

제 1 장

WEB

- 1) Web이란?
- 2) Frontend와 Backend의 이해
- 3) Web동작과 HTTP 이해
- 4) 개발자 도구

1-1. Web이란?

통신이란?

의사소통(Communication)이란 사람 또는 집단 사이에서 정보를 주고 받으며 생각, 감정, 의견 등을 전달하고 이해하는 과정이다. 언어적·비언어적 방식 모두를 포함하며, 말이나 글뿐만 아니라 표정, 몸짓, 시선, 음성 톤 등 다양한 요소가 복합적으로 작용한다. 우리는 일상 속에서 수많은 사람과 의사소통을 하며 그 속에서 서로의 생각과 감정을 공유한다.



가까운 상태와 의사소통 할 때는 말과 행동으로 의사를 효과적으로 전달할 수 있다. 그러나 상대와의 거리가 조금만 멀어지면 말과 행동만으로는 의사소통을 원활히 이루어질 수 없다. 멀리 있는 사람들에게 정보를 전달하고 의사소통을 하기 위해서 통신 기술이 발전하였다.

통신(Telecommunication)이란 공간적으로 떨어져 있는 주체들 간에 정보를 전기적·전자적·기계적인 매체나 기술을 이용해 원격으로 전달하는 기술적 과정이다. 통신은 주로 기술적인 수단을 통해 이루어진다.

통신의 역사는 매우 오래되었다. 과거에는 직접 말을 전하거나, 멀리 떨어진 상대방에게 중요한 소식을 알리기 위해 봉화와 같은 원시적인 신호 수단을 사용하기도 했다. 그렇기에 시간과 거리의 제약을 충분히 해소하지 못했다.

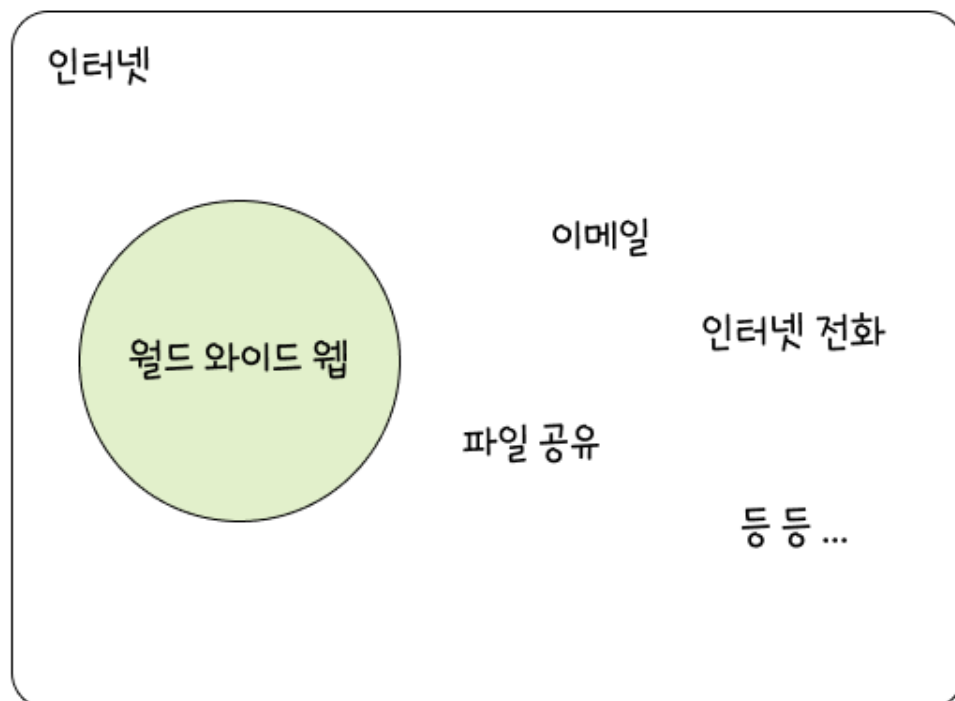


현대에는 기술이 발전하여 전 세계에 걸친 통신망을 바탕으로 다양한 나라의 사람들과 실시간으로 통신할 수 있게 되었다. 화상으로 강의하고, 메신저로 이야기하고, 메일로 소식을 전하는 것들이 이제는 일상의 한 부분으로 자리잡았다. 점차 시간적 공간적 제약이 거의 없어지고 있다.

웹이란?

웹(World Wide Web, W3, Web)은 현대 정보 통신을 대표하는 통신 수단 중 하나이다. 인터넷에 연결된 전세계 사용자들이 서로의 정보를 공유할 수 있는 장소를 의미한다. 거미줄처럼 엮인 공간이라는 뜻의 월드 와이드 웹(World Wide Web)을 줄여서 웹(Web)이라고 부른다.

흔히 인터넷과 웹을 혼동해서 사용하곤 하지만, 이는 엄밀히 다른 것이다. 인터넷(Internet)이란, 전세계 수많은 컴퓨터와 네트워크 장치들이 연결된 거대한 글로벌 통신망이다. 그리고 웹은 그 인터넷 위에서 작동하는 하나의 정보 서비스 체계이다. 인터넷을 도서관으로 비유한다면 웹은 그 도서관 안에 있는 책 한 권으로 비유할 수 있다. 집합 관계로 표현한다면 인터넷 속에 웹이 포함되어 있다.



