**리눅스 한 학기 살기 보고서**

2022105737 장찬욱

## **1주차 활동**

### **라즈베리파이에 우분투 설치**

* **노트북과 데스크탑에 우분투 wsl 환경 설치**

## **2주차 활동**

* **라즈베리 파이의 우분투 환경에서 도커 스웜 생성**
* **데스크탑의 리눅스 VM과 노트북의 리눅스VM를 스웜에 조인**

**설치한 프로그램: Docker**

저번 학기 **마이크로서비스프로그래밍**을 수강하며 도커를 사용해보았습니다. **도커 스웜**을 실습할 때 하나의 컴퓨터 안에서 VM을 2개 만들고 스웜에 조인시키는 방식으로 실습했으나, 실제로는 **물리적으로 다른 컴퓨터 2대를 조인시켜보고 싶었습니다.**

이에 **라즈베리파이**에 우분투를 설치하고 도커를 설치하고 스웜을 생성했습니다. 데스크탑과 노트북을 워커 노드로 하여 조인해보려 했습니다. 두 컴퓨터에서 각각 리눅스 VM을 만들고 VM으로 라즈베리파이의 스웜에 조인했습니다.

조인 명령어를 사용하니 오류가 발생하였고 **방화벽 규칙** 문제였습니다. 스웜 조인에 필요한 2377 포트 등 여러 포트를 열지 않았었고 열리도록 몇 가지 **ufw** 명령어들을 사용하여 방화벽 규칙을 추가, 리로드, 이네이블하여 적용했습니다. 이후 두 컴퓨터 모두 조인에 성공했습니다.

# **진행과정**

## **1. 라즈베리파이에서 IP 확인**

ifconfig

### 위 명령어의 출력결과에서 IP를 찾았습니다.

### ping <라즈베리파이\_IP>

데스크탑과 노트북에서 위 명령어를 입력하여 같은 공유기에 연결된 노트북, 데스크탑, 라즈베리파이가 패킷을 주고받고 있는 것을 확인했습니다.

## **2. Curl 및 Docker 설치 (라즈베리파이 및 VM) 우분투 기준**

sudo apt update

sudo apt install -y curl

curl -fsSL https://get.docker.com | sudo sh

sudo systemctl enable docker

sudo systemctl start docker

docker --version

## **3. Docker Swarm 초기화 (라즈베리파이에서)**

## **라즈베리파이를 매니저로 설정**

sudo docker swarm init --advertise-addr <라즈베리파이\_IP>

### 위 명령 실행 후 출력되는 join 토큰 확인하고 기록 (워커 추가용)

### **4. Swarm에 워커 조인 (데스크탑/노트북 VM에서)**

sudo docker swarm join --token <토큰값> <라즈베리파이\_IP>:2377

## **5. 방화벽 설정 - Docker Swarm 관련 포트 열기 (라즈베리파이에서)**

sudo ufw allow 2377/tcp # Swarm 매니저 포트

sudo ufw allow 7946/tcp # 노드 간 통신

sudo ufw allow 7946/udp

sudo ufw allow 4789/udp # overlay 네트워크

sudo ufw allow ssh # SSH 유지

## **방화벽 적용**

sudo ufw reload

sudo ufw enable

sudo ufw status

**8. 상태 확인**

## **매니저(라즈베리파이)에서 노드 상태 확인**

sudo docker node ls

## **느낀 점:**

원래 도커를 윈도우에서 wsl2를 통해 사용해 왔는데 도커 명령어 자체는 앞에 sudo를 붙여야 오류없이 잘 작동한다는 점에서 차이가 느껴졌습니다. 이제 권한에 대해 명시적으로 관리해야 한다는 사실을 깨달았습니다. 윈도우에서는 wsl2를 설치하고 도커를 사용해야 했는데 OS를 리눅스로 설치하니 명령어 몇 개로 도커를 사용할 수 있어서 매우 편리했습니다. 이외에도 방화벽 설정, 네트워크 설정을 할 때 윈도우보다 짧고 간결한 명령어로 설정 가능한 것이 좋았습니다. 터미널을 통해 직접적으로 컴퓨터를 조작할 수 있다는 느낌이 들었고 코드 작성 및 수정도 nano를 사용하다보니 GUI를 사용할 일이 없었던지라 GUI가 꼭 필요한 것일까 의문이 들기도 했습니다.

**3주차 활동**

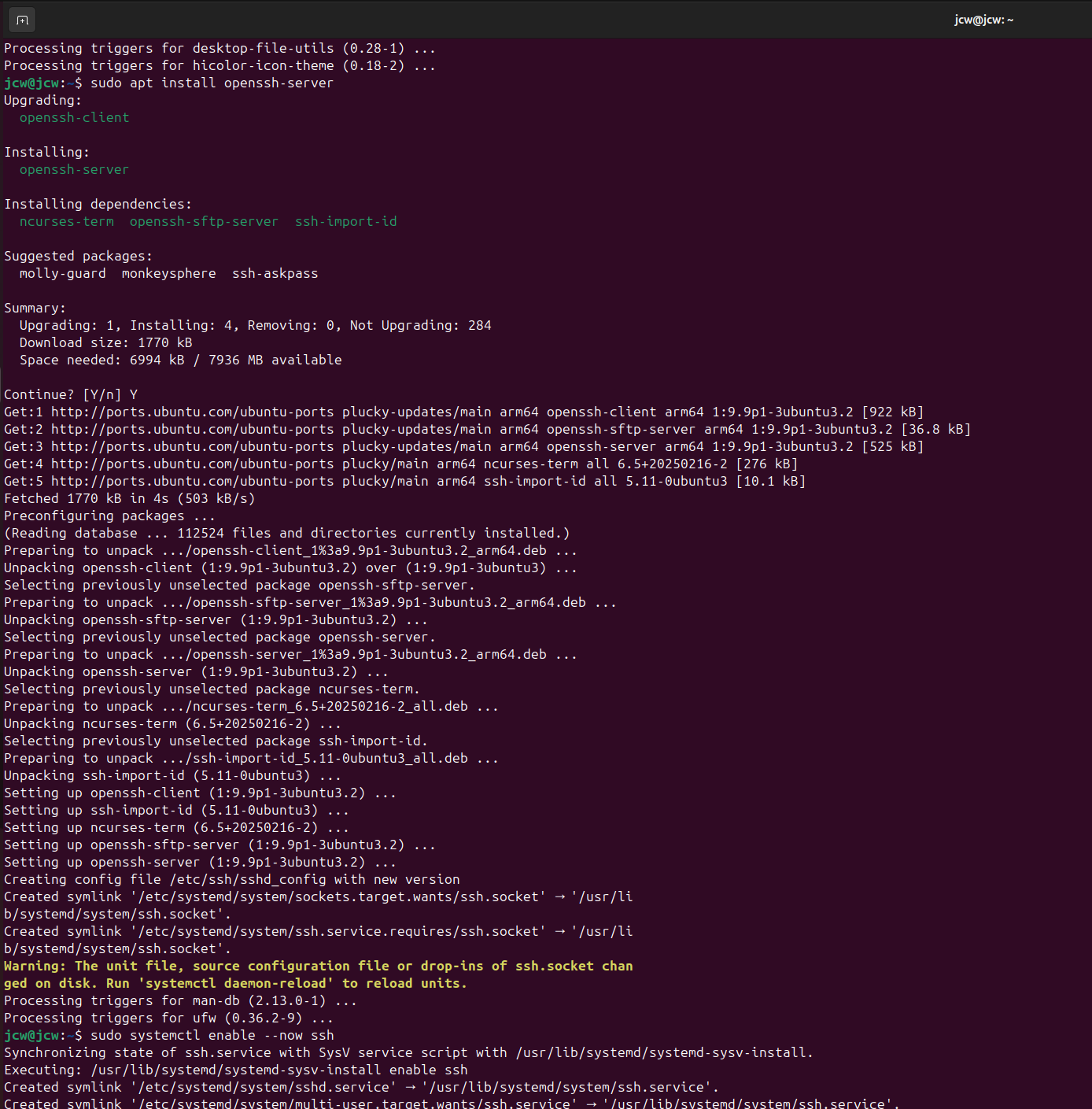
* **라즈베리 파이의 우분투에서 SSH 서버 설치**
* **노트북의 SSH 클라이언트로 라즈베리파이의 SSH 접속**

**사용한 프로그램: openssh-server**

### **라즈베리 파이에 SSH 설치 및 노트북에서 라즈베리파이 접속**

1. **SSH 서버 설치 및 실행**

sudo apt install apache2 –y



**SSH 서비스 상태 확인**

sudo systemctl status ssh

sudo systemctl enable --now ssh

sudo systemctl status ssh

1. **방화벽 규칙 설정**

sudo ufw allow ssh

sudo ufw enable

sudo ufw status

1. **공유기 페이지에서 22번 포트(SSH)와 80번 포트(HTTP) 포트포워딩 설정**
2. **노트북 SSH 클라이언트로 접속을 시도**

**포트 포워딩 전후로** 공유기의 WAN IP로 SSH **접속 시도할 때 결과가 달라지는 것**을 확인할 수 있었다. **포트 포워딩한 이후로는 SSH 접속이 성공**하는 것이 확인된다.

1. 공유기의 외부 IP를 통해 접속한 것이기 때문에 집 공유기가 아니라 핸드폰 핫스팟을 이용한 **외부 네트워크에서도 공유기를 통하여 라즈베리파이에 접속 가능했습니다.**

**느낀 점:** 학교나 집 앞 카페에서도 집에 있는 라즈베리파이에 접속할 수 있고 이런 저런 과제를 하는 데에 사용할 수 있게 되니 너무 간편하고 좋았습니다. 이러한 일들을 가능하게 만드는 네트워크 기술에 대해 흥미가 생겼습니다. 내 컴퓨터에서 다른 컴퓨터로 패킷을 전송하는 과정을 알아보다가 이를 설명하는 데에 OSI 7계층, NIC, 커널, 직렬화 등의 개념이 필요하다는 것을 알게 되었습니다. 운영체제, 컴퓨터네트워크, 컴퓨터구조 등의 전공필수 과목들이 왜 필수로 지정되어 있는지 이해가 되는 시간이었습니다.

## **4주차 활동**

### **라즈베리 파이에 SSH로 접속하여** **APACHE 웹 서버** **설치**

### **노트북에서 포스트맨 사용해 아파치 웹사이트 작동 확인**

**사용한 프로그램: apache2**

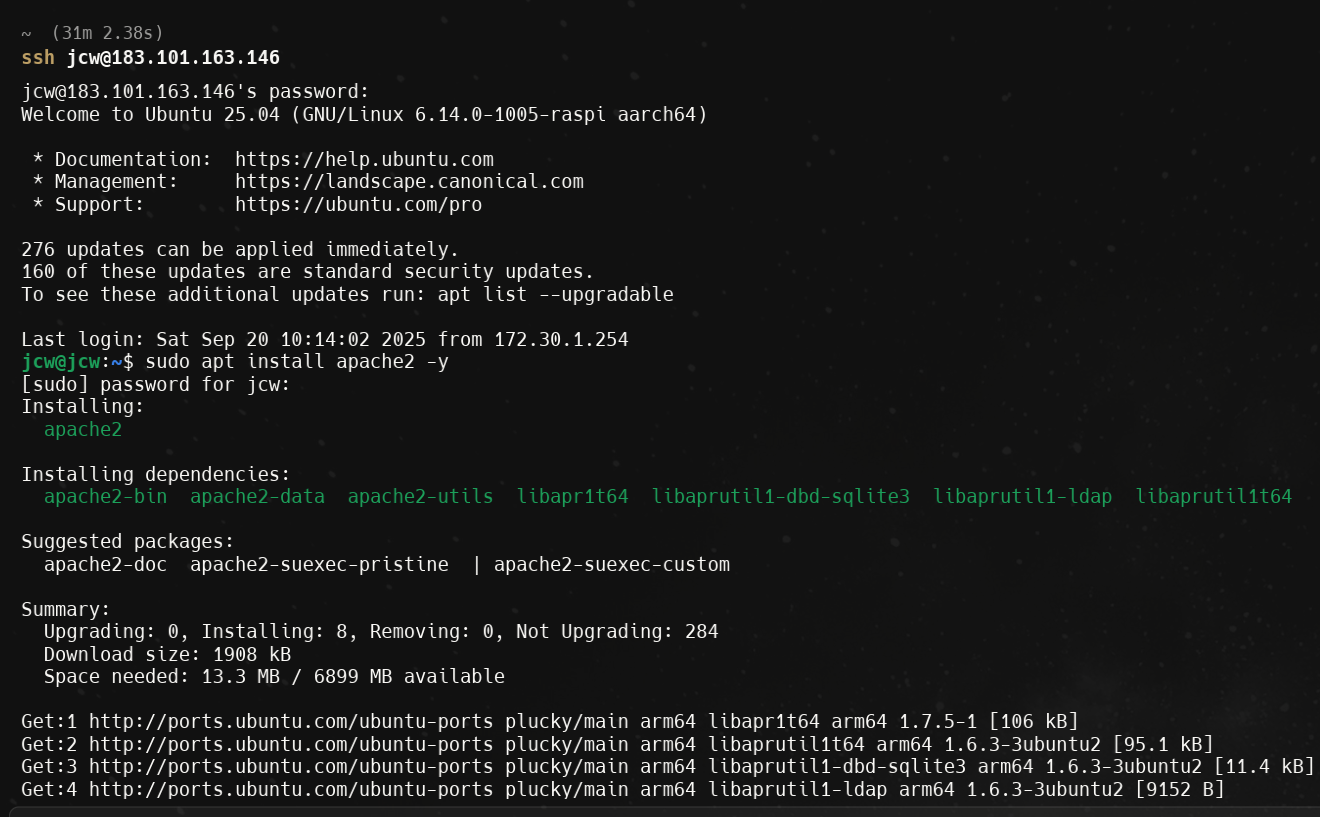
저번주 활동에서 노트북의 SSH 클라이언트로 라즈베리 파이에 접속한 가능한 상태까지구현했습니다. SSH 클라이언트로 라즈베리파이에 접속한 상태에서 웹서버를 설치해봤습니다.

