되었다.
첫번째 줄은 request-LINE 으로 서버에 HTTP/1.1 로 Get 메소드를 이용하여 / 주소에 요청을 보낸다는 뜻이다. Method, request-line 으로 이루어져 있다.
다음줄부터는 리퀘스트의 헤더의 내용이다.

HOST : 요청하는 호스트에 대한 호스트 명 및 아이피 주소이다. 프로젝트에서는 Localhost 를 이용해 8080 포트로 접속하였다. 전체 URI 가 필요하여 localhost:8080 이 적혀있다.

Connection : 클라이언트와 서버간 연결의 정보이다. HTTP 메시지를 보내고 다시 받기위해 연결을 유지해야 하기 때문에 keep-alive 로 되어 있다.

Upgrade-Insecure-Requests : 콘텐츠가 암호화되고 인증되었는지에 대한 유저의 선호도이다. 1 이면 서버는 사용자에게 안전한 콘텐츠를 제공해야 한다. 즉 HTTPS 로 요청 파일을 보내줘야 한다.

User-agent 는 리퀘스트를 보낸 유저의 정보를 담고있다. 즉. 브라우저의 정보가 담겨져 있다. Mozilla/5.0 기반으로 KTHML 을 사용한 Chrome 이다. 뒤의 Safari 는 비슷한 버전을 의미한다.

Accept 는 클라이언트가 어떤 Content 를 이해 할 수 있는지 나타낸다. 해당 줄은 MIME type 으로 표현되어 있다. 파일종류/확장자로 표현되어 있는데 몇몇은 그렇지 않다. */*는 어떤 미디어 타입도 가능하다는 뜻이다

q 는 품질을 의미한다 q=0.8 의 의미는 품질이 80%이상인 경우에만 해당 내용을 선호한다는 뜻이다.
v 는 버전을 의미한다.

Accept-encoding 과 Accept-Language 는 클라이언트가 원하는 인코딩 방식과 원하는 가능한 언어이다.

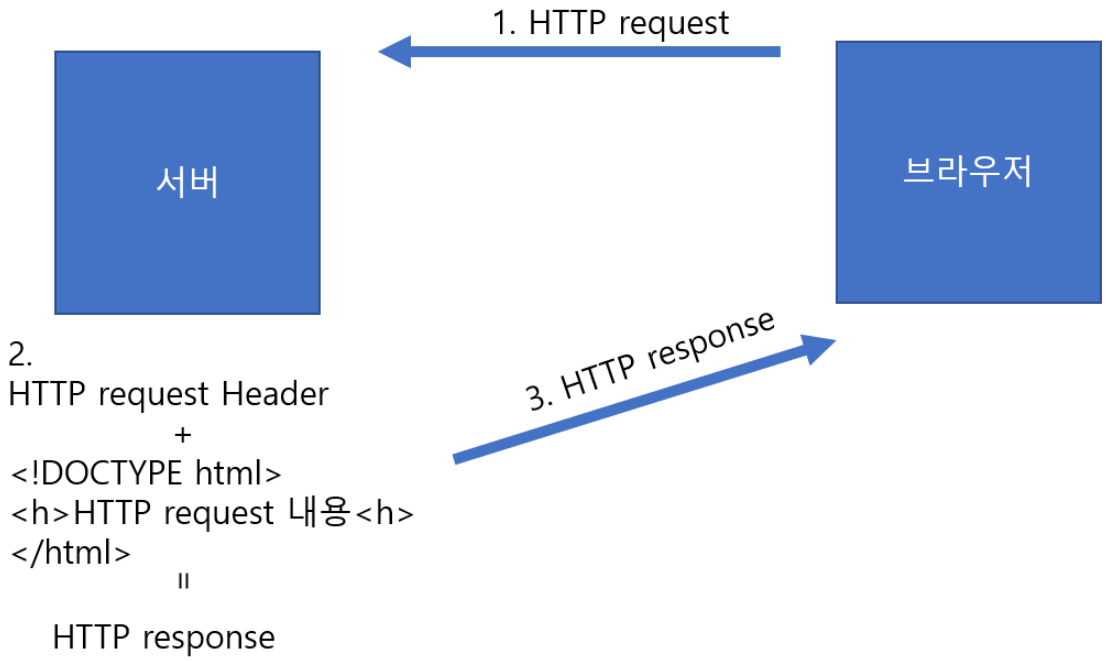
Sec-fetch-site 는 자원의 origin 과 요청 initiator 의 origin 의 관계를 나타낸다
Sec-fetch-mode 는 요청 모드를 의미한다.
Sec-fetch-user 는 탐색 요청이 클라이언트에 의해 활성화 되었는지 알려준다. ?1 인 경우 탐색 요청이 유저 이외의 것에 의해 트리거 되었다,
Sec-fetch-dest 는 요청의 목적지를 의미한다. 이 데이터가 document 형태로 사용될 거라는 것을 의미한다.

제 2 장

1 장의 코드를 통해 C 언어를 이용한 소켓 프로그래밍에 대해 이해하였다. 2 장에서는 프로젝트의 요구사항을 만족시키기 위해 위 코드를 확장 시켰다. 클라이언트는 브라우저가 대신하기 때문에 서버 측 코드만 수정해주면 되었다. Shell 에서 이루어졌던 통신과 달리 브라우저로 통신하기 때문에 서버는 단순히 데이터만 보내면 안된다. 그 데이터에 헤더파일을 붙여 HTTP 통신을 해야 한다. 브라우저는 GET method 만을 이용하기 때문에 최소한의 헤더만 사용하였다.

제 1 절 서버 디자인

브라우저가 HTTP request 를 보냈으니 서버는 HTTP response 를 보내야 한다. response 에 대한 헤더는 브라우저의 개발자 콘솔에서 볼 수 있다. 우선 화면에 출력해야 하므로 content-type 은 text/html 이다. 그리고 content-length 를 지정해 줘야 하는데, C 언어 내장 함수로 간단히 구할 수 있다. 단순히 response message 만 출력하는 것이지만 처음에는 메시지의 내용, 즉 Body 를 어떻게 구성할지 막막하였다. 검색을 통해 헤더는 작성하였지만 메시지 내용이 문제였다. 다행히 개발자 콘솔에서 html 요청에 대한 response 의 내용을 볼 수 있었다. 단순히 html 형식으로 body 를 구성해주면 되었다.



서버에서 HTTP request 를 html 형식에 맞게 바꿔 html 형식으로 body 를 구성한다. 이 body 를 제작한 request header 에 붙여 브라우저에 response 해준다.

제 2 절 코드 구현 과정

변수의 선언위치나 약간의 함수 위치를 제외하면 1 장의 서버코드와 거의 완벽히 유사하다. 다른 점이라면 read()와 write() 사이와 함수 1 개를 더 만든 것, 메시지의 메모리 크기가 다른 점이다. 나머지 구현 과정은 1 장에 설명되었다.

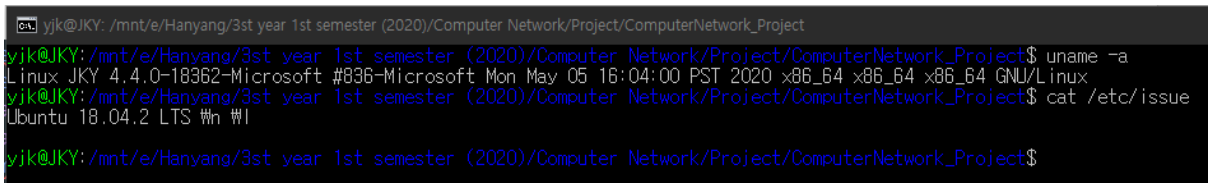
request message 와 response message 의 크기를 충분히 크게 하기 위해 MSG_SIZE 를 늘렸다.

```
#define MSG_SIZE 1000000
```