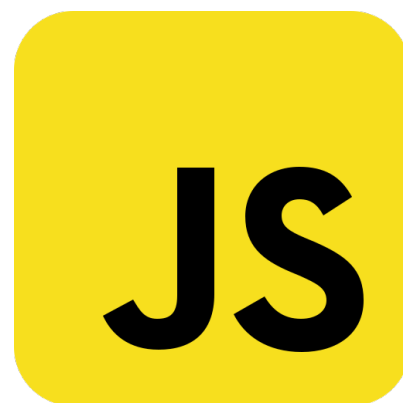
웹의 구성 요소

(1) 웹 페이지(Web Page)

웹에는 수많은 웹 사이트(Web Site)들이 있고, 각각의 웹 사이트는 웹 페이지로 구성되어 있다. 즉, 웹 사이트는 관련된 여러 웹 페이지의 모임 또는 온라인 공간이다.

구분	웹 페이지(Web Page)	웹 사이트(Web Site)
정의	하나의 문서 혹은 화면	여러 웹 페이지의 모임 또는 집합체
구성	HTML 문서 한 개	다수의 웹 페이지와 미디어, 스크립트
URL	고유 URL 하나	도메인 주소 아래 여러 페이지의 집합
규모	단일 문서	복합적이고 방대한 정보 공간
예시	블로그 글 한 편, 상품 정보 페이지	포털 사이트, 기업 홈페이지 등

웹 페이지는 사용자가 웹을 이용할 때 화면에 표시되는 하나의 문서이다. 이 문서에는 HTML, CSS, JS로 구성되며, 웹의 구조, 디자인, 그리고 동작을 각각 담당한다.



- HTML(Hyper Text Markup Language)

웹 페이지의 기본 구조와 내용을 정의한다.

텍스트, 이미지, 링크 등 웹 문서의 뼈대를 구성하며, 태그와 속성을 사용해 문서를 체계적으로 작성할 수 있다.

- CSS(Cascading Style Sheets)

HTML로 작성된 구조에 스타일을 입힌다.

색상, 글꼴, 배치, 크기 등 시각적 요소를 조절하여 웹 페이지를 아름답고 보기 좋게 만든다.

- JS(JavaScript)

웹 페이지에 동적인 기능과 상호작용을 추가한다.

사용자의 입력 처리, 화면의 실시간 변화, 애니메이션, 데이터 통신 등 다양한 동작을 가능하게 한다.

웹은 인터넷 상에서 텍스트나 그림, 소리 영상 등과 같은 멀티미디어 정보를 하이퍼텍스트(Hyper-Text) 방식으로 연결하여 제공한다. 하이퍼텍스트란 문서 내부에 다른 문서로 연결되는 참조 링크를 포함하는 기술을 의미한다. 이는 웹 페이지들이 서로 유기적으로 연결될 수 있도록 해주며, 앞서 설명한 HTML, CSS, JavaScript의 조합으로 구현된다.

결국 웹은 이렇게 수많은 웹 페이지들이 하이퍼링크(Hyper Link)를 통해 서로 연결되어 만들어진 거대한 정보망이다.

(2) 웹 브라우저(Web Browser)

웹 브라우저(Web Browser)란 웹을 컴퓨터로 불러와 실행하는 응용 프로그램이다. 사용자가 인터넷이나 웹 서버에 저장된 웹 페이지와 콘텐츠를 쉽게 보고, 탐색하며 상호작용할 수 있도록 돕는 소프트웨어이다. 쉽게 비유하자면, 웹 브라우저는 인터넷이라는 거대한 도서관에서 원하는 책을 찾아 읽을 수 있게 도와주는 책갈피이나 안내자 역할을 한다.

대표적인 웹 브라우저로는 구글 크롬 (Google Chrome), 모질라 파이어폭스 (Mozilla Firefox), 애플 사파리 (Apple Safari), 마이크로소프트 엣지 (Microsoft Edge), 네이버 웨일 (Naver Whale) 등이 있다.



웹 브라우저의 주요 역할로는 다음과 같다.

- 웹 페이지 요청 및 수신

사용자가 주소(URL)를 입력하거나 링크를 클릭하면, 웹 브라우저는 해당 웹 서버에 웹 페이지를 요청한다. 서버에서 받은 데이터를 수신한다.

- 웹 페이지 해석 및 렌더링

HTML, CSS, JS로 구성된 웹 페이지 데이터를 해석하여 텍스트, 이미지, 영상 등 시각적 요소로 화면에 표시한다.

- 사용자 인터페이스 제공

주소창, 뒤로가기, 앞으로가기, 탭, 즐겨찾기, 북마크 등 사용자가 웹을 편리하게 탐색할 수 있는 다양한 기능을 제공한다.

- 상호작용 처리

사용자의 클릭, 입력, 스크롤 등의 행동에 반응하여 웹 페이지 내에서 동적인 기능과 실시간 변화를 가능하게 한다.

- 보안 기능

HTTPS 연결을 지원하여 데이터를 암호화하고, 악성 사이트 차단, 개인정보 보호 등 사용자의 보안을 지킨다.