1. **Apache2 설치**

sudo apt install apache2 –y

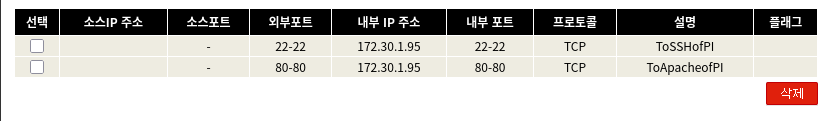
1. **아파치 서버 작동 상태 확인**

sudo systemctl status apache2



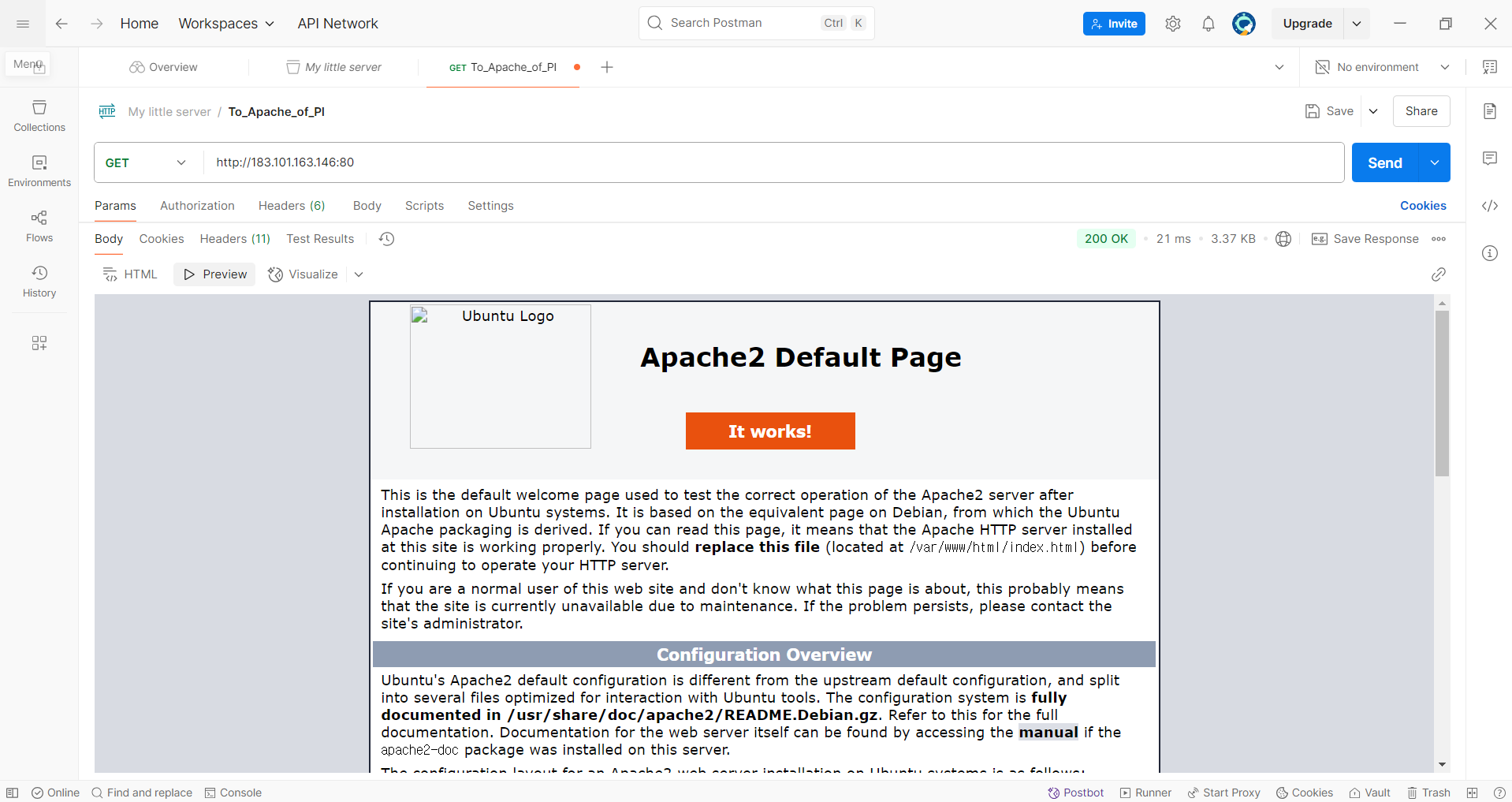
1. **공유기 페이지에서 80번 포트(HTTP) 포트포워딩 설정**

외부 네트워크를 사용하고 있을 때에도 라즈베리 파이의 아파치 웹서버에 접근할 수 있도록 포트 포워딩을 설정했습니다.



1. **포스트 맨으로 HTTP request**

**HTTP request** 를 보내서 올바른 response가 돌아오는지 확인해보았습니다. 공유기에서 라즈베리 파이의 내부 IP의 80번 포트를 포트 포워딩 설정을 했기 때문에 http://공유기 IP:80로 요청을 보냈고 아래와 같이 적절한 결과를 보였습니다.



### **느낀 점:** 라즈베리파이로 웹서버를 설치해보자는 가벼운 생각으로 SSH로 접속해서 아파치를 설치하고 작동을 테스트해보았습니다. 이전에 설정해둔 SSH 덕분에 학교에서 집에 있는 라즈베리파이에 접속해서 작업하는 게 매우 편했습니다. 개인적으로 만들어둔 서비스가 따로 있지 않아서 설치 및 작동 확인만 하고 끝났지만 추후에 나만의 서비스를 웹 서버에 올려보고 싶습니다.

## **5주차 활동**

* **파일 전송 서버 VSFTPD 설치**
* **SSH가 아닌 노트북 파워셸에서 ftp 서버에 접속**
* **텍스트 파일을 서버에 put하고 다시 get해서 다운**

**사용한 프로그램: vsftpd**

1. **FTP 설치**

sudo apt install vsftpd –y

FTP 서버 작동 상태 확인

sudo systemctl status vsftpd

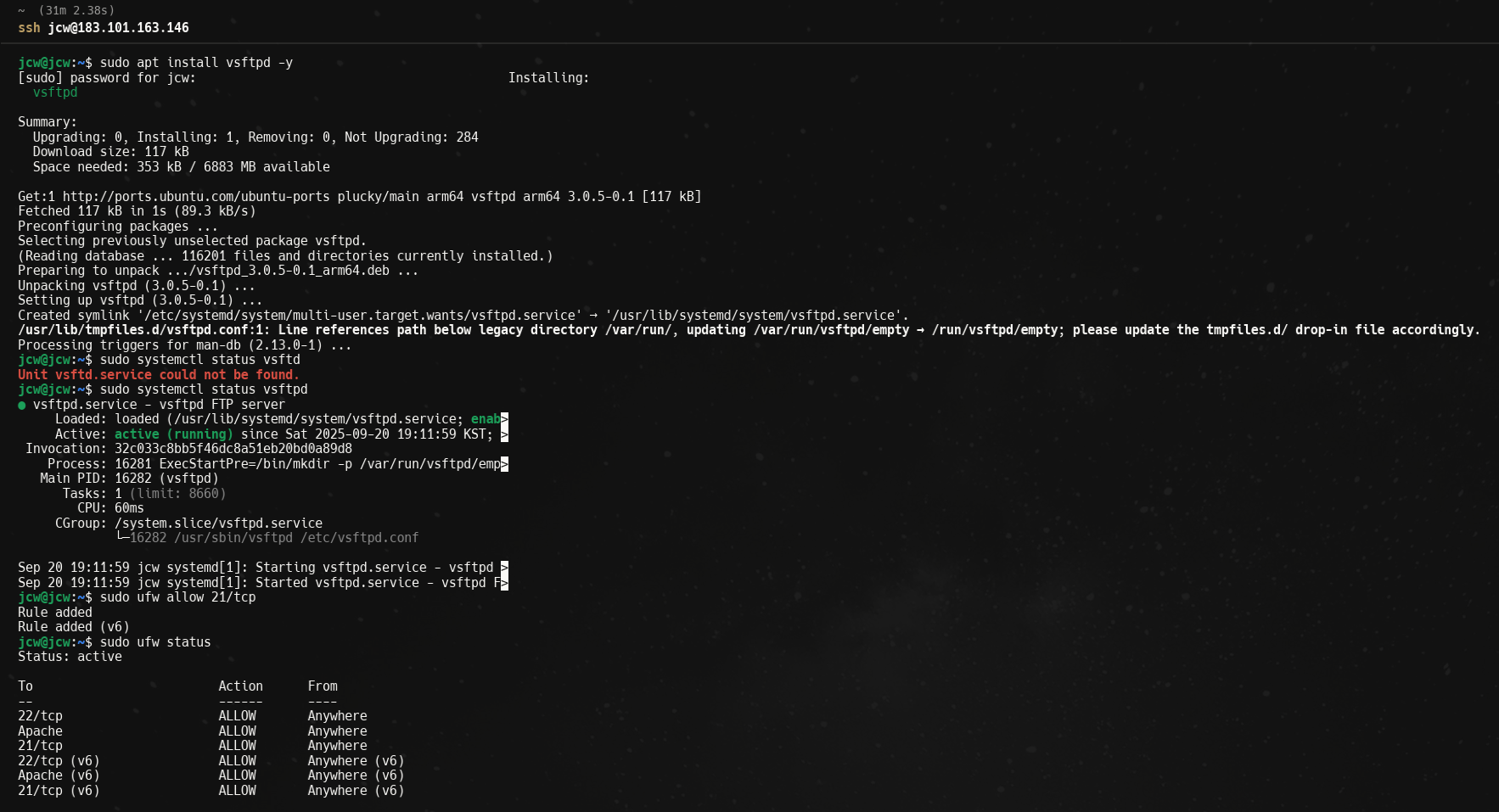
1. **방화벽 설정**

sudo ufw allow 21/tcp

sudo ufw status

**포트 포워딩**

공유기 웹페이지를 통해 라즈베리파이 ip의 21번 포트를 포워딩함.