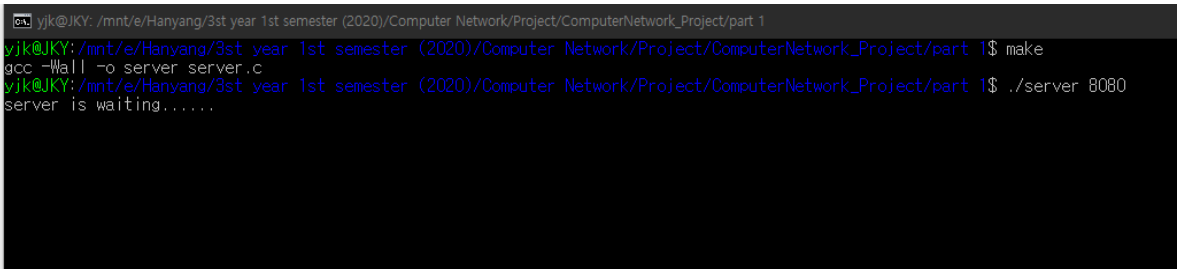
response_msg 를 만들기 위해 request_print 함수를 만들었다. 줄 바꿈 구현을 위해 줄바꿈 문자를 찾아
 줄바꿈 요소로 바꾸어 주었다. 그리고 content-length 는 <h></h>(길이 7) + \n 을
로 바꾼 request_msg (길이 strlen(req)) 로 구하였다.

```
void request_print(char *response_msg, char *request_msg);
void request_print(char *response_msg, char *request_msg){
    char *p = strtok(request_msg, "\n");
    char *req = malloc(sizeof(char)*MSG_SIZE);
    memset(req,0,MSG_SIZE);
    while( p!= NULL){
        strcat(req,p);
        strcat(req,"<br>"); // 줄 바꿈 구현
        p = strtok(NULL, "\n");
    }
    int content_length = 7 + strlen(req);
    sprintf(response_msg, "HTTP/1.1 200 OK\r\nContent-Type: text/html\r\nContent-
length: %d\r\n\r\n<h>%s</h>",content_length,req);
    free(req);
}
```

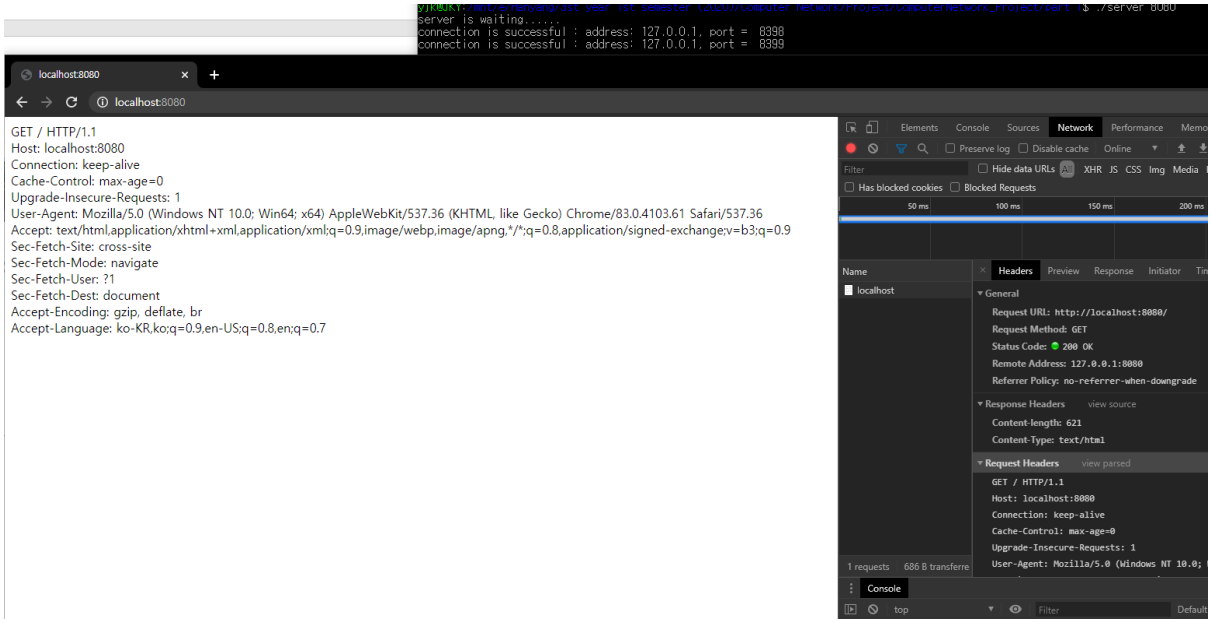
제 3 절 동작 예시



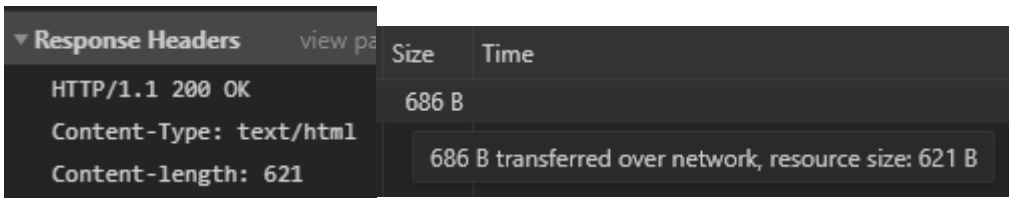
실행환경은 위와 같다.



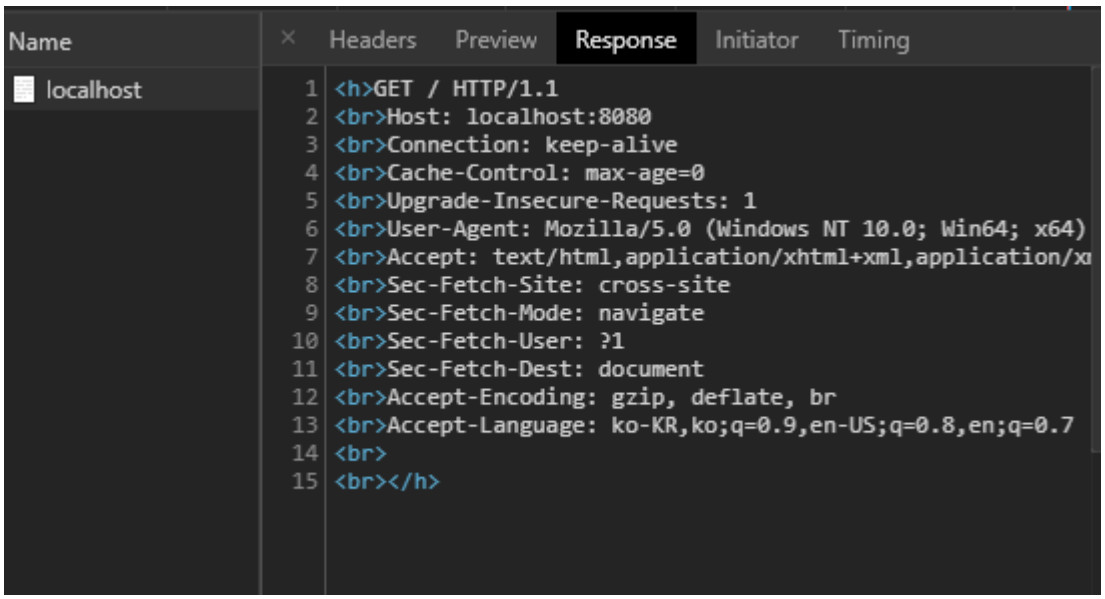
make 로 컴파일을 하고 포트번호 8080 으로 서버를 실행시킨다.



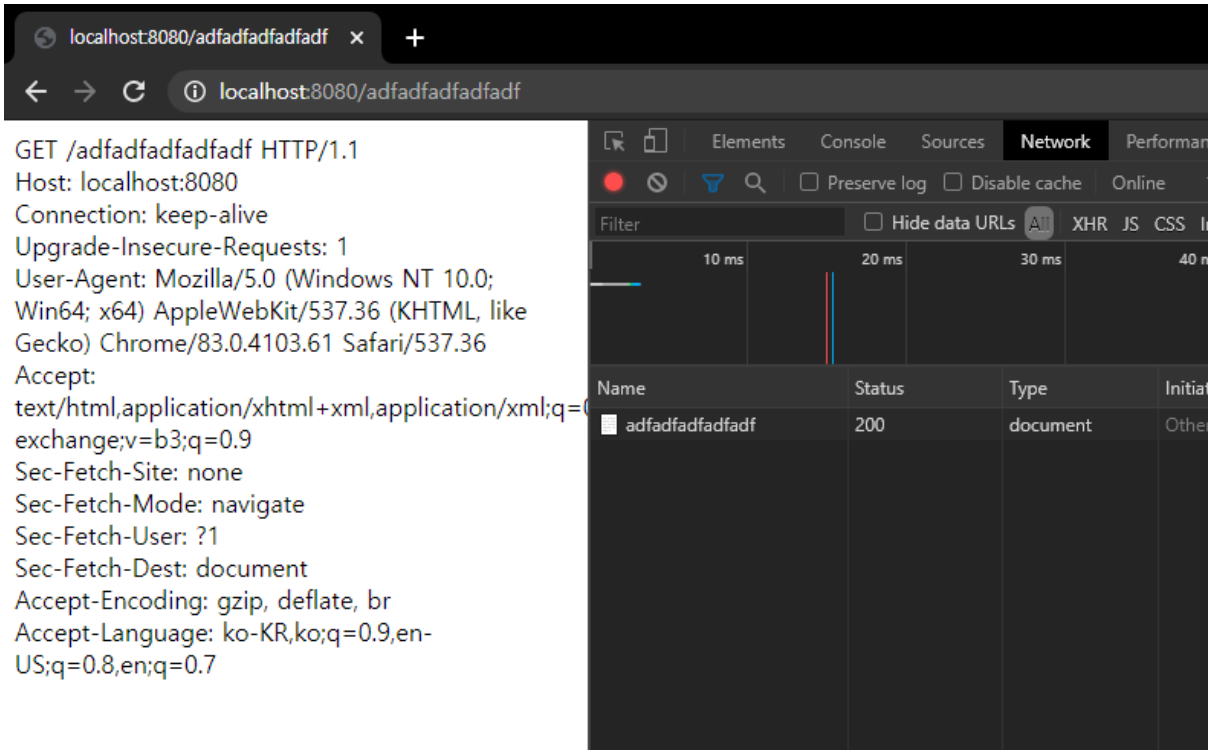
그 후 localhost:8080 을 통해 서버에 접속할 수 있다. 개발자 콘솔로 response 가 제대로 전달된 것을 확인할 수 있다.



response 메시지의 크기 686B 와 resource size, 즉 body size 는 621B 를 확인할 수 있다.