- 확장기능 지원

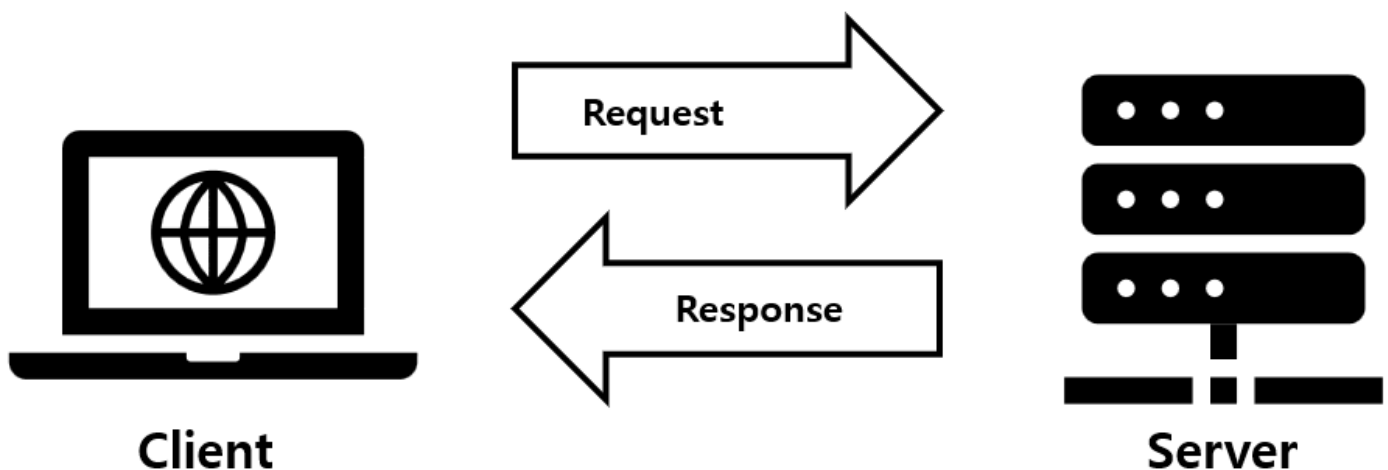
플러그인이나 확장 프로그램을 설치해 기능을 추가하거나 맞춤형 환경을 만들 수 있다.

이러한 기능들로 사용자는 웹 브라우저를 통해 웹 서핑을 즐길 수 있다. 웹 브라우저의 개발자 도구는 1-4장에서 자세히 다룬다

(3) 서버와 클라이언트(Server & Client)

서버란 네트워크를 통해 다른 컴퓨터나 클라이언트의 요청을 받아 처리하고, 그에 대한 응답을 제공하는 컴퓨터 또는 소프트웨어를 말한다. 일반적으로 사용자는 웹 브라우저를 통해 서버로부터 정보를 받아 화면에서 확인할 수 있다. 웹 서버의 대표적인 종류로는 아파치(Apache), Nginx 등이 있다.

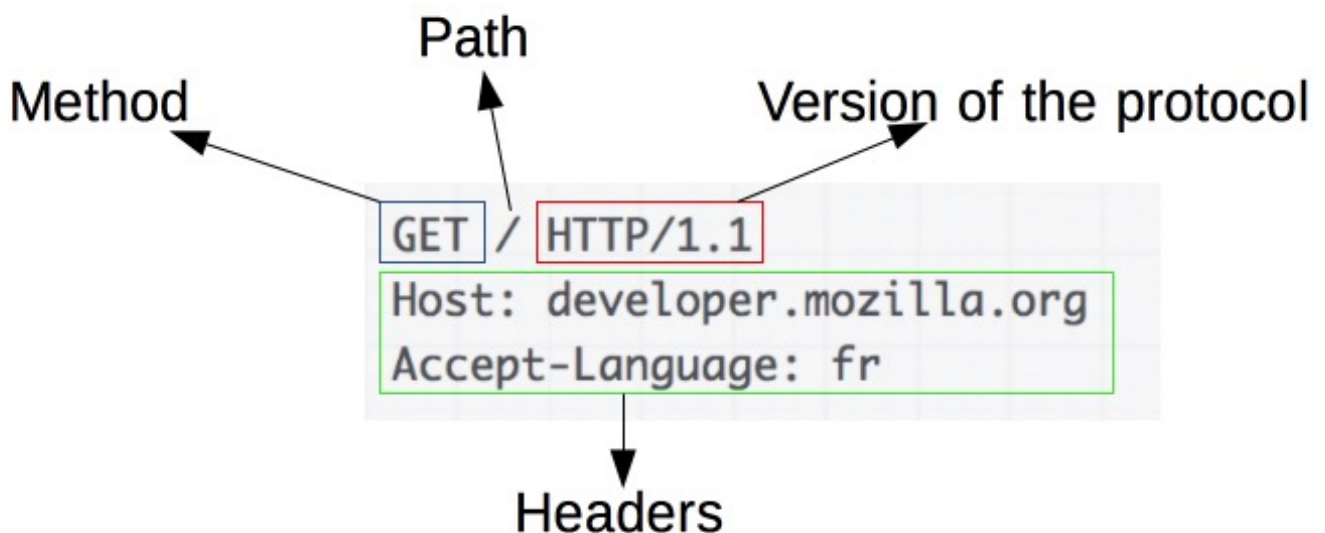
클라이언트란 서버에서 제공하는 서비스를 사용하는 주체로, 크게 물리적 장치와 소프트웨어 프로그램으로 나눌 수 있다. 클라이언트 장치는 최종 사용자가 웹에 접속하기 위해 사용하는 시스템으로 데스크톱, 노트북, 스마트폰, 태블릿 등이 이에 해당한다. 클라이언트 프로그램은 사용자가 웹을 통해 요청할 수 있게 해주는 프로그램으로 웹 브라우저가 그 예시이다.



(4) HTTP/HTTPS

프로토콜이란 컴퓨터와 네트워크 장치들이 서로 정보를 주고 받을 때 지켜야 하는 규칙이나 약속을 말한다. 웹에서는 주로 HTTP(HyperText Transfer Protocol)와 HTTPS(Hyper Text Transfer Protocol Secure)을 주로 사용한다.

HTTP는 서버/클라이언트 모델을 따라 데이터를 주고 받기 위한 프로토콜이다. 인터넷에서 하이퍼텍스트를 교환하기 위한 통신규약으로, 80번 포트를 사용하고 있다. 따라서 HTTP 서버가 80번 포트에서 요청을 기다리고 있으며, 클라이언트는 80번 포트로 요청을 보내게 된다.