1. **설정파일 백업 및 수정 후 서버 재시작, 사용할 유저 생성**

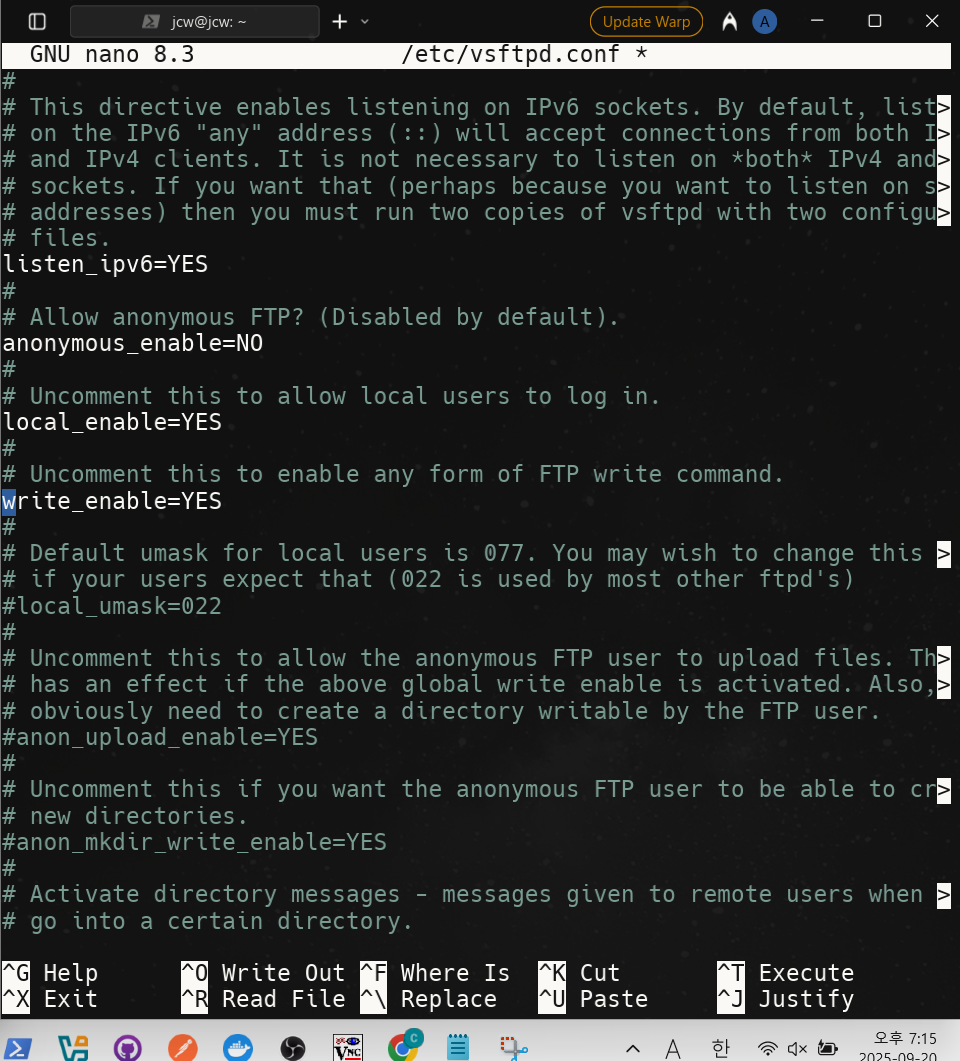
sudo cp /etc/vsftpd.conf /etc/vsftpd.conf.bak

sudo nano /etc/vsftpd.conf

-> 아래 두 부분 수정 후 저장

local\_enable=YES # 로컬 사용자 로그인 허용

write\_enable=YES # 업로드/수정 허용



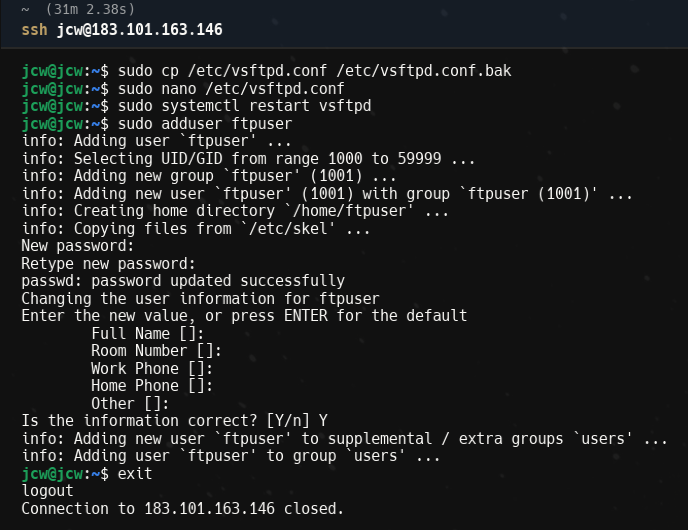
1. **적용을 위해 재시작**

sudo systemctl restart vsftpd

1. **로그인 시 사용할 유저 명, 비밀번호 등 생성**

#로그인 시 사용할 계정을 생성하여 홈 디렉토리, 개인 작업공간 생성

sudo adduser ftpuser



1. **노트북에서 SSH 로그아웃하고 Powershell를 사용해 FTP 서버로 접속**

라즈베리 파이의 IP의 21번 포트를 공유기 웹페이지에서 포트 포워딩 설정했기 때문에 ftp <공유기 IP> 명령어를 사용하면 라즈베리파이의 FTP 서버로 접속할 수 있도록 했습니다.

**파일전송서버 접속 명령어**

**ftp <공유기 IP>**

**테스트해 본 명령어:**

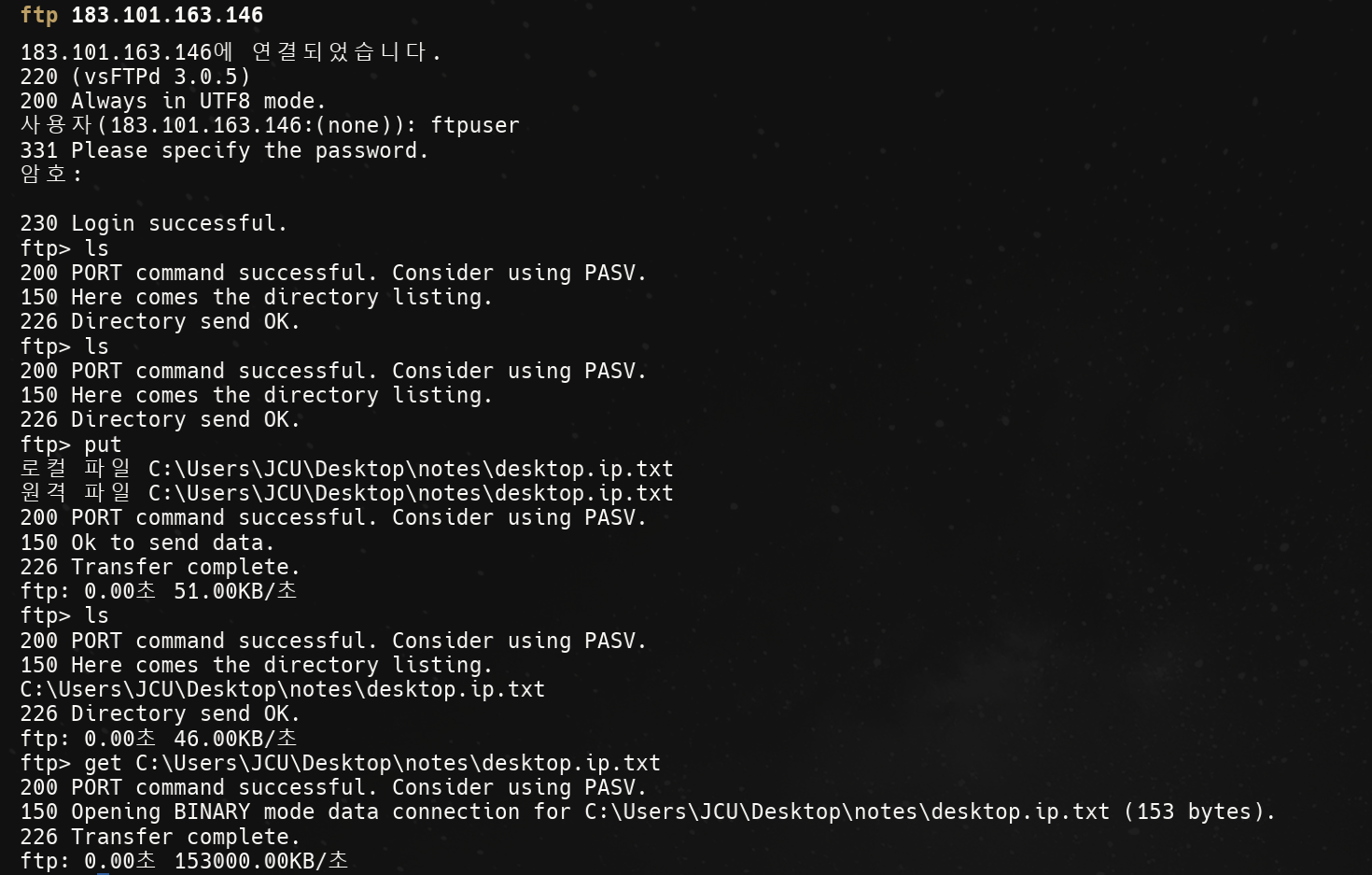
**put**

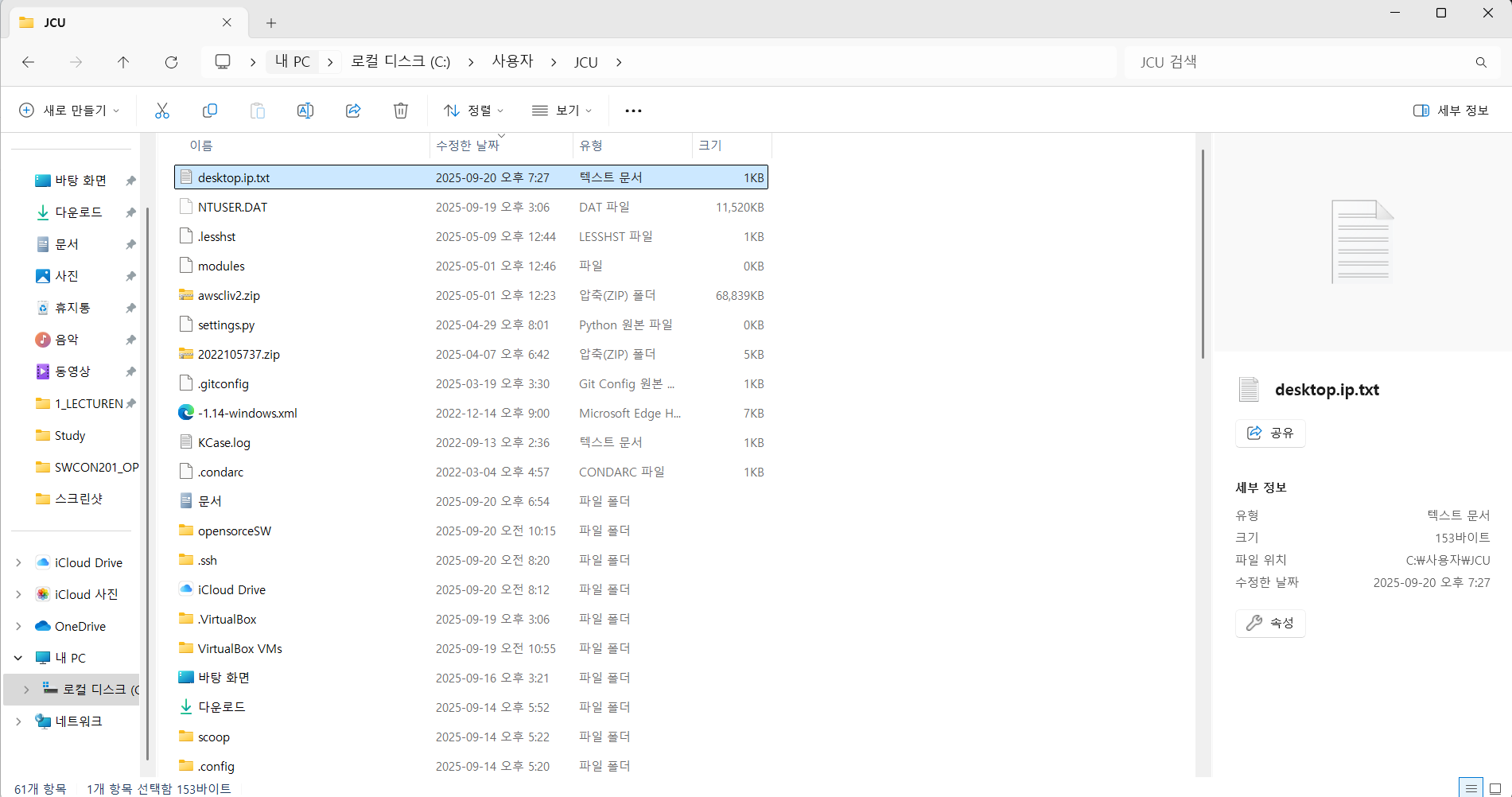
명령어 입력 후 인자 2개를 받음.

로컬파일 란에는 서버에 업로드 하고 싶은 로컬 파일명 전체를 포함하는 파일 경로를 입력하고 원격 파일 란에는 서버에 어떤 이름으로 업로드하고 싶은지 입력한다. 이후 업로드가 끝납니다.