그리고 클라이언트에게 보낸 response 도 잘 도착한걸 볼 수 있다.



그리고 URI 에 상관없이 무조건 request message 만 출력한다.

```
yjk@JKY:/mnt/e/Hanyang/3st year 1st semester (2020)/Computer Network/Project/ComputerNetwork_Project/part 1$ ./server 8080
server is waiting.....
connection is successful : address: 127.0.0.1, port =  13308
connection is successful : address: 127.0.0.1, port =  13309
connection is successful : address: 127.0.0.1, port =  13312
connection is successful : address: 127.0.0.1, port =  13314
connection is successful : address: 127.0.0.1, port =  13316
connection is successful : address: 127.0.0.1, port =  13318
connection is successful : address: 127.0.0.1, port =  13320
connection is successful : address: 127.0.0.1, port =  13321
connection is successful : address: 127.0.0.1, port =  13326
connection is successful : address: 127.0.0.1, port =  13327
connection is successful : address: 127.0.0.1, port =  13330
connection is successful : address: 127.0.0.1, port =  13333
connection is successful : address: 127.0.0.1, port =  13335
connection is successful : address: 127.0.0.1, port =  13337
connection is successful : address: 127.0.0.1, port =  13339
connection is successful : address: 127.0.0.1, port =  13341
connection is successful : address: 127.0.0.1, port =  13343
connection is successful : address: 127.0.0.1, port =  13346
connection is successful : address: 127.0.0.1, port =  13348
connection is successful : address: 127.0.0.1, port =  13350
connection is successful : address: 127.0.0.1, port =  13352
connection is successful : address: 127.0.0.1, port =  13355
connection is successful : address: 127.0.0.1, port =  13357
connection is successful : address: 127.0.0.1, port =  13359
connection is successful : address: 127.0.0.1, port =  13361
connection is successful : address: 127.0.0.1, port =  13362
```

서버의 진행상황

제 3 장

2 장에서 클라이언트가 요청하면 HTML 형식으로 성공적으로 HTTP response 를 보내주었다. 이 다음에는 어떤 파일을 요청하면 요청한 파일에 해당되는 형식에 맞게 response 해 주어야 한다. 이 프로젝트에서는 5 개의 파일 확장자 (html, GIF, JPEG, MP3, PDF)만 인식하도록 했다. 해당 파일 확장자가 아닌 경우에는 HTTP 404 로 response 해주었다. 2 장과는 다르게 URI 에 의해 서버코드에 오류가 날 수 있기 때문에 예외처리를 해주었다.

제 1 절 서버 디자인

어떤 URI 인지 확인하기 위해 request message 를 strtok 을 이용해 분석하였다. 그래서 클라이언트가 요청한 파일이름을 구할 수 있었다. 그 다음에는 content-type 을 구하기 위해 파일이름을 strchr 와 strcmp 를 이용해 분석했다. 파일 확장자를 구할 수 있어 확장자에 알맞은 content-type 을 구할 수 있었다. 파일이름을 통해 파일 요청여부를 알 수 있어 파일 요청이 없다면 part1 의 요구사항처럼 request message 를 화면에 출력했다. 2 장의 request_print 함수를 거의 똑같이 사용했다.

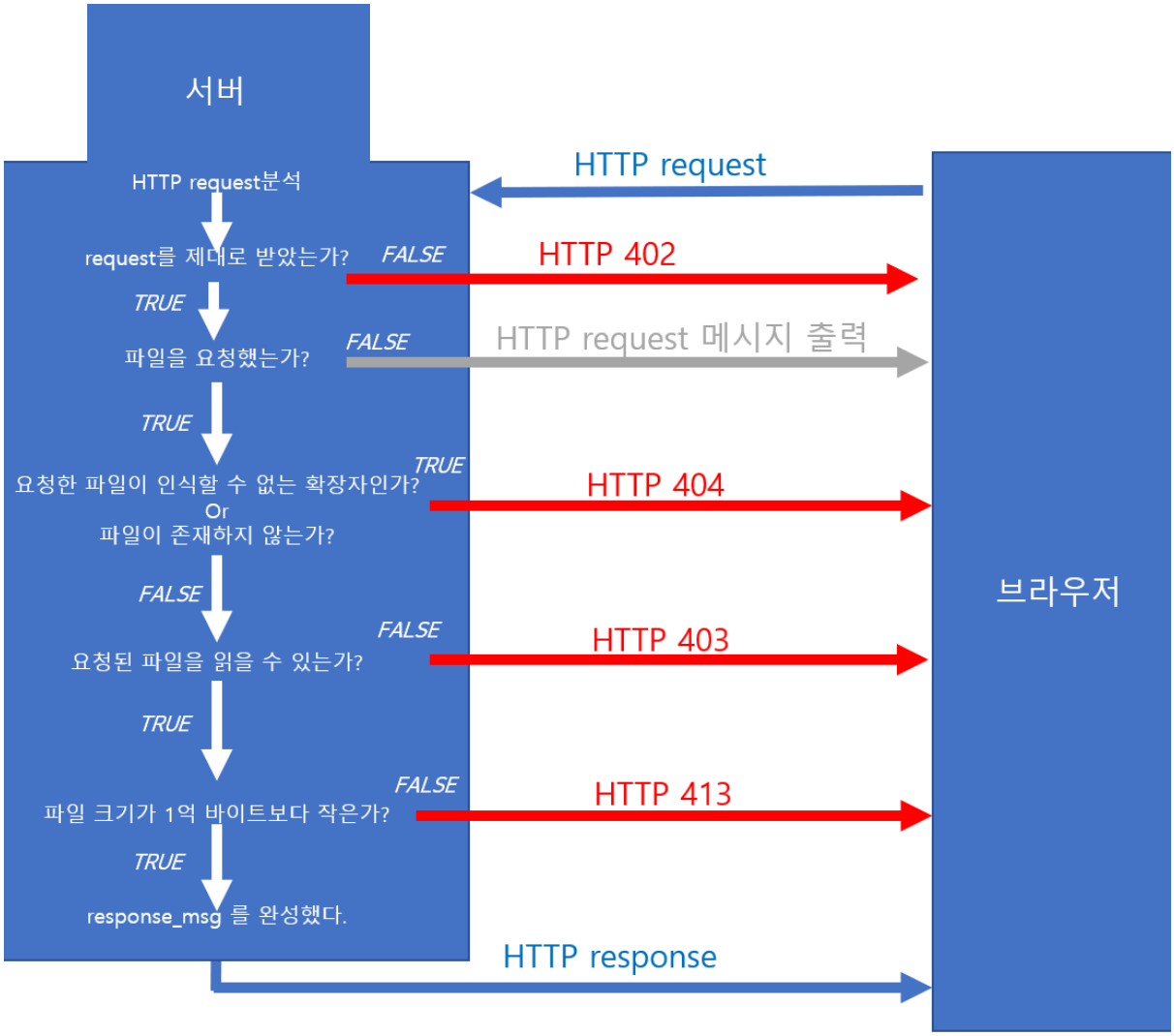
만약 파일 요청이 있다면 content-type 이 제대로 되었는지 해당 파일이 서버 컴퓨터에 존재하는지 확인한다. 만약 제대로 안되었거나 파일이 없다면 HTTP 404 로 클라이언트에 응답해준다. content-type 을 인식할 수 있고, 해당 파일이 존재한다면 각 content-type 에 맞게 response_msg 를 만들어 준다.

HTML 파일을 보내기 위해서 해당 파일을 열어서 그 내용을 버퍼에 넣었다. 그리고 그 버퍼를 헤더와 합쳐 클라이언트에 보내주었다. 눈으로 읽을 수 있는 텍스트이기 때문에 문제없이 클라이언트에 잘 보내졌다.