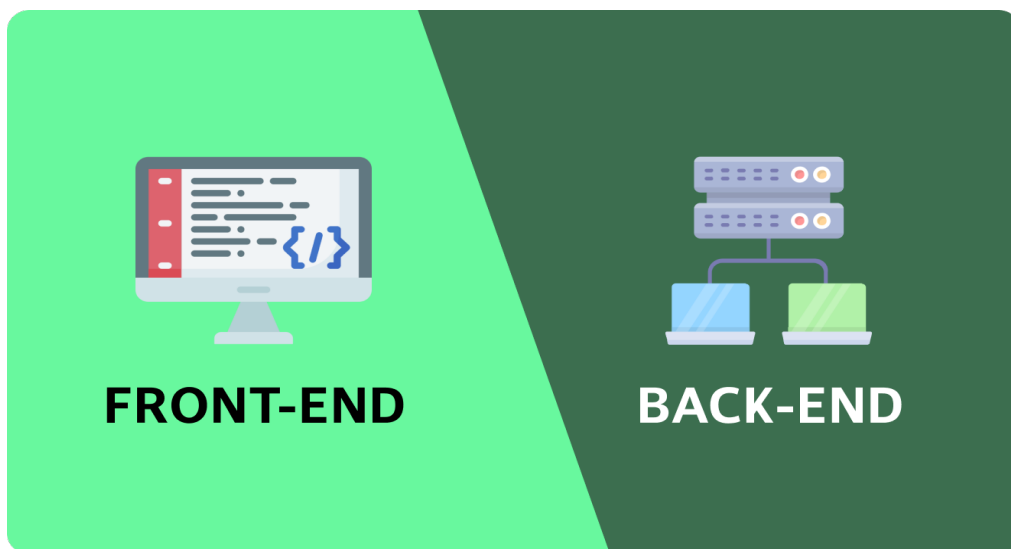
HTTPS는 HTTP에 데이터 암호화가 추가된 프로토콜이다. HTTP와 다르게 443번 포트를 사용하며, 네트워크 상에서 중간에 제3자가 정보를 볼 수 없도록 데이터를 암호화하여 안전한 통신을 지원한다

HTTP에 대한 내용은 1-3장에서 더 자세히 배운다.

1-2. Frontend와 Backend의 이해

웹 개발의 두 가지 핵심 요소는 바로 프론트엔드(Front-end)와 백엔드(Back-end)이다. 각각 독립적인 역할과 책임을 가지고 있어 웹 개발에서 매우 중요한 위치를 차지한다.

프론트엔드는 사용자가 웹 페이지를 열었을 때 보게 되는 화면을 만드는 작업이다. 반면, 백엔드는 사용자가 웹 페이지를 이용하는 동안 필요한 비즈니스 로직과 기능을 구현하는 작업이다.



사용자는 웹 페이지를 처음 접속했을 때 보이는 화면만을 인지하기 때문에 프론트엔드와 백엔드가 분리되어 있다는 사실을 인지하지 못할 수 있다. 하지만 두 영역은 서로 독립적이며, 각각 고유한 역할과 책임을 수행한다.

따라서 프론트엔드/백엔드의 차이를 명확히 이해하고, 각자의 역할과 관련 기술을 숙지하는 것은 웹 개발 프로젝트를 보다 깊이 있게 이해하는 데 큰 도움이 된다.

프론트엔드

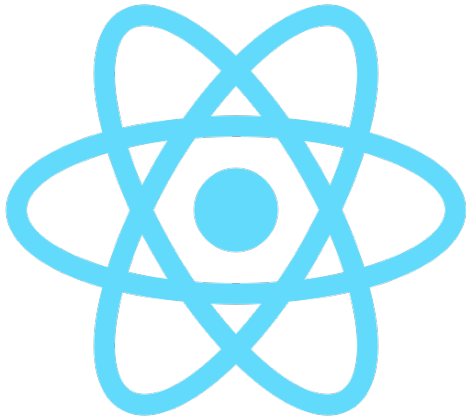
프론트엔드는 사용자가 웹 페이지를 방문했을 때 직접 보고 상호작용하는 웹의 시각적·인터랙티브 부분이다. 사용자에게 정보를 전달하고, 사용자 입력을 받아 처리하며, 브라우저에서 볼 수 있는 웹 사이트의 레이아웃, 디자인, 버튼, 이미지, 텍스트, 애니메이션 등 모든 화면 요소를 포함하여 사용자 인터페이스(UI)를 개발하는 영역이다.

프론트엔드는 사용자 경험(UX)을 개선하고 웹 페이지의 시각적인 부분과 동작을 구현하여 웹 사이트가 효과적으로 작동하고 사용자가 쉽게 이용할 수 있도록 한다. 프론트엔드 개발에는 HTML, CSS, JavaScript 기반으로 하며, React, Vue, Angular 같은 현대적 프레임워크와 라이브러리를 활용한다.

백엔드

백엔드는 사용자가 직접 볼 수 없는, 웹 사이트나 웹 서비스의 서버 측에서 작동하는 부분이다. 웹 사이트나 애플리케이션이 원활하게 동작하도록 데이터를 처리하고 저장하며, 비즈니스 로직을 수행하는 역할을 담당한다. 사용자 인증, 데이터베이스 관리, 서버 로직 관리, API 제공 등 웹 서비스의 핵심 기능을 수행한다. 백엔드 개발에는 Spring, Django, Express 같은 프레임 워크를 사용하며 데이터베이스로는 MySQL, PostgreSQL, MongoDB 등과의 연동이 중요하다.