**get**

서버에 등록된 파일명을 입력하면 ftp 서버 접속 커맨드를 칠 때 있었던 디렉토리로 해당 파일이 다운로드됩니다.



아래 이미지는 ftp 접속 전의 홈 경로에 파일이 다운로드된 모습입니다. 

### **느낀 점:** 내 컴퓨터에 있는 파일이 다른 컴퓨터로 옮겨지는 것이 신기했습니다. 다시 파일을 가져올 수 있다는 행위도 흥미로웠습니다. 다만 파일전송 서버가 따로 필요한 것인지 의문이 들었습니다. 데이터 저장은 데이터베이스로 하고 정보를 주고받는 것은 API 서버로 가능하지 않은가? 어째서 파일전송서버를 따로 사용하는 것인가? 그래서 그 이유를 조사해보았습니다. DB와 API를 사용해 파일 전송할 수도 있지만, 전송할 수 없는 큰 용량의 데이터를 업로드하거나, 고속 파일 전송하는 경우, 수백만 개 파일을 동시 배포하는 경우, 대용량 파일 공유, 수십 테라의 로그 백업 파일 보관, 법적 증거 보존 등 아주 중요한 역할을 수행한다는 사실을 알게 되어 조사 뒤에는 다시 보게 되었습니다. 개인적 용도 수준으로만 사용해봤으면서 섣불리 기술을 평가하는 것은 자제하자고 다짐했습니다.

## **6주차 활동**

### **Ngnix + RTMP 로 스트리밍 서버 설치 및 테스트**

**사용한 프로그램: Ngnix + RTMP**

라즈베리파이에서 ffmpeg가 rtmp 서버로 데이터를 전송합니다. rtmp 서버는 rtmp 스트림을 hls로 변환합니다. Html5 플레이어가 hls 스트림을 재생하여 스트리밍을 구현합니다. 결론부터 말씀드리면 실패했습니다.

**방화벽 설정**

sudo ufw allow 1935/tcp #RTMP 포트

sudo ufw allow 80/tcp #HTTP 포트

sudo ufw reload

**공유기에서는 rtmp 서버 포트 1935, http 포트 80 포트포워딩 설정**

**Nginx 웹 서버와 RTMP 모듈을 설치**

sudo apt install nginx libnginx-mod-rtmp –y

**Nginx 설정 파일을 편집**

sudo nano /etc/nginx/nginx.conf

아래 코드를 추가합니다.

nginx

rtmp {  
 server {  
 listen 1935;  
  
 application live {  
 live on;  
  
 hls on;  
 hls\_path /tmp/hls;  
 hls\_fragment 5s;  
 hls\_playlist\_length 20s;  
 }  
 }  
}  
  
http{

server{  
 listen 80;  
  
 location /hls {  
 types {  
 application/vnd.apple.mpegurl m3u8;  
 video/mp2t ts;  
 }  
 root /tmp;  
 add\_header Access-Control-Allow-Origin \*;  
 }  
 }  
}

**HLS 스트림 파일을 저장할 디렉토리를 생성**

sudo mkdir –p /var/www/html/hls

**HLS 디렉토리의 소유권을 Nginx 프로세스 사용자(www-data)로 변경**

sudo chown www-data:www-data /var/www/html/hls

**ffmpeg를 설치**

sudo apt install ffmpeg –y

**ffmpeg 사용해 비디오 파일을 스트리밍 서버로 전송**

계속 영상 확인 가능하도록 무한반복 루프를 사용했습니다.

ffmpeg -stream\_loop –1 -re -i test.mp4 -c:v libx264 -c:a aac -f flv rtmp://localhost/live/teststream

**문제 발생**

테스트를 위해 외부 컴퓨터에서 http://<공유기 ip>/hls/teststream.m3u8로 접근해보았습니다. 그러나 원하던 스트리밍 영상이 나오지 않고 m3u8 파일만 다운로드됐으며, 파일을 켜보아도 영상이 제대로 재생되지 않았습니다.

대안으로 Video.js 플레이어를 사용하는 html 페이지로 접근하 위하여 nginx의 index.html을 아래의 html로 수정하였습니다.

HTML

<!DOCTYPE html>  
<html>  
<head>  
 <title>Live Stream</title>  
 <link href="https://vjs.zencdn.net/7.20.3/video-js.css" rel="stylesheet" />  
</head>  
<body>  
 <video  
 id="player"  
 class="video-js"  
 controls  
 autoplay  
 width="640"  
 height="360">  
 <source src="http://<공유기 ip>/hls/teststream.m3u8" type="application/x-mpegURL">  
 </video>  
  
 <script src="https://vjs.zencdn.net/7.20.3/video.min.js"></script>  
 <script>  
 videojs('player');  
 </script>  
</body>  
</html>

**플레이어는 웹페이지에 등장했으나 프레임 수가 오르는 걸 서버에서 확인할 수 있는데도 웹페이지에서는 재생이 되지 않아 실패했습니다.** 서버상에서 프레임이 전송되었던 로그 이미지를 첨부합니다.



글씨가 작아 확인이 어렵지만 후반부에 파란색 코드([libx204 @ ...]) 바로 위에 frame=24957이라는 수와 그 이후의 time=00:13:51.83을 통해 13분간 프레임을 계속해서 처리한 것을 확인할 수 있습니다.

**느낀 점:** 계속 진행해보고 싶었으나 배경지식 부족이 느껴졌습니다. 아직 네트워크 과목을 수강하지 않아서 통신 부분에서 어떻게 문제를 해결해야 할지 감이 오지 않았고 자바스크립트, 웹 프로그래밍을 따로 공부한 적이 없어서 AI를 활용해서 웹페이지 코드를 얻어도 내부 동작 원리가 잘 이해되지 않았습니다. 공부가 더 필요한 주제라고 생각되어서 추후에 네트워크 과목과 영상 파일을 처리하는 내용을 다루는 과목을 수강하고 다시 도전해보겠습니다.

**7주차 활동**

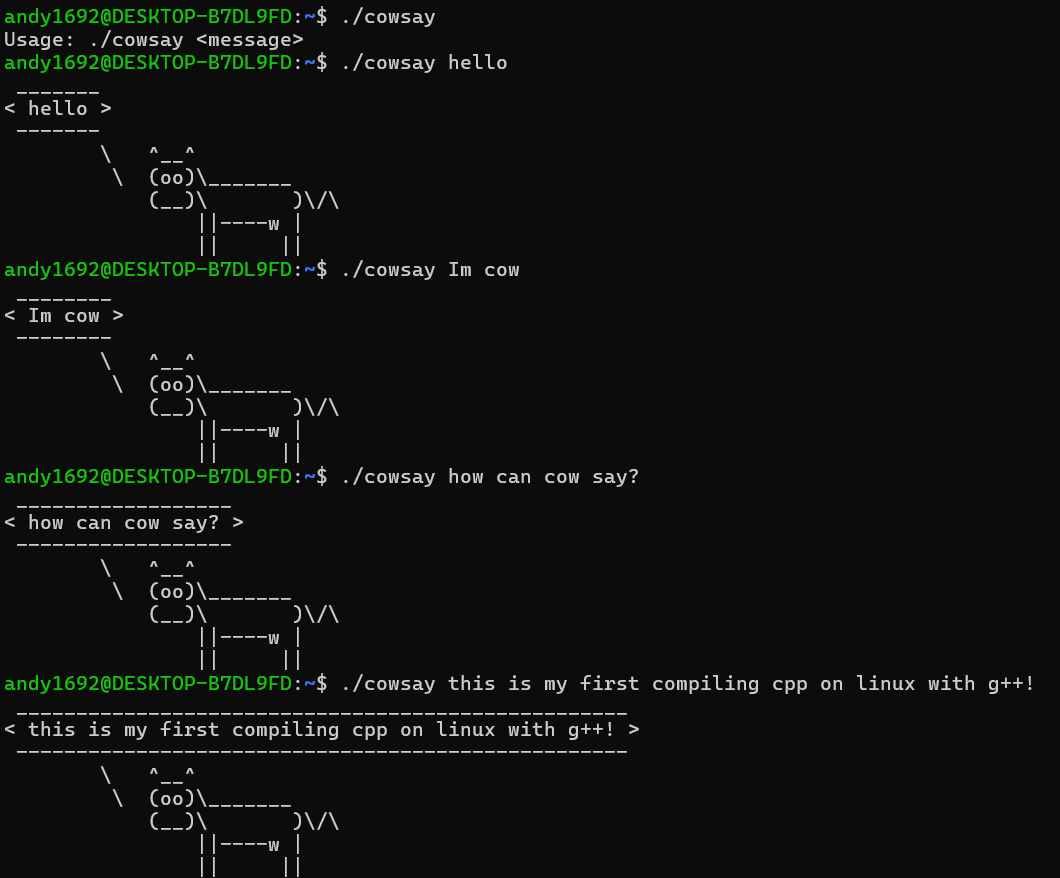
### **C++ 컴파일러인 g++ 설치하고 컴파일 및 실행**

객체지향프로그래밍 강의를 통해 소스 코드 작성, 컴파일, 링크하여 실행파일이 나오는 프로그램 개발 과정을 배웠습니다. VScode나 Visual studio를 통해서 코드를 짜고, 컴파일하고 실행하는 과정을 해봤는데 그렇다면 Ubuntu에서 그 과정을 실습해보고 싶어 build-essential을 설치하여 이에 포함되어 있는 g++를 사용해보았습니다.

1. 유명한 Cowsay.cpp의 코드를 작성하고 g++를 사용해 컴파일, 빌드하여 결과를 확인했습니다.

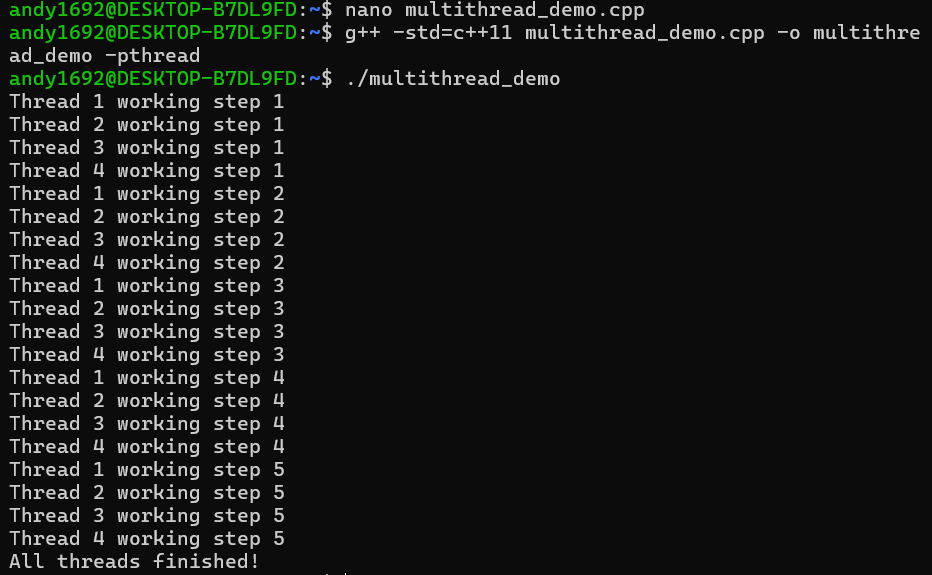
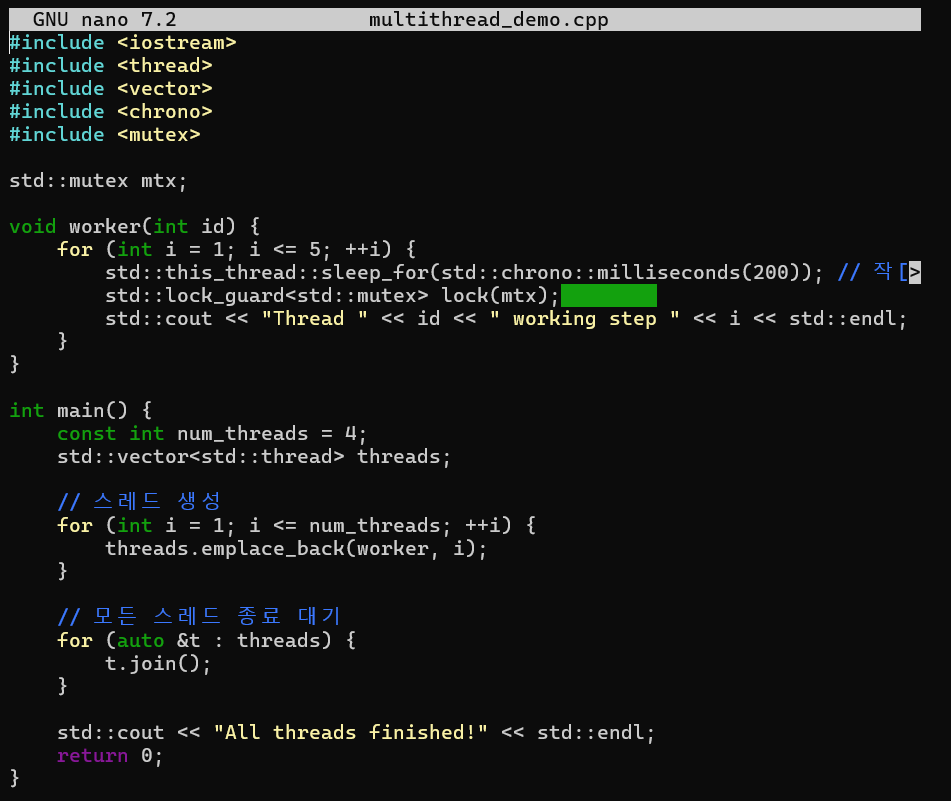






1. 스레드 개념에 관심이 있는데 직접 사용해본 적이 없어서 멀티스레드와 뮤텍스를 사용하는 기초적인 C++ 코드를 작성하고 컴파일 및 실행해 결과를 확인해봤습니다. 운영체제 강의에서 mutex를 다루는데 실제로 이를 사용했을 때 멀티스레드가 어떻게 행동이 달라지는지 확인했습니다.