다음으로 사진 파일을 보내기 위해서 파일을 열었다. 직접 확인이 가능했던 HTML 파일과 다르게 제대로 파일을 읽는지 확인이 불가능 하였기 때문에 많은 시간이 걸렸다. 세그멘테이션 오류부터 시작해 다양한 오류를 겪었다. 특히 response body 를 볼 수 없는 점이 오류 해결에 어려움을 주었다. 제대로 파일을 읽는지 확인하기 위해 헥스 에디터를 이용해 버퍼에 바이너리 데이터가 제대로 들어가는지 확인했다. 가장 오래 힘들었던 부분은 바이너리 데이터도 잘 들어갔는데 content-length-mismatch 에러가 일어나는 부분이었다. 버퍼의 데이터를 하나하나 검사하고, 개행문자, NULL 문자, 공백이 버퍼에 들어갔는지 확인해보았다. 단순한 문제였지만 어떤 부분에서 길이 오류가 났는지 알 수가 없는 것이 힘들었다. 해결법은 개발자 콘솔을 이용했다. 콘솔의 네트워크 항목에서 request 에 대한 response 가 나와있었다. 그 중 SIZE 항에 마우스를 갖다 대면 "xB transferred over network, resource size : yB" 라고 나와있었다. 서버에서 파일 크기를 하나하나 다 출력하고 있었기 때문에 x 가 헤더와 메시지크기, y 가 메시지 크기를 의미하는 것을 알게 되었다. 문제점은 strlen 함수에 있었다. strlen 은 널 문자를 기준으로 길이를 측정하는데 바이너리 데이터는 중간에 널 문자가 들어갈 수가 있다. 그래서 파일을 열었을 때 파일 크기를 저장해서 헤더 크기와 더해줌으로서 response_msg 크기를 구할 수 있었다.

오디오 파일은 세그멘테이션 오류를 많이 겪었다. 그래서 gdb 를 이용해서 어느 부분에서 일어나는지 확인해서 메모리 할당을 충분히 해주었다. PDF 파일도 위와 같은 과정으로 제작했다.

각 파일에 대한 함수를 제작하고 살펴보니 각 함수가 거의 동일하였다. 그래서 함수 인수로 content-type 을 받아 하나의 함수로 통일시켰다. 그리고 코드를 읽기 편하게 기능적인 부분을 함수로 따로 작성했다.



서버 도식화 그림

제 2 절 코드 구현 과정

```
#define MSG_SIZE 100000000
void request_handle(char *response_msg, char *request_msg, int *response_size);
// 리퀘스트 처리 함수
void request_print(char *response_msg, char *request_msg, int *response_size);
// response message 를 만드는 함수 (프로젝트 Part1)
void request_file_handler(char *response_msg, char *request_msg, char *filename, int *respo
nse_size, char *content_type);
// Part2 요구사항 = 파일 인식
void response_error(char *response_msg, int *response_size, int error_code);
// HTTP 클라이언트 에러 처리
char *get_content_type(char *filename, char *file_extension);
// 파일 확장자 및 content-type
char *http_status_code(int code); //  HTTP 상태
void error_print(char *error_msg); // 에러문 출력 및 서버 종료
```

오디오 파일이나 PDF 파일은 텍스트 파일에 비해 크기 때문에 MSG_SIZE 를 충분히 크게 해주었다. 총 7 개의 함수를 제작했다. 각 함수에 대한 설명은 아래서 계속된다.

main 함수에서 소켓을 이용해 클라이언트와 연결하는 과정은 1 장과 동일하다. 단, response_msg 의 길이를 저장할 변수를 선언했다. 사용한 이유는 바이너리파일은 strlen() 을 이용해 길이를 구할 수 없기 때문이다.

그리고 네트워크문제인지, 브라우저 문제인지 가끔씩 request 메시지가 길이가 0 인 상태로 서버에 올 때가 있다. 그래서 무한 반복문에 조건문을 하나 만들어 아래처럼 작성했다.

```
while(1){
        memset(response_msg, 0 ,MSG_SIZE); // 클라이언트로 보낼 메시지 초기화
        memset(request_msg, 0 ,MSG_SIZE);   // 클라이언트에서 받은 메시지 초기화
        if( (socket_fd = accept( server_socket_fd, (struct sockaddr *)&client_addr, &req_cl
ient)) < 0){
            error_print("Fail to accpet");
        }
        printf("connection is successful : address: %s, port = % d\n",
            inet_ntoa(client_addr.sin_addr), ntohs(client_addr.sin_port));

        if( read(socket_fd, request_msg, MSG_SIZE) < 0 ){
            error_print("Fail to read");
        }
        if(strlen(request_msg) == 0) { response_error(response_msg, &response_size, 402); }

        else{ request_handle(response_msg, request_msg, &response_size);}
        if ( write(socket_fd, response_msg, response_size)  < 0 )
            error_print("Fail to writing to socket.");
        }
        close(socket_fd);
    }
```

클라이언트에게 request message 를 정상적으로 받으면 request_handle 함수를 실행한다. 인자로는 두개의 메시지의 주소와 response_msg 의 크기를 저장하는 변수의 주소로 넘겨주었다.

```
void request_handle(char *response_msg, char *request_msg, int *response_size){
    char *request_msg_copy = malloc( sizeof(char)*MSG_SIZE); //strtok 때문에 복사본을 만든다.
    strcpy(request_msg_copy, request_msg); // 문자열 복사
    char *filename; // 파일 이름. 파일 이름 앞에 '/' 문자가 붙는다.
```

```
    char content_type[50]; // content-type
    char *file_extesion; // content-type 을 구하기 위한 파일 확장자
    file_extesion = malloc(sizeof(char)*300);
    filename = malloc(sizeof(char)*600);
    strtok(request_msg_copy, " "); // GET method 를 반환한다.
    strcpy(filename, strtok(NULL," ")); // 파일 이름을 저장한다.
    free(request_msg_copy); // 메모리 해제

    if (!strcmp(filename, "/") ){ // 파일 입력이 없는 경우에는
        request_print(response_msg, request_msg, response_size); // request message 를 출력
해준다. (프로젝트 Part1)
    }
    else{ // 입력받은 파일이 있을 경우
        strcpy(content_type, get_content_type(filename,file_extesion)); // 파일의 확장자 명
과 content-type 을 함수를 이용해 가져온다.
        if( !strcmp(content_type, "nomake") || access(filename+1,F_OK) < 0 ){ // 만약 인식하
지 못하는 확장자명이거나 인식하지만 해당 파일이 존재하지 않는 경우에는
            response_error(response_msg, response_size, 404); // HTTP 404 로 응답한다.
        }
        else{ // 아무런 이상이 없는 경우에는
            request_file_handler(response_msg, request_msg, filename+1, response_size, cont
ent_type); // 함수를 이용해 response_msg 를 만든다. // 파일이름앞의 '/'을 없애기 위해 +1 해주었
다.
        }
    }
    free(filename); // 메모리 해제
    free(file_extesion); // 메모리 해제
}
```

request_handle 함수에서는 strtok 를 이용해 request_msg 를 분석해 파일 이름을 얻었다. 그리고 strcmp 를 이용해 파일요청이 들어왔는지 확인하고 get_content_type 을 이용해 파일의 확장자와 헤더가 들어갈 content-type 을 구했다.

```
char* get_content_type(char *filename, char *file_extension){ //파일 이름으로 확장자와 conten
t-type 을 얻는다.
    if (!strcmp(filename, "/") ){ // 요청 받은 파일이 없을 경우
        return "text/html"; // request 파일을 출력해주기 위해 html 형식을 반환한다.
    }

    char name[100]; // 파일 이름을 복사할 변수
    strcpy(name, filename); // 문자열 복사
    char *p = strchr(name,'.'); // 파일 이름에서 '.'을 찾는다.
    if ( p == NULL){ // '.' 이 없다.
        return "nomake"; // 파일 확장자가 없으면 인식할 수 없다.
    }
    if ( !strcmp(p, ".")){ // '.' 이후에 적혀있는 문자열이 없다
        return "nomake"; // 파일 확장자가 없으면 인식할 수 없다.
    }

    char file_type[20]; // 파일 확장자를 얻기 위한 문자열
     // 파일 이름 중간에 '.' 이 들어갈 수 있으므로 맨 마지막에 있는 '.'을 찾아준다.
    while( p != NULL){
        strcpy(file_type, p);
        p = strchr(p+1, '.');
    }
    strcpy(file_extension, file_type+1); // file_type 맨 앞에 '.'문자가 있으므로 제외하고 파일
 확장자만을 복사한다.
```

```
    //파일 확장자에 알맞는 content-type 을 반환한다.
    if(strcmp(file_type+1, "html") == 0){
        return "text/html";
    }
    if(strcmp(file_type+1, "mp3") == 0){
        return "audio/mpeg3";
    }
    if(!strcmp(file_type+1, "gif")){
        return "image/gif";
    }
    if(!strcmp(file_type+1, "jpg") || !strcmp(file_type+1, "jpeg")){
        return "image/jpeg";
    }
    if(!strcmp(file_type+1, "png")){
        return "image/png";
    }
    if(!strcmp(file_type+1, "pdf")){
        return "application/pdf";
    }
    return "nomake"; //지원하지 않는 확장자
}
```