프론트엔드 프레임워크



백엔드 프레임워크



1-3. Web통신과 HTTP 이해

URI (Uniform Resource Identifier)

URI는 인터넷 상에서 자원을 식별하는 포괄적인 식별자 개념이다. 인터넷 서비스를 전제로한 인터넷상의 통일된 정보 자원의 식별 체계이다. 문서, 이미지, 서비스 등 모든 웹 자원을 구분하는 고유한 문자열로, 크게 URL과 URN으로 나뉜다.

URL(Uniform Resource Locator)

URL은 URI의 한 종류로, 자원의 위치(주소)와 접근방법(프로토콜)을 나타낸다. 즉, 인터넷에서 자원을 찾기 위한 주소 역할을 한다. 일반적으로 사이트 도메인을 포함하며, 웹뿐만 아니라 컴퓨터 네트워크 상의 자원을 모두 나타낼 수 있다.

URN(Uniform Resource Name)

URN은 URI의 또 다른 종류로, 자원의 이름(식별자)만을 나타내며 위치 정보는 포함하지 않는다. 주로 자원의 영구적인 식별을 위해 사용하며, 자원의 위치와는 무관하게 이름만으로 자원을 식별하는 방식이다.

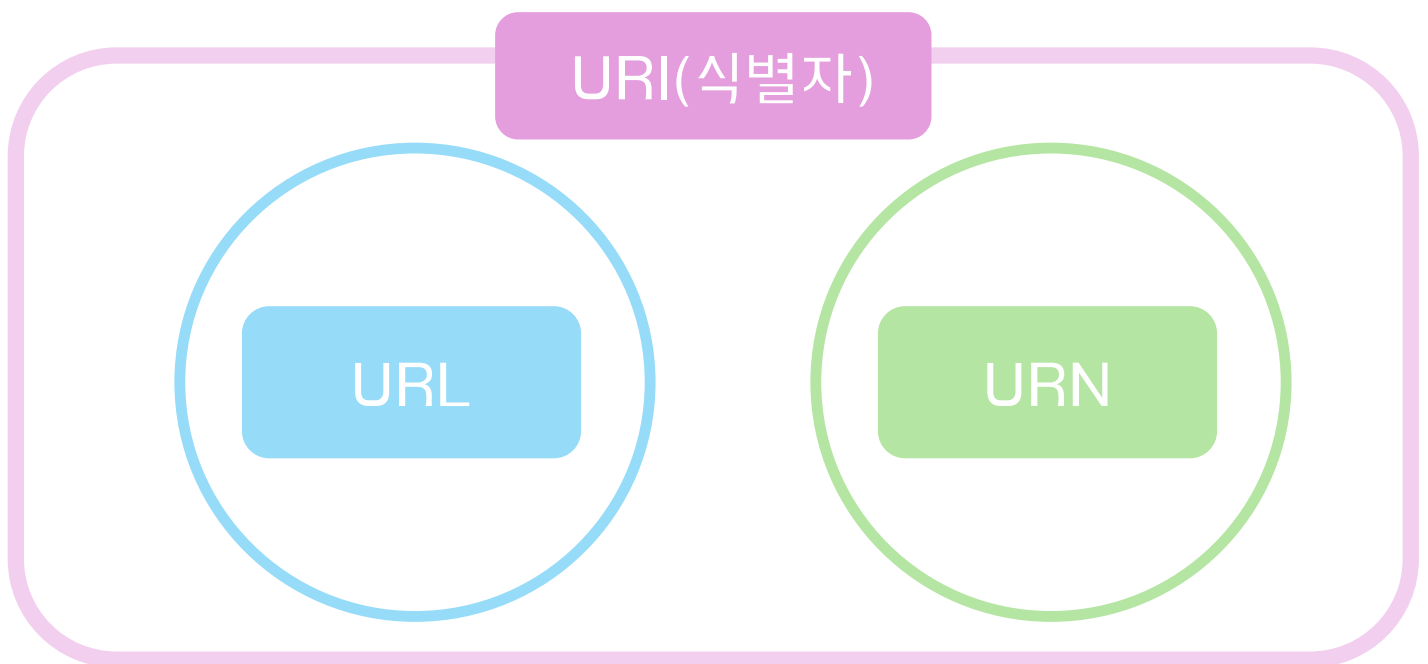
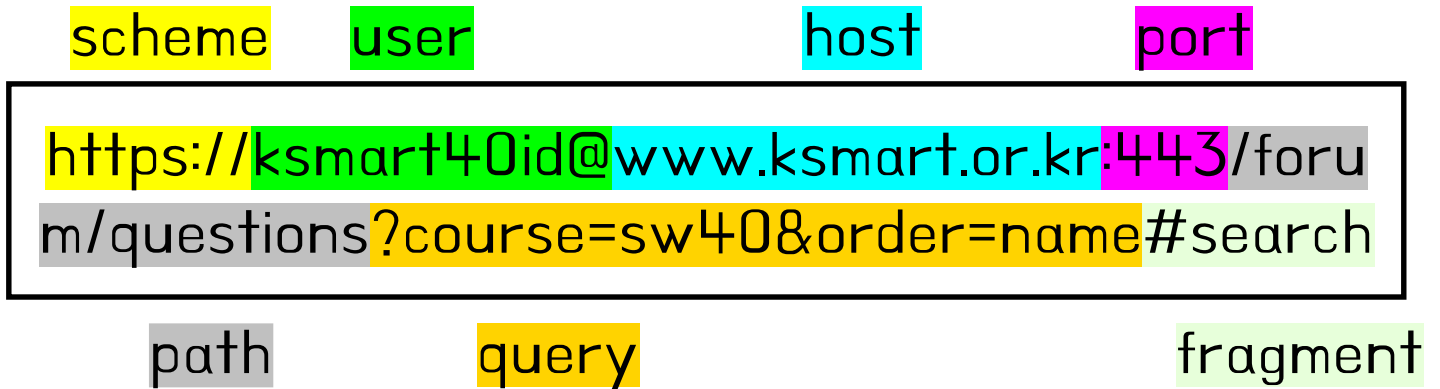
URI의 구조

```
scheme:[//[user[:password]@]host[:port]]
[/path][?query][#fragment]
```

URI의 구조는 다음과 같다. 사용하지 않는 일부 부분은 생략할 수 있다.