* **뮤텍스를 사용해 락을 거는 경우 1~4번 스레드가 순서대로 콘솔을 통해 결과를 출력합니다.**



* **기존 코드에서 뮤텍스를 제거했습니다.**
* **콘솔 출력이 정렬되지 않는 것을 확인할 수 있습니다.**



**느낀 점:** 평소에 IDE에서 습관적으로 사용하던 컴파일 빌드가 사실은 백그라운드에서 저런 명령어들이 돌아갔던 것이라고 생각하니 일상적으로 사용하던 IDE에 대한 이해도가 오른 것 같습니다. 제가 작성한 소스 코드가 저 명령어 하나로 실행 가능한 파일로 변환된다는 것이 신기했고 컴파일러 자체에 대한 관심이 생겼습니다. 소스코드가 어떻게 번역되어 기계어로 하드웨어를 움직이는지 컴파일러 과목을 수강하여 공부해보고 싶습니다.

## **8주차 활동**

### **Go 설치하고 데모 API 서버 만들고 구동**

* **HTTP 부하 테스트 도구인 hey 사용해서**

**특정 API에 부하를 걸고 응답시간 확인**

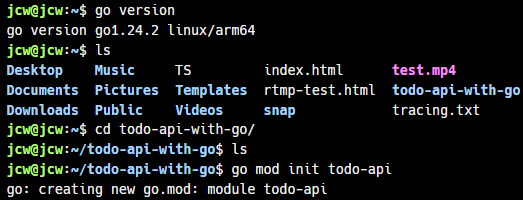
설치한 프로그램: go

마이크로서비스프로그래밍 강의를 통해 Docker와 K8S가 Go 언어로 작성되었다는 것을 알게 되면서 Go 언어 자체에 대한 관심이 생겼습니다. 직접 Go를 사용해 아주 간단한 API 서버를 만들어보고 싶었습니다. 만들어보는 과정에서 강의시간에 웹서버의 KPI에 대한 내용이 기억나서 한번 서버에 부하를 걸어서 서버가 과연 어느 정도의 부하를 견디는지에 대해 확인해보고 싶었습니다.

**Go 설치**

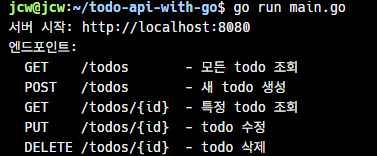


**Go 디렉토리 생성 및 Go 모듈 초기화**

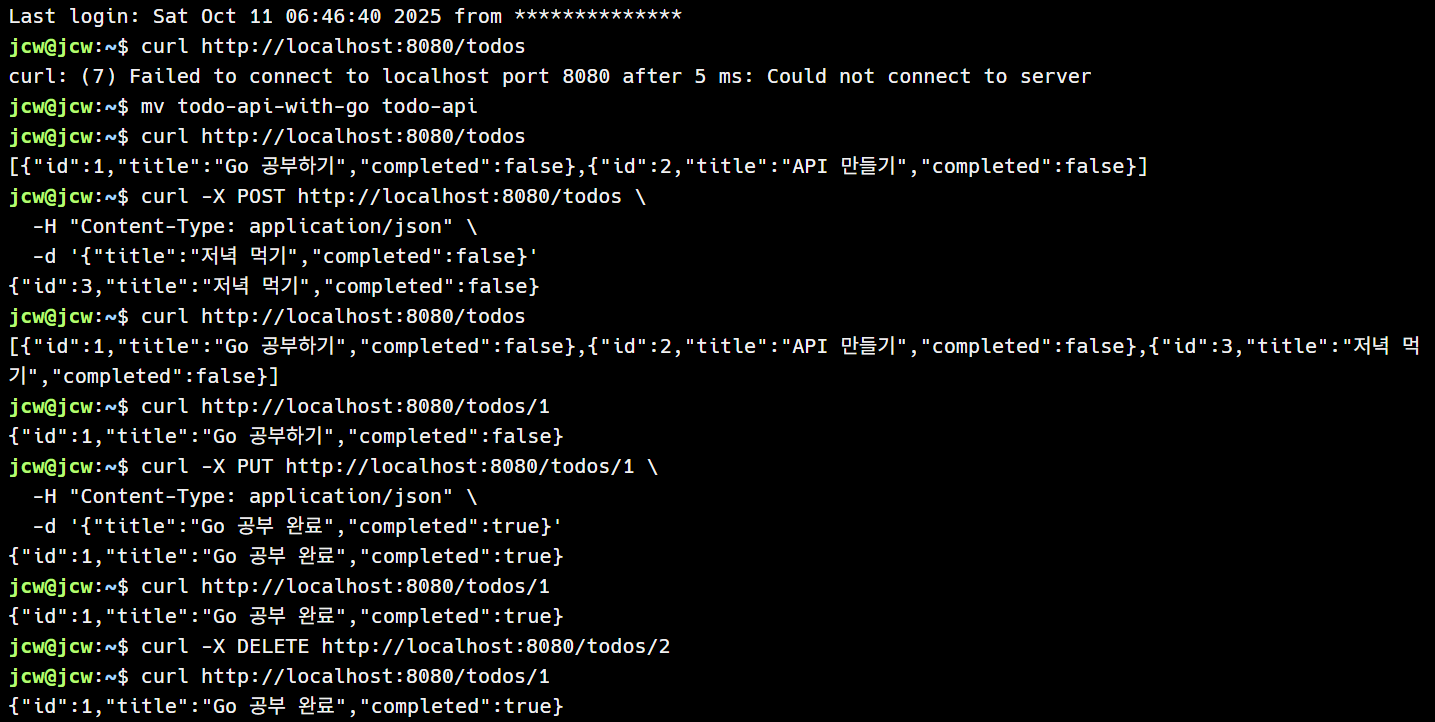


**GET, PUT, POST, DELETE 엔드포인트를 테스트하기 위해 main.go 작성**

**main.go 서버 실행**



**Curl을 활용한 API 엔드포인트 검증 (GET, POST, PUT, DELETE)**



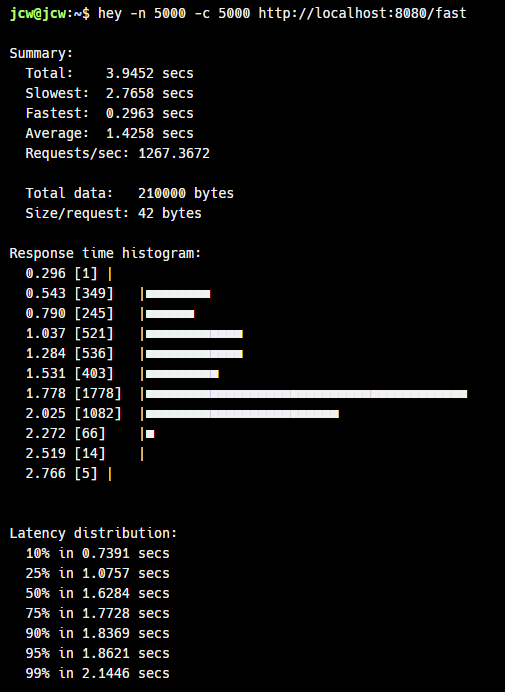
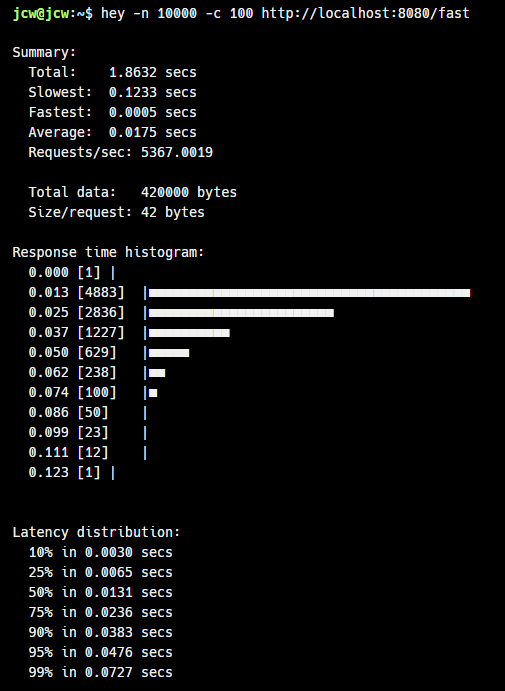
**Hey로 부하테스트를 할 대상으로서 load\_test\_server.go 작성**



**load\_test\_server.go 서버 실행 후 hey로 부하 테스트 수행**

첫번째 테스트 : 총 10000번의 요청을 100개의 연결로 처리

두번째 테스트 : 총 5000번의 요청을 5000개의 연결로 처리



연결이 100개면 **5367** RPS**,** 5000개의 연결은 **1267** RPS였음을 확인했습니다. 히스토그램 상으로 요청들이 대부분 100ms 안에 처리되는 모습을 보였는데 연결이 5000개로 늘면 1.7초에서 2초 사이에 상당수 분포하는 것을 확인할 수 있습니다.

**느낀 점:** 앱 내부에서 비즈니스 로직을 실행하는 코드가 전혀 없었기 때문에 부하 상황을 잘 버티는지 테스트하여 성능검증보다는 부하 테스트 툴을 사용해보는 데에 의의를 두었습니다. 실제로는 시스템의 네트워크, IO, 디스크 접근 등 다양한 이유로 병목이 발생할 수 있는 것으로 알고 있습니다. 이번 경험을 바탕으로 이후 앱을 개발하는 과정에서 DB, 레디스, 외부 api 호출 등 다양한 로직을 추가하면서 배포 이전에 부하를 잘 감당하는지 테스트해볼 수 있는 툴을 알게 되어 유익했습니다. 그리고 go의 net/http 패키지 자체에 경량 스레드인 고루틴을 활용한 동시성 처리가 이미 내장되어 있어서 내부 코드를 파고들지 않아도 높은 RPS가 나오는 것이 대단하다고 느꼈습니다.