위 함수는 file_extension 에 파일 확장자를 저장하고 확장자에 해당하는 content-type 을 반환한다. 이 때, 예외 처리를 위해 strcmp 를 이용해 모든 예외처리를 해주었다. 인식할 수 없는 경우에는 content-type 대신 "nomake"라는 문자열을 반환한다.

request_handle 에서 content-type 을 "nomake"로 얻거나 요청된 파일이 서버에 없다면, request_error 함수를 이용해 HTTP 404 로 클라이언트에 응답한다. 모든 조건에 해당되지 않으면 request_file_handler 함수를 이용해 response_msg 를 만든다.

```
void request_file_handler(char *response_msg, char *request_msg, char *filename, int *respo
nse_size, char *content_type){ // 파일이 요청되었다면 그에 해당되는 response 를 해주는 함수.
    FILE *openfile; // 서버에서 여는 요청받은 파일
    if( (openfile = fopen(filename, "rb")) == NULL ){ // 파일이 열리지 않았다면
        response_error(response_msg, response_size, 403); // HTTP 403 으로 응답한다.
        return ;
    }
    //파일 크기를 구하기 위한 과정
    fseek(openfile, 0, SEEK_END); // 파일 포인터를 맨 뒤로 옮긴다.
    int file_size = ftell(openfile); //현재의 위치 값
    fseek(openfile, 0, SEEK_SET); // 파일을 읽기위해 파일 포인터를 맨 앞으로 옮긴다.


    if(file_size > MSG_SIZE){ //만약 파일 크기가 전송 크기보다 더 크다면
        response_error(response_msg, response_size, 413); // HTTP 413 으로 응답한다. (이 코드
가 맞는지는 확실하지 않다.)
        fclose(openfile);
        return ;
    }

    int leng= sprintf(response_msg, "HTTP/1.1 200 OK\r\ncontent-Type: %s\r\ncontent-
length: %d\r\n\r\n", content_type, file_size); // HTTP 헤더를 만든다.
    *response_size = file_size + leng; //HTTP 헤더길이 + content-
length = response message 길이
    unsigned char *send_buffer; // 파일을 읽을 버퍼
    send_buffer = malloc(MSG_SIZE); // 버퍼에 메모리 할당
    fread(send_buffer, file_size, 1, openfile); // openfile 을 읽어 send_buffer 에 저장한다.
```

```
    memcpy(response_msg+leng,  send_buffer, file_size); // HTTP 헤더에 읽은 파일 내용을 붙인다.
    free(send_buffer); // 메모리 해제
    fclose(openfile); // 파일 닫기
}
```

이 함수에서 filename 을 이용해 파일을 연다. 만약 열리지 않다면 HTTP403 으로 클라이언트에 응답한다. 파일이 열리면 fseek 과 ftell 을 이용해 파일의 크기를 구한다. 파일의 크기를 구한 후 파일 크기가 매우 크다면 HTTP413 으로 클라이언트에 응답한다. 파일이 브라우저에 전달될 수 있을 정도의 크기면 파일 크기과 content-type 을 이용해 HTTP 헤더를 만든다. 그리고 fread 를 이용해 파일크기만큼 파일을 읽는다. memcpy 를 이용해 헤더와 파일을 읽은 버퍼를 붙인다.

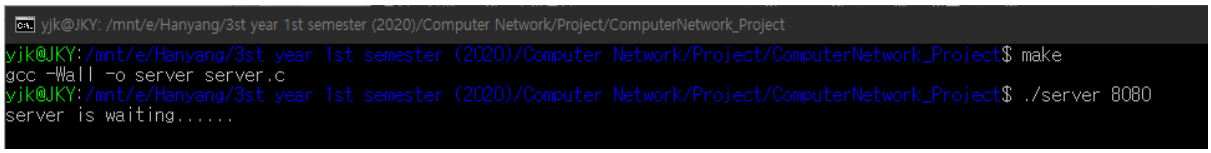
파일이 성공적으로 읽혔기 때문에 response_msg 는 정상적으로 완성되었고 write 를 통해 클라이언트에 전달되어 브라우저에 결과가 나타난다.

오류가 났을 때는 HTTP 에러 코드를 클라이언트에 보내기 위해 두개의 함수를 만들었다.

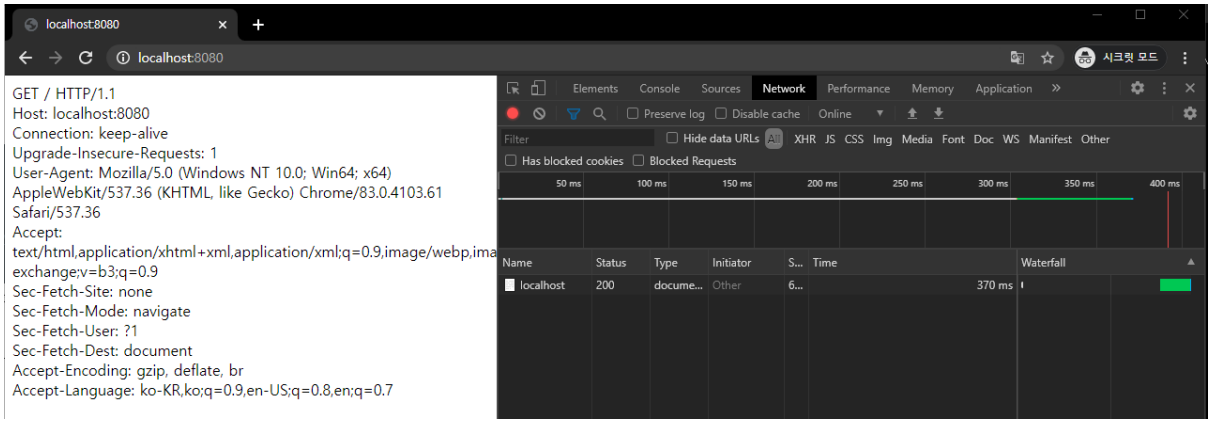
```
void response_error(char *response_msg, int *response_size, int error_code){ //클라이언트에서 잘못 요청이 들어왔을 시 어떤 에러인지 보여주는 함수.
    char *status_str = http_status_code(error_code); // 에러 코드를 이용해 어떤 에러인지 문자열로 갖고온다.
    int content_length = 18+strlen(status_str); // content-length field 를 위해 값을 구하였다.

    // response_msg 를 만들고 response 메시지의 길이를 구했다. 어떤 에러인지 보여주기 위해 html 로 response 했다.
    *response_size = sprintf(response_msg, "HTTP/1.1 %d %s\r\nContent-Type: text/html\r\nContent-length: 27\r\n\r\n<h1>HTTP %d %s</h1>",error_code,status_str,error_code,status_str);
}
char *http_status_code(int code){ // HTTP 에러 코드에 따른 에러문구를 얻기위한 함수
    switch(code){
        case 400:
            return "Bad Request";
            break;
        case 404:
            return "Not Found";
            break;
        case 403:
            return "Forbidden";
            break;
        case 413:
            return "Payload Too Large";
            break;
        default:
            return "I don't know"; // 일부만 구현했다.
    }
}
```