요소	설명
scheme	웹 서버와 어떤 프로토콜로 통신할지 표시
user, password	서버에 접근하기 위한 사용자 정보 사용자의 이름, 비밀번호
host	도메인 혹은 IP, 접속하고 싶은 컴퓨터 접속할 웹 서버의 주소에 대한 정보
port	접속할 웹 서버의 포트에 대한 정보
path	서버에 제공하는 자원의 경로 접근할 웹 서버의 리소스 경로로 '/'로 구분
query	웹 서버에 전달하는 파라미터 ? 뒤에 위치하고 key=value형식으로 나열, &로 구분
fragment	문서 내 특정 위치(책갈피 등)을 가르킴 # 뒤에 식별자를 붙여 사용

아래는 URI 주소의 예시이다.



이 전체 URI는 인터넷 상의 자원을 식별하는 포괄적 개념이며, URL은 자원의 위치와 접근 방법을 나타내는 URI의 한 종류이고, URN은 자원의 이름만으로 식별하는 URI의 한 종류이다. 위 URI에서 프로토콜부터 path 까지의 범위를 URL이라고 지정할 수 있다.

DNS(Domain Name System)

URI의 host부분은 도메인 이름(Domain name) 또는 IP주소(IP address)의 값을 가질 수 있다. IP 주소는 네트워크 상에서 장치를 식별하기 위한 숫자 주소이지만, 불규칙한 숫자 조합이라 사람이 기억하기 어렵다. 따라서 사람이 이해하고 기억하기 쉬운 도메인 이름을 사용하며, DNS는 이 도메인 이름을 대응하는 IP 주소로 변환해주는 시스템이다. Domain name을 Host값으로 이용할 때, 브라우저는 Domain Name Server에 도메인 이름을 질의하여 IP 주소를 받아오고, 이를 이용해 통신을 수행한다.

웹 통신 과정

1. 사용자 요청(Request)

사용자가 웹 브라우저에 URL을 입력하거나 웹 페이지 내 링크를 클릭한다. 브라우저는 이 URL에 대한 정보를 요청하는 HTTP 요청 메시지를 생성해 해당 서버로 보낸다.

2. DNS 조회

브라우저는 URL의 도메인 이름을 DNS 서버에 질의하여 대응하는 IP주소를 얻는다. 이 IP주소를 통하여 서버와 통신한다.

3. 서버 연결 및 요청 전송

브라우저는 얻은 IP주소를 사용해 서버와 연결을 맺고, HTTP 요청 메시지를 전송한다. 이 요청에는 사용자가 원하는 자원(웹 페이지, 이미지) 등에 대한 정보가 포함된다.

4. 서버 처리

서버는 클라이언트의 요청을 받아 처리한다. 필요한 데이터를 데이터베이스에서 조회하거나 웹 페이지를 생성하는 등의 작업을 수행한다.

5. 서버 응답(Response)

서버는 요청 처리 결과를 HTTP 응답 메시지로 만들어 클라이언트에 보낸다. 응답에는 요청한 웹 페이지의 HTML, 이미지, CSS, JS등이 포함된다.