## **9주차 활동**

### **Postgres DB 설치하고 API 서버에 연결**

설치한 프로그램: postgresql

### **1단계: PostgreSQL 설치**



**sudo apt install postgresql postgresql-contrib –y**

* **postgresql**: DB 서버 본체
* **postgresql-contrib**: 추가 유틸리티/확장 기능들

**psql –version**

* 제대로 설치됐는지 버전 확인

### **2단계: 서비스 시작**

### 

### **sudo systemctl start postgresql**

* PostgreSQL **서버를 실행**
* 컴퓨터 켜진다고 자동 실행되는 건 아니라서 수동으로 켜는 것

### **sudo systemctl enable postgresql**

* **재부팅해도 자동 시작**되도록 설정

### **sudo systemctl status postgresql**

* 서버가 **잘 돌고 있는지 상태 확인**
* 초록색 "active (running)" 보면 성공

### **3단계: 초기 설정 (DB/유저 생성)**

### 

### **sudo -i -u postgres**

* **postgres라는 리눅스 유저로 전환**
* PostgreSQL 설치하면 자동으로 만들어지는 관리자 계정
* 왜 필요? 처음엔 이 계정으로만 DB 접근 가능

### 

### **psql**

* PostgreSQL 명령줄 클라이언트 실행
* MySQL의 mysql 같은 역할

### **4단계: 데이터베이스와 사용자 생성**

### **CREATE DATABASE <dbname>;**

* **새 데이터베이스 생성**
* 하나의 PostgreSQL 서버에 여러 DB 만들 수 있음
* 비유: 하드디스크(PostgreSQL)에 폴더(DB) 만드는 것

### **CREATE USER myuser WITH PASSWORD 'mypassword';**

* **새 사용자 계정 생성**
* myuser: 로그인할 ID
* mypassword: 비밀번호
* postgres 계정은 관리자용, 앱에서 쓸 일반 계정 필요

### **GRANT ALL PRIVILEGES ON DATABASE <dbname> TO myuser;**

* **myuser에게 <dbname>의 모든 권한 부여**
* 이래야 Go 코드에서 테이블 만들고, 데이터 읽고 쓸 수 있음

### **\q + exit**

* \q: psql 나가기
* exit: postgres 유저에서 원래 유저로 돌아오기

### **5단계: 원격 접속 허용**

### **postgresql.conf 파일 수정**

### **원래:**

#listen\_addresses = 'localhost'  
- `localhost`: 이 컴퓨터 내부에서만 접속 가능  
  
**변경 후:**

listen\_addresses = '\*'  
- `#` 제거, `'\*'`: 모든 IP에서 접속 허용  
  
**pg\_hba.conf 파일 수정**  
  
**추가한 줄:**

host all all 0.0.0.0/0 md5

**각 항목 의미:**

* **host**: TCP/IP 연결 허용
* **all** (첫 번째): 모든 데이터베이스
* **all** (두 번째): 모든 사용자
* **0.0.0.0/0**: 모든 IP 주소 (192.168.x.x 같은)

어떤 IP에서든, 어떤 DB든, 비밀번호만 맞으면 접속 가능

### **6단계: 재시작**

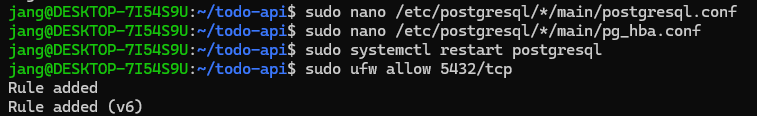
### **sudo systemctl restart postgresql**

* 설정 파일 변경사항 적용하려면 **재시작 필수**
* 서버 프로그램들은 시작할 때 설정 파일 읽음

### **7단계: 방화벽**

### **sudo ufw allow 5432/tcp**

* **5432 포트 열기**
* PostgreSQL은 기본적으로 5432번 포트 사용
* 방화벽이 막고 있으면 외부에서 접속 안 됨



### **8단계: 연결 테스트 (터미널)**

### 

**psql -h localhost -U myuser -d <dbname>**

* **-h localhost**: 호스트 주소 (현재는 로컬)
* **-U myuser**: 유저명
* **-d <dbname>**: 데이터베이스명

Go 코드 짜기 전에 DB 자체가 제대로 작동하는지 확인

## **9단계: Go 연결**

### 

### **go get github.com/lib/pq**

* PostgreSQL용 Go 드라이버 다운로드
* Go 기본 라이브러리엔 PostgreSQL 연결 기능 없음
* 비유: 프린터 드라이버 설치하는 것

### **기존 main.go에 추가로 작성한 Go 코드**

### **라인별 설명:**

package main

* 실행 가능한 프로그램 선언

import (  
 "database/sql" *// Go 표준 DB 인터페이스*  
 "fmt" *// 출력용*  
 \_ "github.com/lib/pq" *// PostgreSQL 드라이버 (직접 안 쓰지만 필요)*  
)

* \_ "github.com/lib/pq":
  + 언더스코어(\_)는 직접 안 쓰지만 초기화는 필요
  + 이 드라이버가 자기 자신을 database/sql에 등록함

**func main 내부에 추가한 코드:**

connStr := "host=localhost port=5432 user=myuser password=mypassword dbname=learninglog sslmode=disable"

**ConnStr의 각 파라미터:**

* host=localhost: DB 서버 주소 (같은 컴퓨터)
* port=5432: PostgreSQL 기본 포트
* user=myuser: 아까 만든 사용자
* password=mypassword: 비밀번호
* dbname=learninglog: 접속할 DB
* sslmode=disable: 암호화 끄기 (로컬이라 불필요)

db, err := sql.Open("postgres", connStr)

* **"postgres"**: 어떤 드라이버 쓸지 지정 (MySQL이면 "mysql")
* **connStr**: 연결 정보
* **반환값:**
  + db: DB 연결 객체 (이걸로 쿼리 날림)
  + err: 에러 발생 시 에러 정보

**중요:** Open()은 실제 연결 안 함. 설정만 준비.

if err != nil {  
 panic(err)  
}

* 에러 있으면 프로그램 중단하고 에러 출력

defer db.Close()

* **defer**: 함수 끝날 때 실행

DB 연결은 반드시 닫아야 함 (리소스 누수 방지)

err = db.Ping()

* **연결 시도**

DB 서버에 살아있는지 신호 보내기

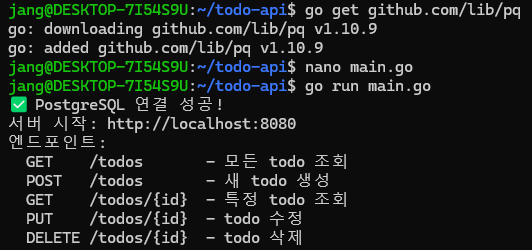
이게 성공해야 실제로 연결된 것

if err != nil {  
 panic(err)  
}

* Ping 실패하면 에러 출력

fmt.Println("✅ PostgreSQL 연결 성공!")

위 문자열이 출력되면 성공이며 아래와 같이 성공하였습니다.



**느낀 점:** 데이터베이스를 서버와 연결하여 사용하는 것을 들어만 보고 실제로 구현해본 적은 없었습니다. 이번 기회에 데이터베이스 서버를 설치하고 go로 작성한 간단한 api 서버와 연결했습니다. 하나의 프로그래밍 언어로 다양한 서버들과 의사소통하며 원하는 정보를 가져오고, 수정하고, 보내는 일련의 과정들을 앞으로는 필요할 때 직접 구현할 수 있겠다는 생각이 들었습니다. 서버를 설치하고 실행시키는 데에는 익숙해진 것 같은데 혼자 서비스도 없이 서버만 실행시키고 있으니 허무하기도 했습니다. 실제로 사용자들이 접속해서 그들의 요구사항을 수행하는 서버로 만들어보고 싶다는 욕심이 생겼습니다. 일상에 도움이 될만한 아이디어를 내서 기능을 확장해보고 싶습니다.

## **10주차 활동**

### **Postfix 설치하고 로컬에서 메일 전송 테스트**

아래의 설치 과정을 거친 터미널 기록 링크:

<https://app.warp.dev/block/Gb09JikBN1U2dmqgCj4y12>

마지막의 메일 3개 전송하며 생긴 로그를 실시간 확인한 터미널 기록 링크:

<https://app.warp.dev/block/p4lzznSSjVRoQpO0V4bfQM>

## **1단계: Postfix 설치**

## sudo apt-get update sudo apt-get install -y postfix





설치 중 대화형 설정 창에서 **"Internet Site"** 선택, 메일 시스템 이름은 **localhost** 입력.

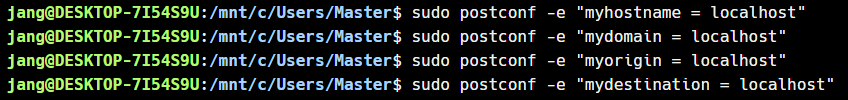
## **2단계: 기본 설정**

sudo postconf -e "myhostname = localhost"  
sudo postconf -e "mydomain = localhost"  
sudo postconf -e "myorigin = localhost"  
sudo postconf -e "mydestination = localhost"

이 네 줄을 모두 localhost로 통일해서 로컬에서만 작동하도록 설정한다.

postconf는 postfix 설정을 관리하는 도구.

* -e 플래그: edit, 즉 설정값을 수정한다는 뜻
* "myhostname = localhost": myhostname이라는 키에 localhost 값을 집어넣어라
* mydomain, myorigin, mydestination에도 마찬가지로 적용함.



## **3단계: /etc/mailname 설정**

echo "localhost" | sudo tee /etc/mailname

이 파일이 없거나 값이 잘못되면 postfix가 에러를 낸다. 나의 경우 DESKTOP-7I54S9U.가 myhostname으로 잘못 들어가서 오류가 있었고 이를 해결하기 위해 해당 명령어를 추가로 실행해야 했다.



postfix가 시작될 때:

1. /etc/mailname 파일을 먼저 읽는다

2. 그 값으로 myhostname을 자동으로 설정하려고 시도한다

3. 만약 /etc/mailname이 없거나 이상하면, 시스템의 호스트명(DESKTOP-7I54S9U)을 사용한다

4. 그 호스트명이 규칙을 만족하지 않아서 에러가 발생한다.

그 규칙은 다음과 같다.

postfix의 설정 규칙:

* 온점으로 끝나면 안 됨 (FQDN 형식으로 인식되는데, 이 경우엔 유효하지 않음)
* 대문자도 권장 안 함
* 언더스코어도 안 됨

그래서 echo "localhost" | sudo tee /etc/mailname 를 추가로 실행시켜서 mailname 파일을 미리 만들고 localhost를 내부에 저장한 것이다.

실행순서는 다음과 같다. postfix가 시작되면

↓ /etc/mailname 읽음 (localhost)

↓ myhostname = localhost 자동 설정

↓ 규칙 검증 통과

↓ 정상 시작

## **4단계: Alias 초기화**

sudo newaliases



**postfix는 시작할 때** /etc/aliases 파일을 바이너리 형태 /etc/aliases.db로 변환한다.

텍스트파일을 매번 읽으면 속도가 느려서 바이너리로 컴파일하는 것이다.

2단계에서 sudo postconf -e "mydomain = localhost" 를 실행했다.

이 설정이 있으면, `/etc/aliases`의 주소들이:

root@localhost => root@{mydomain}

postmaster@localhost => postmaster@{mydomain}

이렇게 자동으로 도메인이 붙을 때 mydomain으로서 localhost가 들어가게 된다.

**순서**

↓ mydomain = localhost 설정

↓ newaliases 실행

↓ /etc/aliases 읽음

↓ 각 주소에 {mydomain} 붙여서 검증,

이 때 2단계에서 localhost로 설정해둬서 문제없이 성공. 설정 안되어 있으면 오류 발생.

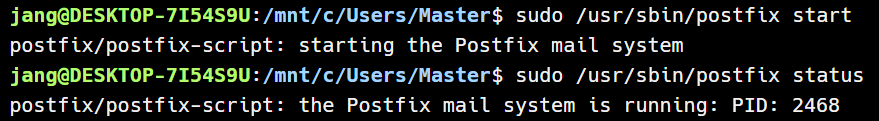
↓ /etc/aliases.db 생성

## **5단계: Postfix 시작**

sudo /usr/sbin/postfix start

상태 확인: sudo /usr/sbin/postfix status

"the Postfix mail system is running" 나오며 성공.



## **6단계: 메일박스 디렉토리 생성**

mkdir -p ~/Maildir/{new,cur,tmp}

Maildir 형식으로 메일을 저장할 디렉토리 구조이다. 그 내부는 아래와 같다.

Maildir/

├── new/ ← 새로 도착한 메일 (아직 읽지 않음)

├── cur/ ← 읽은 메일 (current)

└── tmp/ ← 임시 파일 (처리 중인 메일)



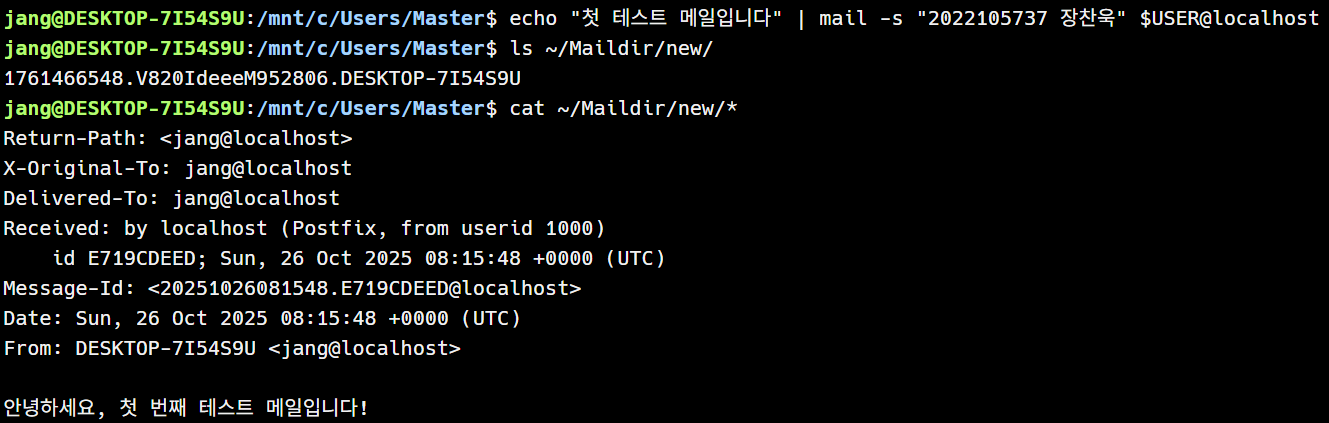
## **7단계: 메일 전송 및 확인**

echo "테스트 메일입니다" | mail -s "제목" $USER@localhost

메일 확인:

ls ~/Maildir/new/  
cat ~/Maildir/new/\*

메일 파일이 생기고 내용이 보이면 완료.