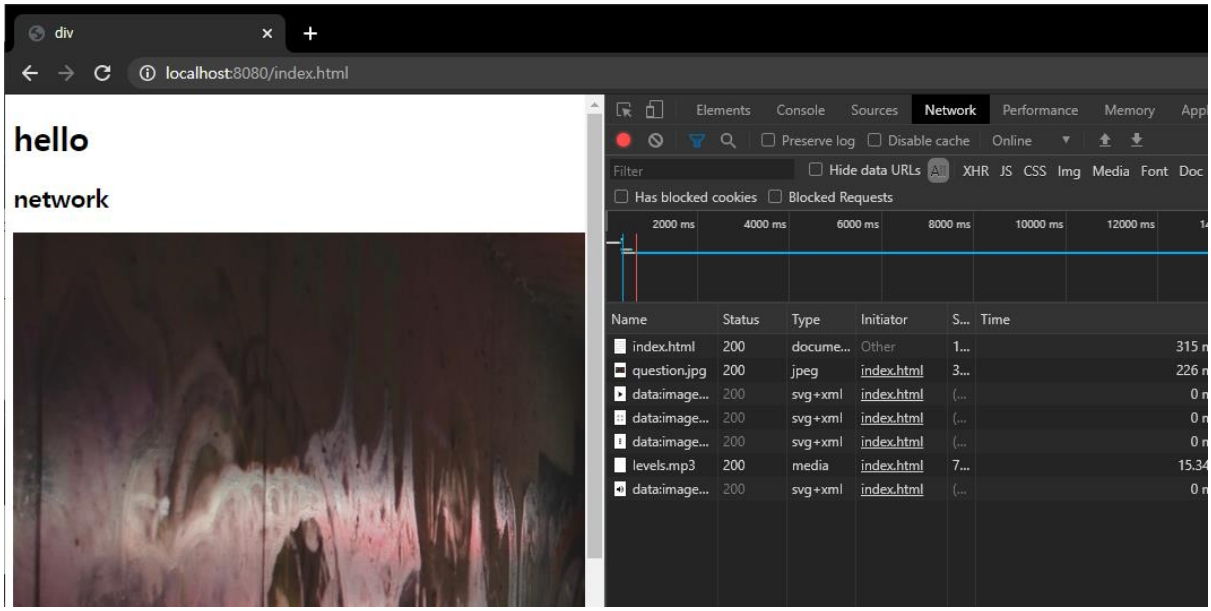
단순하게 에러가 발생하면 에러 코드와 그 내용을 <h1>태그로 감싸 html 형식으로 response_msg 를 만든다.