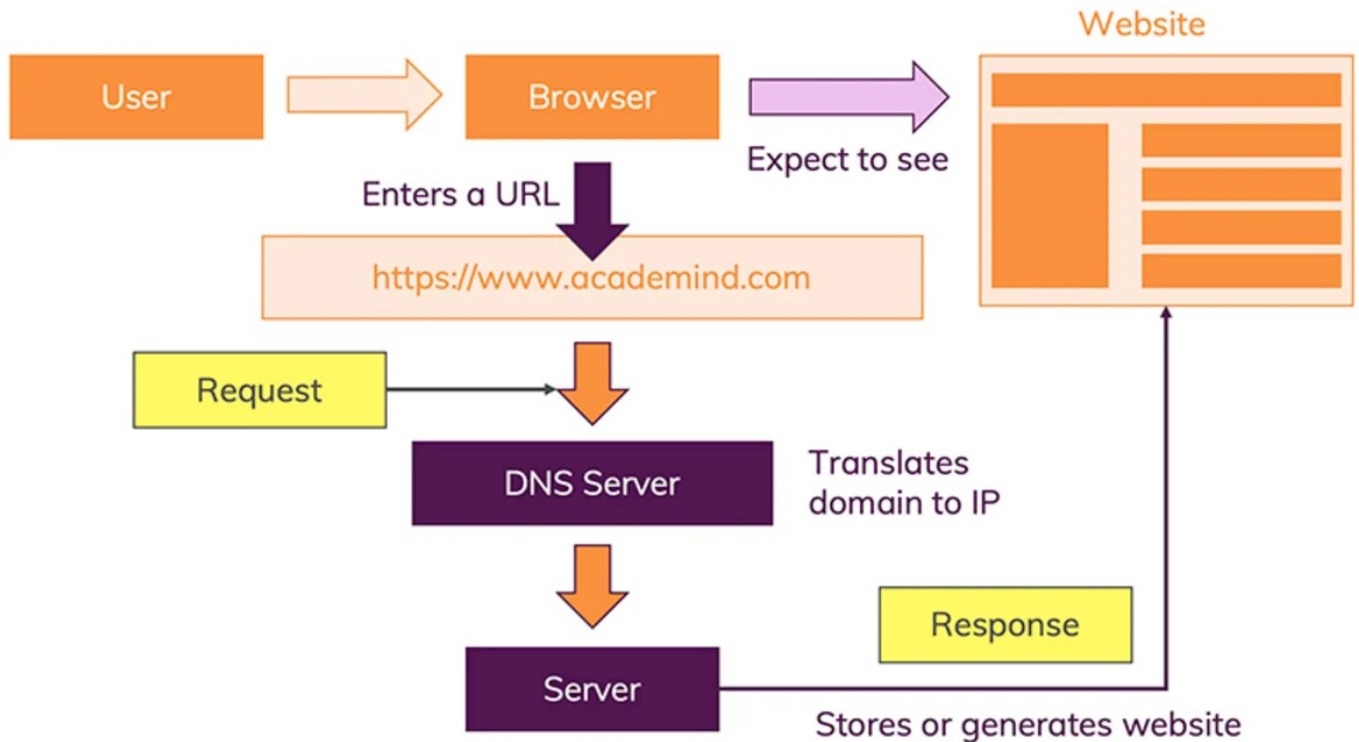
6. 브라우저 렌더링 및 표시

브라우저는 받은 데이터를 해석해 화면에 웹 페이지를 표시한다. HTML 구조에 CSS 스타일을 적용하고, JS를 실행해 동적인 기능을 구현한다.

7. 사용자 상호작용 및 반복

사용자는 페이지 내에서 버튼 클릭, 폼 입력 등 다양한 행동을 할 수 있다. 이러한 행동은 다시 서버에 새로운 요청을 보내는 과정으로, 1~6번 과정을 반복한다.

사용자 → 브라우저 → DNS 조회 → 서버 요청 → 서버 처리 → 응답 → 브라우저 렌더링 → 사용자 상호작용으로 이어지는 요청-응답 사이클이다.



웹의 데이터 교환

앞서 웹 통신 과정에서 살펴본 것처럼, 웹 브라우저(프론트엔드)는 서버(백엔드)에 요청을 보내고 서버는 이를 처리한 후 응답을 보낸다. 이 과정에서 서로 약속된 규칙과 형식을 사용해 소통하는데, 이 규칙이 바로 HTTP 프로토콜이다.

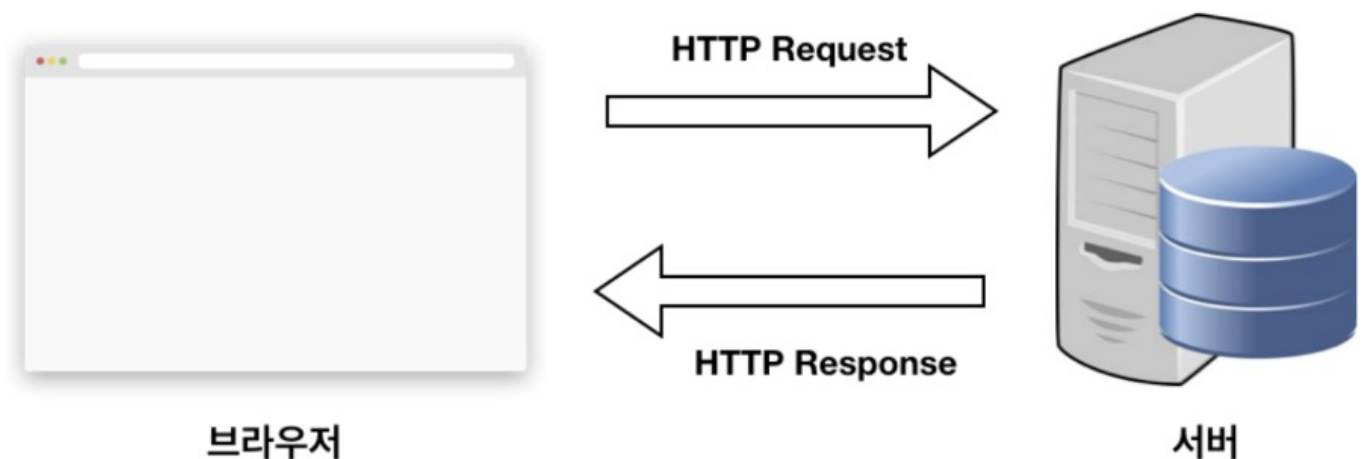
HTTP 통신에는 데이터가 주로 HTTP 메시지의 본문(바디)에 JSON 형식으로 전달되며, 필요한 추가 정보는 HTTP 헤더에 담겨 전송된다.

HTTP(Hyper Text Transfer Protocol)

HTTP란 웹(WWW) 상에서 데이터를 주고 받기 위한 약속 과 규칙, 즉 통신 프로토콜이다. 클라이언트(보통 웹 브라우저)와 서버 간에 요청(request)과 응답(response)을 통해 작동하며 HTML, CSS, JS, png, gif, mp4 등 다양한 종류의 데이터들을 전송할 수 있도록 설계되었다.

HTTP는 TCP/IP 통신 프로토콜을 기반으로 동작하며, 기본적으로 80번 포트를 사용한다. 단방향성 방식으로, 하나의 요청에 하나의 응답을 처리하며, 각 요청은 독립적으로 관리되어, 이전 요청과 다음 요청 간에 상태를 유지하지 않는 무상태(stateless) 프로토콜이다.

다만, HTTP는 데이터를 평문(사람이 알아볼 수 있는 정보)으로 전송하기 때문에 도청 및 데이터 변조 위험이 존재한다.