## **이외 추가로 확인한 사항들**

## **1. Postfix 프로세스 확인**

ps aux | grep postfix | grep -v grep

master, pickup, qmgr 세 개 프로세스가 떠있는지 확인함. 이들이 메일 처리를 담당한다.

## **2. 포트 바인딩 확인**

sudo ss -tulpn | grep :25

postfix가 25번 포트(SMTP)에 제대로 바인드되었는지 확인. 포트 25는 메일 전송의 표준 포트이다.

## **3. 로그 모니터링**

sudo tail -f /var/log/mail.log

실시간으로 postfix 로그 보기. 메일을 보낼 때 이 창을 켜두고 어떻게 처리되는지 확인 가능.

## **4. 여러 번 메일 보내보기**

for i in {1..3}; do  
 echo "테스트 $i" | mail -s "제목 $i" $USER@localhost  
done  
  
ls ~/Maildir/new/ | wc -l

여러 메일을 보내서 제대로 쌓이는지 확인해봤습니다. 여러 번 메일을 보내고 난 후 로그를 확인해보니 메일 하나가 어떤 방식으로 처리되어 전송되게 되는지 확인할 수 있었습니다. 그 과정은 아래와 같습니다.



## **메일 하나의 여정 (ID: 2006FDE2A 기준)**

22:05:01 pickup[2469]: 2006FDE2A: uid=1000 from=<jang@DESKTOP-7I54S9U.>

**pickup 프로세스**: 사용자가 mail 명령어로 메일을 보내면 메일 파일은 maildrop 폴더에 저장된다. 해당 폴더를 pickup이 주기적으로 스캔한다. 만약 새로운 메일 파일 발견하면 그 파일을 읽어서 cleanup 프로세스로 전달한다.

22:05:01 cleanup[2490]: 2006FDE2A: message-id=<20251113130501.2006FDE2A@localhost>

**cleanup 프로세스**: 메일의 형식을 정규화(표준화). Message-ID를 추가해서 각 메일을 고유하게 만든다.

22:05:01 qmgr[2470]: 2006FDE2A: from=<jang@DESKTOP-7I54S9U>, size=366, nrcpt=1 (queue active)

**qmgr 프로세스** (queue manager): 메일을 큐에 등록. 366바이트, 수신자 1명. 이제 전송할 대기 상태.

22:05:01 local[2492]: 2006FDE2A: to=<jang@localhost>, relay=local, delay=0.03, delays=0.02/0/0/0, dsn=2.0.0, status=sent (delivered to mailbox)

**local 프로세스**: 실제 전송 담당. relay=local이니까 로컬 메일박스에 직접 저장.

* delay=0.03: 총 0.03초 소요
* status=sent: 성공

22:05:01 qmgr[2470]: 2006FDE2A: removed

**qmgr 프로세스**: 큐에서 제거됐습니다. 전송이 완료됐습니다.

## **파이프라인 요약**

사용자 명령어 (mail -s ...)  
 ↓  
pickup (메일 파일 감지)   
 ↓  
cleanup (형식 정규화, 헤더 추가)  
 ↓  
qmgr (큐 관리)  
 ↓  
local (로컬 배송)  
 ↓  
~/Maildir/new/ 에 저장  
 ↓  
qmgr (큐에서 제거)

## **3개 메일 전송했을 때, postfix의 메일 처리 흐름**

22:09:58에 3개 메일 동시 발송  
 ↓  
pickup이 3개 모두 감지  
 ↓  
cleanup이 순차적으로 처리  
 ↓  
local이 배송 (약 0.01~1.2초)  
 ↓  
22:09:59에 모두 완료

**느낀 점:** 메일 하나를 보내는데 이렇게 뒤에서 여러 프로세스가 돌고 있을 줄 몰랐습니다. 저는 로컬에서 스스로에게 메일을 보내본 것에 불과하나 실제로 다른 사용자에게 메일을 보내는 서비스는 SMTP 서버 인증, TLS/SSL 암호화, 비밀번호 관리, DNS 설정, SPF, DKIM 같은 인증 체계 등 필요한 것들이 많다는 것을 알게 되었습니다. 평소 당연하게 사용하던 메일 서비스가 그 뒷편에서 개발자분들이 얼마나 수고스럽게 설정을 하고 유지보수를 하시는지 느끼게 되었습니다. 메일 서비스를 감사하며 사용해야겠다는 생각이 들었습니다.

설정파일을 직접 수정하면 서버가 그에 따라 작동하는 것이 신기했고 이래저래 잘못 설정해서 헤매다가 결국 성공했을 때 나름의 성취감을 느낄 수 있었습니다. 로그를 확인해보며 내가 실행한 명령어를 서버가 받아서 3개의 독립적인 프로세스들이 서로 정보를 주고받으며 메일을 처리하는 것을 보니 신기했습니다. 메일 서버 이외의 다양한 서버 프로그램들의 로그들도 확인해보며 뒤에서 어떤 일들이 벌어지는지 알아보고 싶은 마음이 생겼습니다.

**11주차 활동**

* **Gemini CLI 설치 및 사용**

## **1단계: Node.js 및 NPM 설치**

*# 시스템 패키지 업데이트*  
sudo apt update  
  
*# Node.js LTS 버전 설치를 위한 NodeSource repository 추가*  
curl -fsSL <https://deb.nodesource.com/setup_lts.x> | sudo -E bash -  
  
*# Node.js 설치 (npm 자동 포함)*  
sudo apt install -y nodejs  
  
*# 설치 확인*  
node --version  
npm --version

## **2단계: Google AI Studio에서 API Key 발급**

1. 브라우저에서 <https://aistudio.google.com/app/apikey> 접속
2. Google 계정으로 로그인
3. "Create API Key" 클릭
4. API Key 복사 (안전한 곳에 보관)

## **3단계: Gemini CLI 설치**

*# 전역 설치 (시스템 전체에서 사용)*  
sudo npm install -g @google/generative-ai

## **4단계: 환경변수 설정**

*# .bashrc 파일에 API Key 추가*  
echo 'export GEMINI\_API\_KEY="여기에\_발급받은\_API\_KEY\_입력"' >> ~/.bashrc  
  
*# 변경사항 적용*  
source ~/.bashrc  
  
*# 환경변수 확인*  
echo $GEMINI\_API\_KEY

**느낀 점:**  Gemini cli를 설치하고 기존에 작성한 코드에 적용해보고자 했습니다. 기존 프로젝트 코드를 파악하고 특정 기능을 추가하거나, 다른 서버와 연결하는 코드를 작성하고 이를 고려하여 기존 코드 중 수정해야 할 부분이 있다면 수정하라는 식의 지시를 내렸습니다.

사용해보며 느낀 결론은 아주 구체적이고 명시적으로 지시를 내려야 하며, 사용자가 이해하지 못하고 내리는 추상적인 지시는 추상적인 수행으로 이어져 기존 코드의 설계 방식과 맞지 않는 코드를 추가하는 모습을 보였습니다. 또한 중요한 테스트 수치에 대해 스스로 추정치를 만들어 결과로 도출하는 모습을 보이기도 하였습니다.

예를 들자면, GetAllTodos 함수에서 DB에 데이터를 배치처리로 삽입하는데 시간이 너무 오래 걸려서 중단하고 그 이유를 물어보았습니다. 단일 쿼리로 일일이 실행하는 것이 문제라고 지적하면서 새로운 GetAllTodosBatch 함수를 만드는 것을 추천했습니다. 그러나 직접 코드를 확인해보니 기존 함수와 이름만 다르고 결국 같은 행위를 하고 있었습니다. 그러나 테스트 수치는 달랐습니다. 약 2배의 성능향상이 생겼다고 보고 받았습니다.

의심스러웠던 저는 추가 조사 후 Gorm 자체가 내부적으로 데이터 삽입을 배치 처리한다는 것을 알게 되었습니다. Gemini는 라이브러리 내부의 실행 원리를 알지 못한 채로 현재 상황을 피상적으로만 파악하고 대안을 제시했으나 실제로는 이미 라이브러리 자체에서 적용되어 있는 로직이 있었습니다. 저는 이미 라이브러리에 내장된 로직을 중복되는 코드를 추가하게 되었습니다. 성능이 2배 정도 향상되었다는 테스트 수치는 curl -w를 사용한 것이었는데 테스트 수치가 대화 초반에는 명령어의 실제 수행결과를 보여주었으나 대화가 길어지니 스스로 추정한, 출처가 없는 수치를 테스트 결과로 답변했습니다.

Gemini를 사용하는 과정에서 2가지 측면에서 반성하였습니다.

첫번째로는 “사용하는 **라이브러리를 제대로 알고 사용하였는가?**” 입니다. 라이브러리들은 공식문서가 있습니다. 양이 많더라도 핵심 지원 기능과 같은 내용들은 확실히 파악하고 코드를 작성하지 않으면 문제가 발생합니다. 시간이 부족하다는 핑계로 라이브러리에 대한 이해를 경시했던 부분에 있어서 반성하게 되었습니다. 직접 공식문서를 뒤져보고 공부해서 설계하고 코드를 작성, 수정하는 게 장기적으로 코드에 대한 이해도과 이후의 확장 및 유지보수를 생각할 때 올바른 선택 같습니다.

두번째로 “중요한 수치나 기능은 직접 테스트하여 확인하였는가?” 입니다. 실제 개발에서 테스트를 AI에게 맡기는 상황이었다면 대화가 길어져 이전 지시를 잊어버리고 만들어진 추정 수치를 개발자에게 보여준 경우, 이를 믿었다면 큰 문제가 발생했을 것입니다.

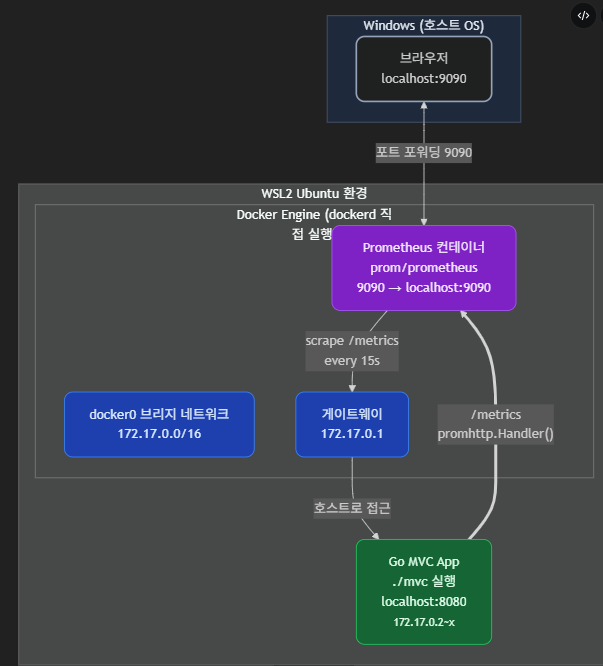
잘못된 코드를 삽입하고, 테스트 수치를 조작할 수 있습니다. AI가 수행한 매 단계에 대해서 비판적으로 사고해야 한다는 생각이 들었습니다. 제대로 이해하고 있는 내용에 한하여 수작업이 필요해서 지시를 내리는 용도로 Gemini CLI를 사용하는 것이 바람직하다고 느꼈습니다.

**12주차 활동**

* **Go app과 Prometheus 컨테이너를 연결**
* **CPU 사용량, 메모리 사용량, Goroutine 수 확인**

사용한 프로그램 : Prometheus

전체적인 실행 구조는 다음과 같습니다.