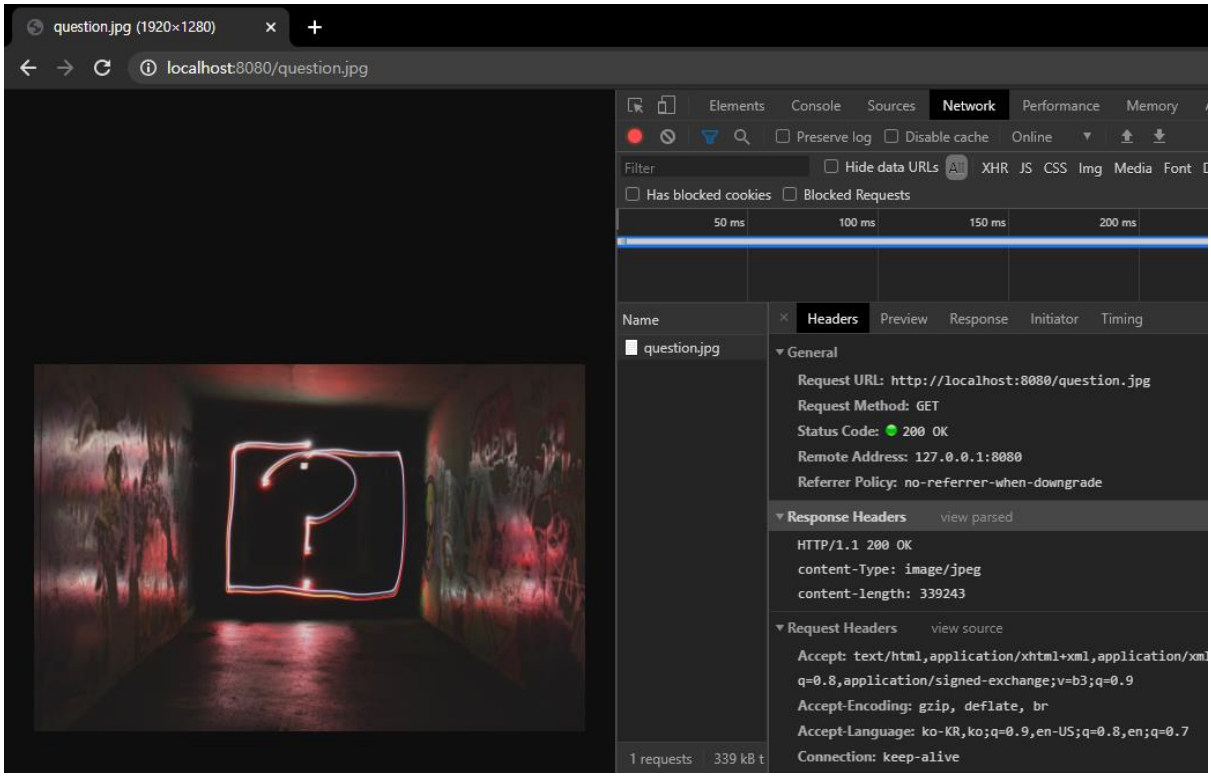
컴파일 및 실행



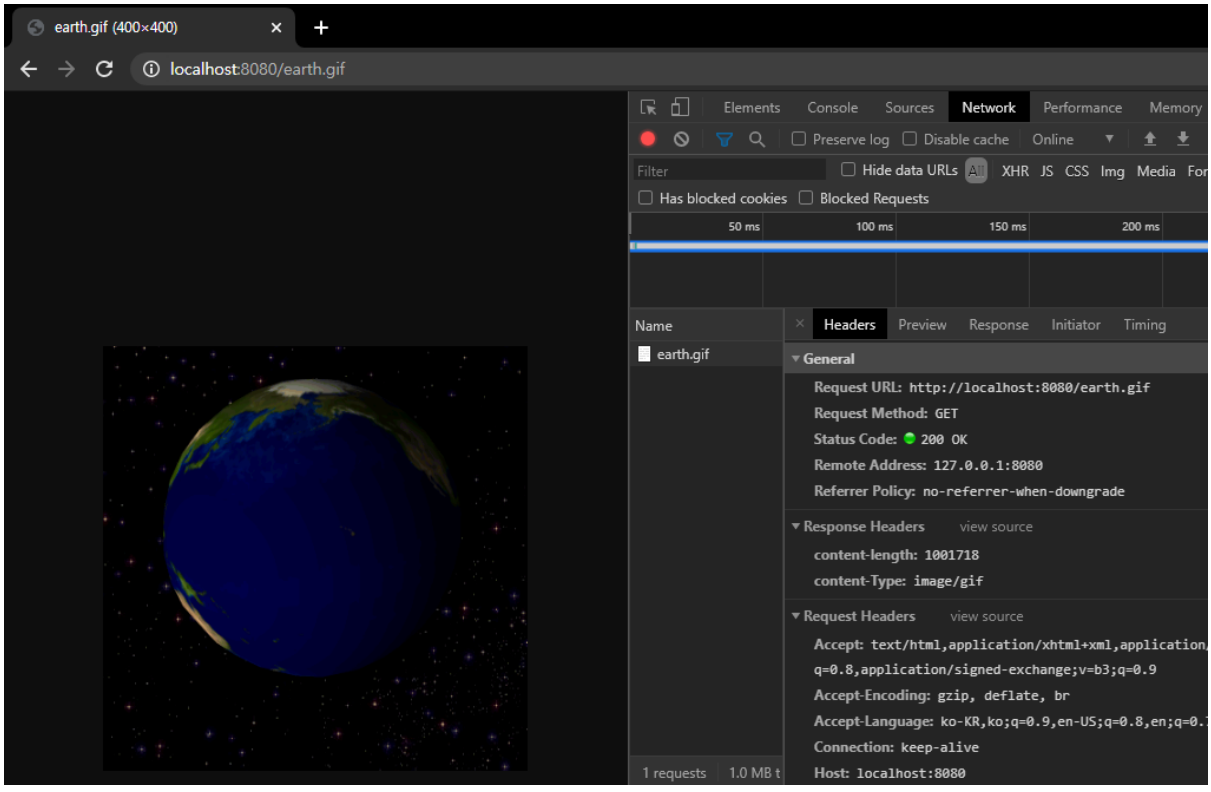
localhost:8080 에 대한 결과



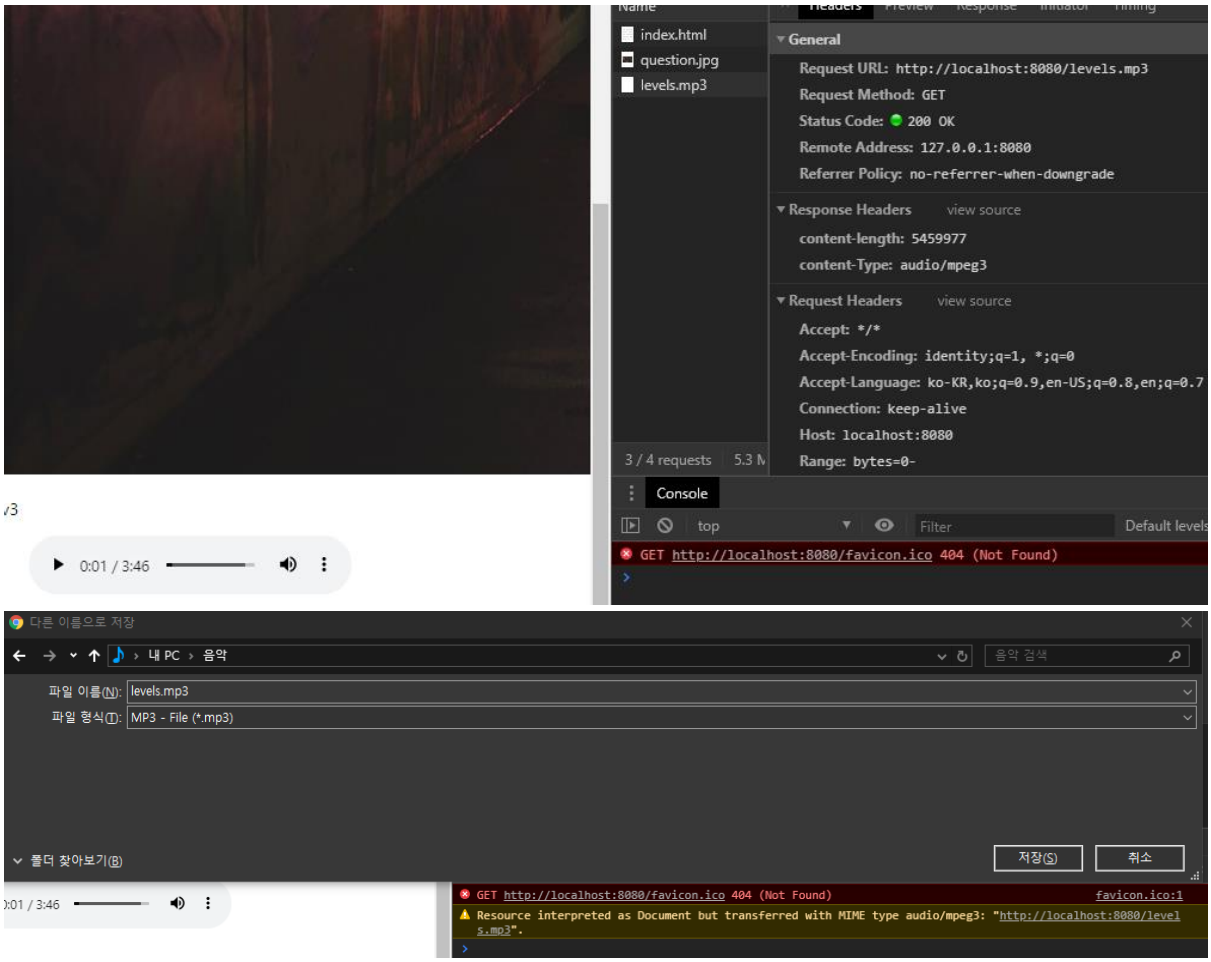
HTML 파일인식



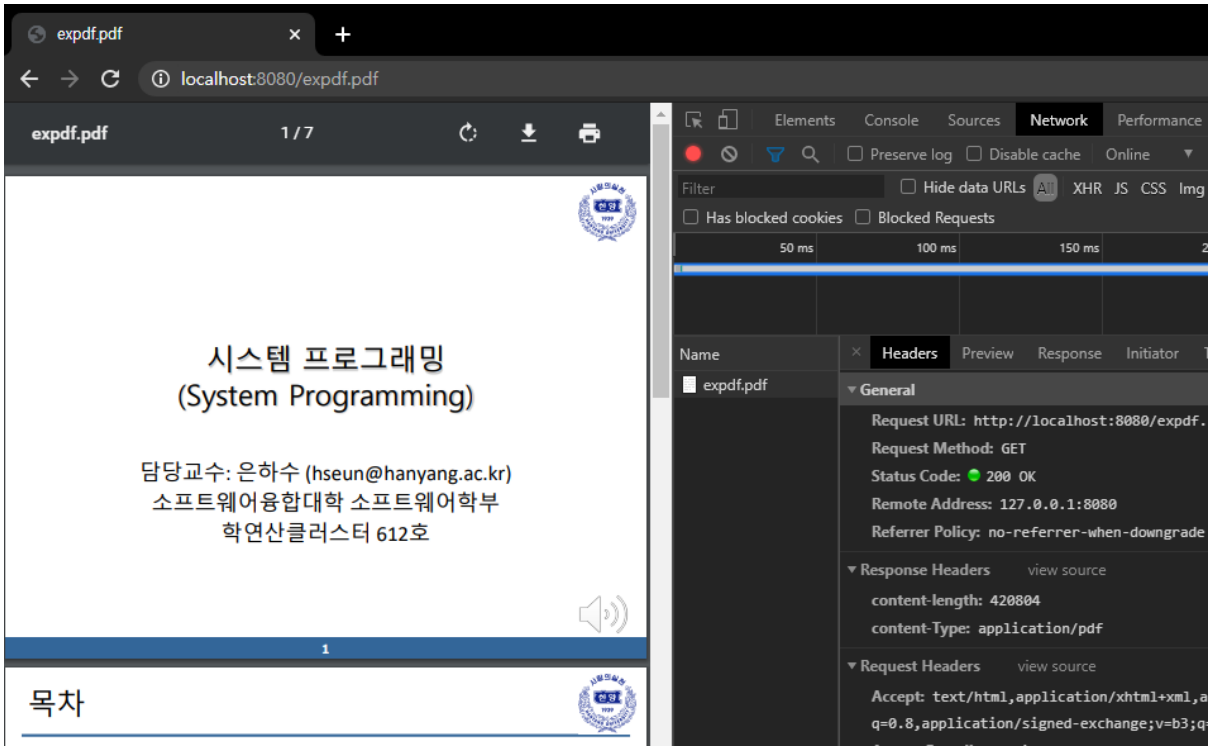
JPG 파일 인식



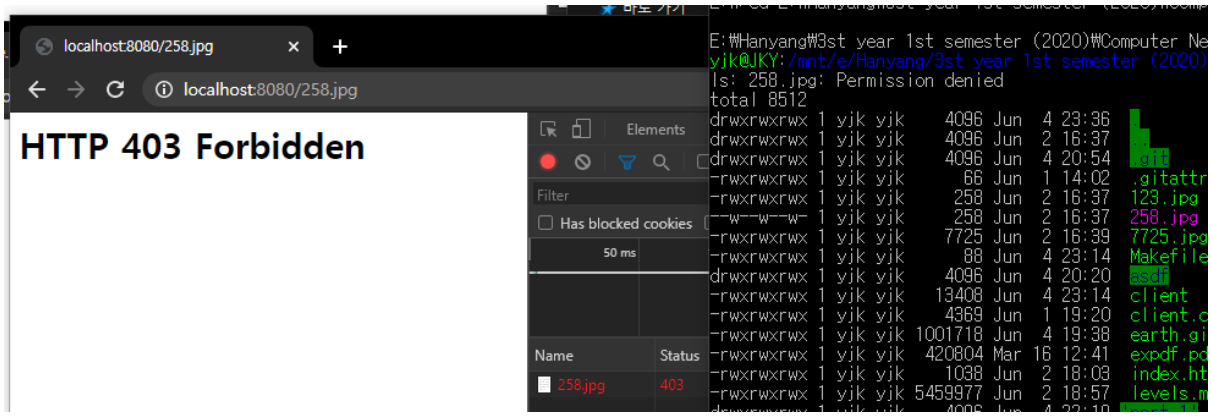
GIF 파일 인식



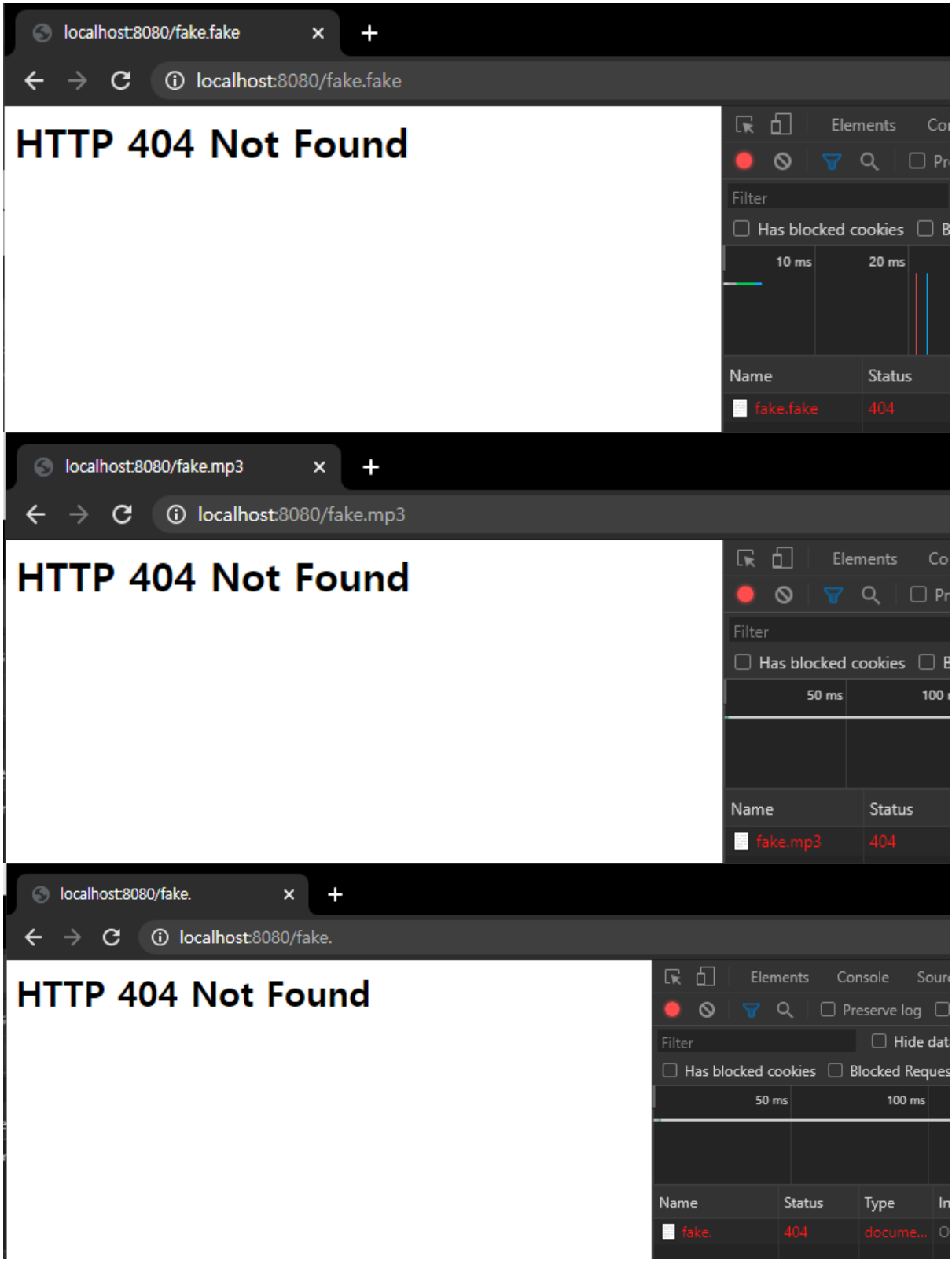
MP3 파일 인식과 favicon.ico 이 없으므로 404 에러가 난다.
audio/mpeg3 에 대해서는 화면에 나오지 않아 localhost:8080/levels.mp3 에 대해 다운로드가 된다.



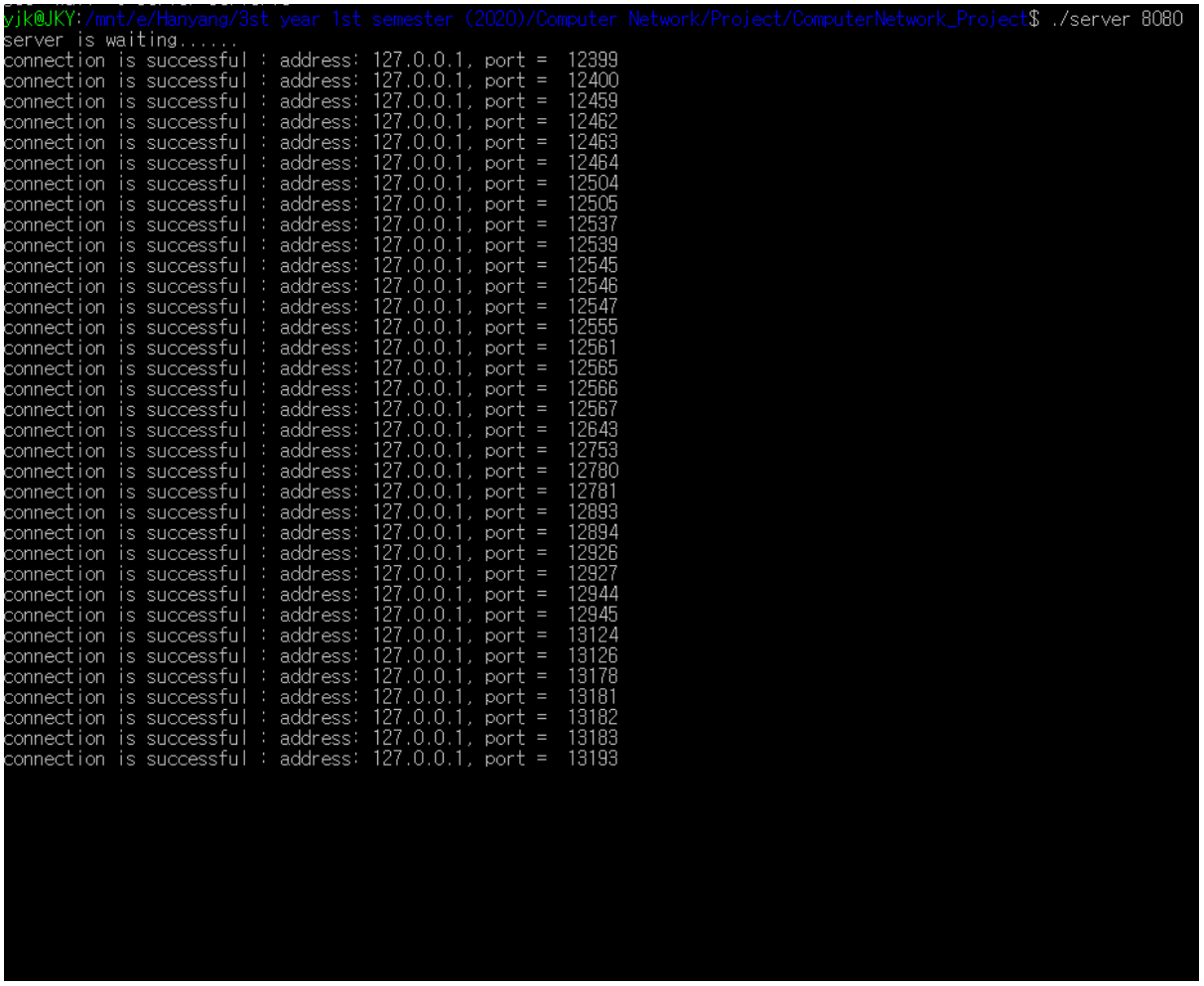
PDF 파일 인식



읽기가 금지된 파일



인식하지 않는 확장자/ 존재하지 않는 파일/ 입력이 없는 확장자


vjk@JKY:/mnt/e/Hanyang/3st year 1st semester (2020)/Computer Network/Project/ComputerNetwork_Project$ ./server 8080
server is waiting.....
connection is successful : address: 127.0.0.1, port = 12399
connection is successful : address: 127.0.0.1, port = 12400
connection is successful : address: 127.0.0.1, port = 12459
connection is successful : address: 127.0.0.1, port = 12462
connection is successful : address: 127.0.0.1, port = 12463
connection is successful : address: 127.0.0.1, port = 12464
connection is successful : address: 127.0.0.1, port = 12504
connection is successful : address: 127.0.0.1, port = 12505
connection is successful : address: 127.0.0.1, port = 12537
connection is successful : address: 127.0.0.1, port = 12539
connection is successful : address: 127.0.0.1, port = 12545
connection is successful : address: 127.0.0.1, port = 12546
connection is successful : address: 127.0.0.1, port = 12547
connection is successful : address: 127.0.0.1, port = 12555
connection is successful : address: 127.0.0.1, port = 12561
connection is successful : address: 127.0.0.1, port = 12565
connection is successful : address: 127.0.0.1, port = 12566
connection is successful : address: 127.0.0.1, port = 12567
connection is successful : address: 127.0.0.1, port = 12643
connection is successful : address: 127.0.0.1, port = 12753
connection is successful : address: 127.0.0.1, port = 12780
connection is successful : address: 127.0.0.1, port = 12781
connection is successful : address: 127.0.0.1, port = 12893
connection is successful : address: 127.0.0.1, port = 12894
connection is successful : address: 127.0.0.1, port = 12926
connection is successful : address: 127.0.0.1, port = 12927
connection is successful : address: 127.0.0.1, port = 12944
connection is successful : address: 127.0.0.1, port = 12945
connection is successful : address: 127.0.0.1, port = 13124
connection is successful : address: 127.0.0.1, port = 13126
connection is successful : address: 127.0.0.1, port = 13178
connection is successful : address: 127.0.0.1, port = 13181
connection is successful : address: 127.0.0.1, port = 13182
connection is successful : address: 127.0.0.1, port = 13183
connection is successful : address: 127.0.0.1, port = 13193

서버의 진행상황