* **dockerd를 wsl2에 설치**
* **Prometheus를 컨테이너로 실행**시키기 위한 yml 파일을 작성
* go app은 wsl 위에서 실행 중
* 프로메테우스는 wsl 위에서 컨테이너로서 동작
* 프로메테우스의 scrape 타겟 ip(go app)를 docker0 브리지 게이트웨이의 ip로 설정
* Go app은 8080 포트를 열고 있기 때문에 이를 통해 접근

prometheus.yml 파일을 사용하여 컨테이너를 실행

Promethues 컨테이너는 yml 파일의 scrape\_configs에 작성된대로 go app의 /metrics url로 15초 간격으로 api call을 보냅니다.

go app는 자신의 메트릭을 프로메테우스 컨테이너로 전송하게 됩니다.

아래는 Go app의 라우터 코드입니다.

프로메테우스 처리를 위해

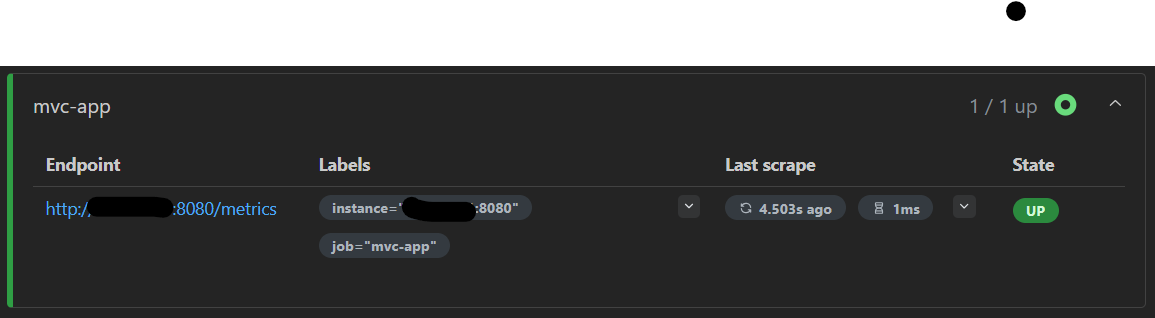
"github.com/prometheus/client\_golang/prometheus/promhttp" 패키지를 import.

프로메테우스 컨테이너로부터 /metrics로 콜이 들어오면

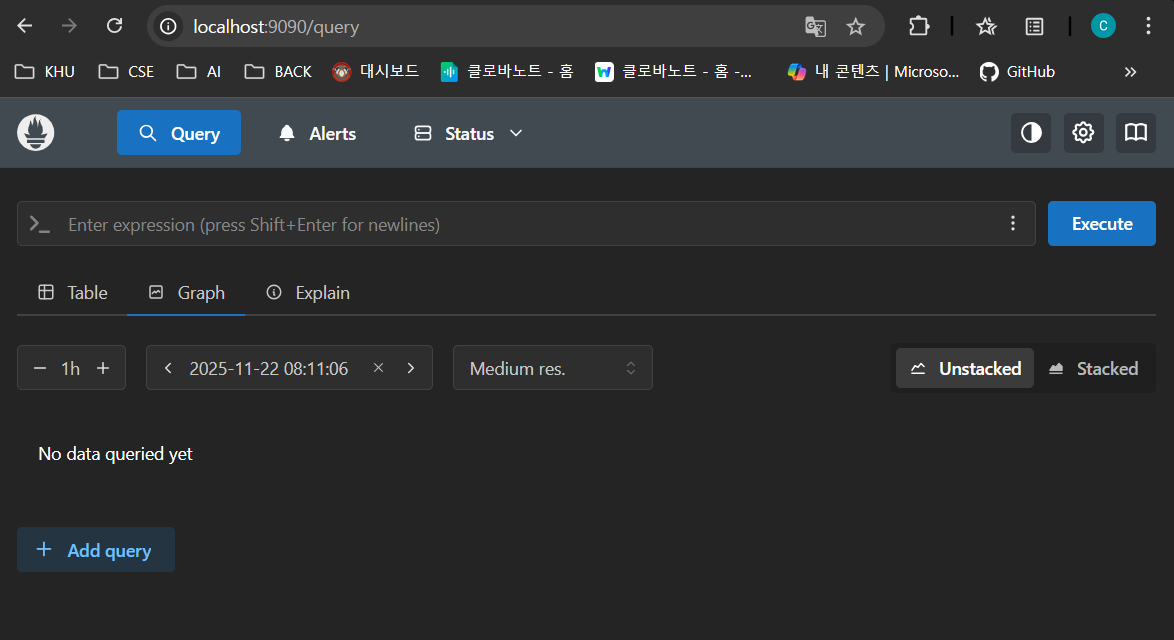
promhttp.Handler()로 처리하도록 go app에 라우트 코드를 한줄 추가하였습니다.

Prometheus 컨테이너는 9090포트를 열고있고 포트포워딩 상태이기 때문에 [http://localhost:9090/targets](http://localhost:9090/targets 에서)를 통해 접근할 수 있습니다.

접속하니 아래와 같이 go app이 동작 중(UP)임을 확인할 수 있었습니다.



웹을 열고 쿼리를 보내서 go app의 현 상태를 확인해 볼 수 있습니다.



이제 쿼리를 실행해 CPU, 메모리 사용률, 고 루틴 수를 확인해보겠습니다.

아래 명령어로 부하를 생성합니다.



-z 30m : 부하 시간 30분

-c 1000 : 연결 개수 1000개

**1. CPU 사용률**

쿼리: rate(process\_cpu\_seconds\_total[2m]) \* 100



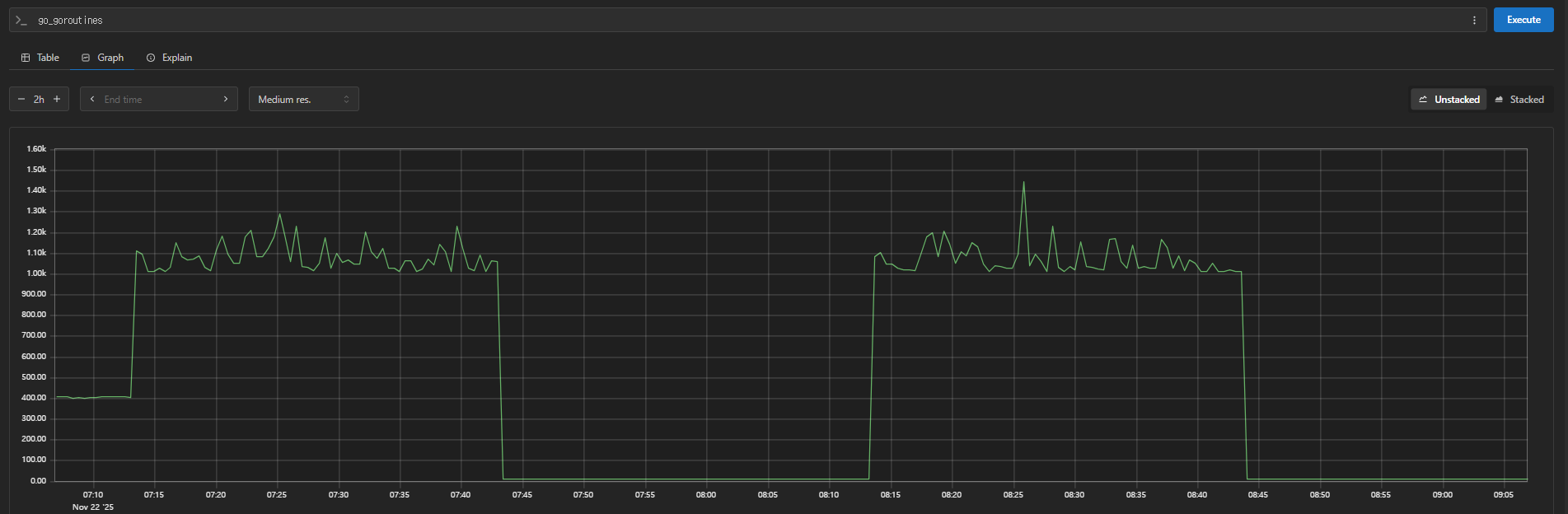
1. **메모리 사용량(MB)**

쿼리:process\_resident\_memory\_bytes / 1024 / 1024



1. **Goroutine 수**

쿼리: go\_goroutines



**두 번의 30분 부하 테스트**를 진행했기 때문에 2개의 골짜기가 생겼습니다.

CPU 사용량과 메모리 사용량, Goroutine 수는 부하테스트를 진행한 30분만큼

증가했다가 테스트가 끝나니 0으로 수렴한 모습을 확인할 수 있습니다.

**느낀 점:** 이번 학기에 신호와 시스템을 수강하고 있는데 그래프가 마치 오실로스코프 그래프 같아서 신기했습니다. 프로메테우스로 도출된 그래프를 푸리에 변환하면 GC 실행이나 부하 테스트가 실행됐다는 사실을 스펙트럼의 형태로 관찰 가능할 거 같습니다. 실제 프로덕션 환경의 서버도 너무 상호작용하는 요소가 많지 않다면 특정 주파수만 유난히 값이 튀는 형태로 결과가 나타나서 서버에 내부적으로 무슨 일이 발생했던 것인지 관찰해볼 수 있는 재밌는 실험이 될 것 같습니다. 서버 분야로 취업하게 된다면 실제 서버 사용량 그래프에 푸리에 변환을 적용해 분석해보고 싶다는 생각이 들었습니다.

**13주차 활동**

**최종 감상문**

한 학기 동안 리눅스로 살면서 가장 큰 적은 예상치 못하게 리눅스가 아니라 라즈베리파이 팬 소리였습니다. 최대한 실제 서버 돌리는 느낌이 어떨지 느껴보고 싶어서 제 방에서 24/7 라즈베리파이를 계속 켜두고 있었습니다. 팬 소리에 어느 정도 익숙해졌다고 생각했는데, 가끔 새벽 시간에 갑자기 시끄러워지는 소리에 잠에서 깨곤 했습니다. 뭐가 문제인지 몰라 케이스를 분해했다가 다시 조립하고, 팬이 잠잠해지면 왜 고쳐졌는지도 모른 채 다시 잠든 게 기억에 남습니다. 분명 제 잠을 방해했지만, 전원이 들어올 때 켜지는 작은 불빛과 백색소음은 이상하게도 저에게 위로가 되었습니다. 어두운 방 속에서 계속 돌아가는 팬 소리가 “나는 멈추지 않고 계속 돌고 있다”라고 말하는 것 같아, 공부와 여러 고민으로 지칠 때 힘이 되어 주었습니다.

이런 환경 속에서 제가 관심이 있던 다양한 서버 프로그램들을 라즈베리파이에 설치해보는 과정은 마치 저만의 아지트에서 노는 것 같았습니다. 문제가 생겨도 “안 되면 밀고 다시 깔지 뭐”라는 마음으로 평소 궁금했던 프로그램들을 마음껏 설치하고 설정도 바꿔보며 실험했습니다. 이론 위주의 과목이 많았던 학기라서, 라즈베리파이는 일종의 일탈 장소 같은 역할을 해주었습니다. 머리로만 배운 내용을 실제로 만져보고 실패해서 다시 밀고 도전하는 일련의 과정을 이렇게 몰입해서 하게 될 줄 몰랐습니다.

다양한 서버 프로그램들의 공식 문서를 살펴보면서 최신 보안 취약점 업데이트와 해결된 버그 내역을 한 줄 한 줄 읽다 보니, 소프트웨어가 단순히 코드 덩어리가 아니라 사람들이 끊임없이 손을 대며 진화시키는 생명체처럼 느껴졌습니다. 문서 속에서 제공하는 기능과 확장성들을 보면, 그 안에 담긴 기술들이 다양하고 신기해서 마법 같다는 생각도 들었습니다.

다만 한 학기 내내 아쉬움으로 남은 건 네트워크 지식이었습니다. 여러 주차 보고서에서도 네트워크 배경지식 부족을 아쉽다고 여러 번 언급했을 정도였습니다. 프로그램들이 서로 통신하며 특정 기능을 수행하는 시스템을 직접 구성해보고 싶었기 때문입니다. 그래서 이번 학기를 통해 추후 수강할 컴퓨터 네트워크 과목에 대한 동기가 더 강해졌습니다. 앞으로는 라즈베리파이를 여러 대 묶어 클러스터를 만들고, 랜선으로 연결하고, 각 장비마다 역할을 부여한 뒤 그에 맞는 소프트웨어를 설치해 작은 데이터 센터 같은 분산 시스템을 직접 구축해보고 싶다는 생각이 들었습니다.