HTTP 요청 메서드 종류

HTTP 요청 메서드(HTTP Request Method)는 클라이언트가 서버에 어떤 작업을 요청하는지를 나타내는 명령어이다.

GET

GET 메서드는 주로 데이터를 조회(Read)하거나 검색(Retrieve)할 때 사용되는 메서드이다. 주로 웹 페이지나 리소스를 가져올 때 쓰이며, 데이터를 읽기 전용으로 읽을 때만 사용하기에 서버 상태를 변경하지 않는다.(멩등성)

POST

POST 메서드는 주로 새로운 리소스를 생성(Create)하거나 제출할 때 사용한다. 서버의 상태를 변경할 수 있으며, 동일한 POST 요청을 여러 번 보내더라도 항상 같은 결과물 보장하지 않는다. 즉, 멩등성이 없고, 같은 데이터를 여러 번 보내면 여러 개의 리소스가 생성될 수 있다.

PUT

PUT 메서드는 리소스를 생성하거나 업데이트하기 위해 서버에 데이터를 보낼 때 사용된다. 동일한 PUT 요청을 보내도 결과는 항상 동일하며, 멩등성을 가진다.

****멩등성 :** 연산을 여러 번 적용하더라도 결과가 달라지지 않는 성질

DELETE

DELETE 메서드는 지정된 리소스를 삭제하는 데 사용한다. 일반적으로 삭제에 필요한 요청 본문(body)은 포함하지 않는다.

PATCH

PATCH 메서드는 리소스의 일부만 수정할 때 사용한다.

HEAD

HEAD 메서드는 GET과 동일하지만 응답 본문(body)은 제외하고 헤더만 받는다. 주로 리소스 존재 여부나 상태 확인에 사용한다.

OPTIONS

OPTIONS 메서드는 서버가 지원하는 메서드나 기능을 확인할 때 사용한다.

PATCH vs PUT

PATCH와 PUT 메서드 모두 수정할 때 사용하는 HTTP 요청 메서드이다. 그러나 둘은 큰 차이점이 있다. PUT은 리소스의 전체를 교체(replace)할 때 사용하며, 클라이언트가 보낸 데이터로 해당 리소스 전체를 덮어쓴다. 따라서 멍등성을 가지며, 같은 PUT 요청을 여러 번 보내도 결과가 같다. 반면 PATCH는 리소스의 일부만 수정(update)할 때 사용되며, 멍등성이 있을 수도 있고 없을 수도 있다. 요약하면 PUT은 전체 리소스 교체하고, PATCH는 부분 리소스 수정한다.

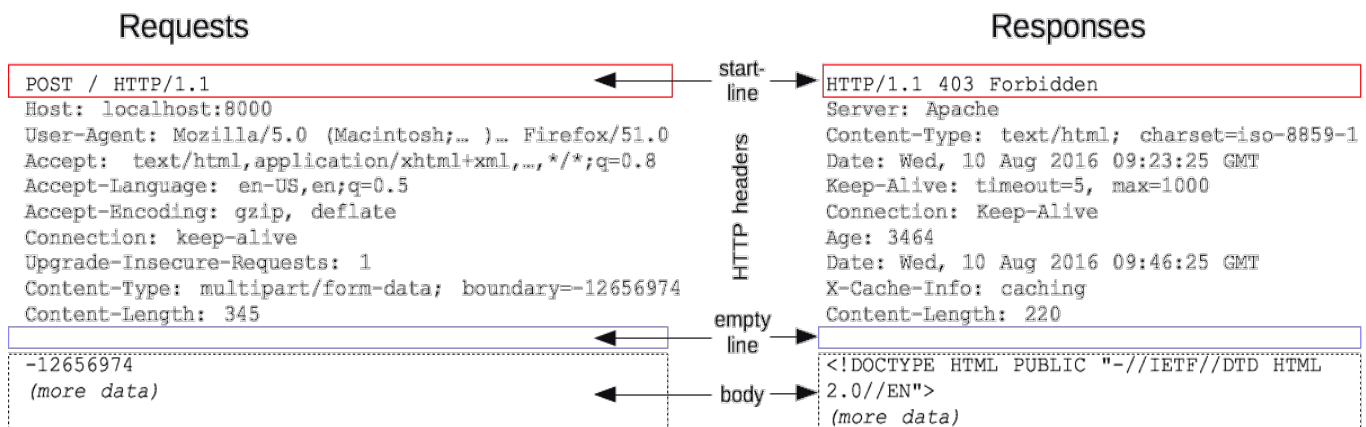
HTTP 상태 코드

HTTP 상태코드는 서버가 클라이언트의 요청을 처리한 결과를 숫자 코드로 나타낸 것이다. 이 코드를 통해 클라이언트는 요청이 성공했는지, 오류가 발생했는지, 추가 작업이 필요한지 등을 알 수 있다.

응답대역	응답코드	설명
정보전송 임시응답	100	Continue (클라이언트로 부터 일부 요청을 받았으며 나머지 정보를 계속 요청함)
	101	Switching protocols
성공	200	OK(요청이 성공적으로 수행되었음)
	201	Created (PUT 메소드에 의해 원격지 서버에 파일 생성됨)
	202	Accepted(웹 서버가 명령 수신함)
	203	Non-authoritative information (서버가 클라이언트 요구 중 일부만 전송)
	204	No content, (PUT, POST, DELETE 요청의 경우 성공은 했지만 전송할 데이터가 없는 경우)
리다이렉션	301	Moved permanently (요구한 데이터를 변경된 타 URL에 요청함 / Redirect된 경우)
	302	Not temporarily
	304	Not modified (컴퓨터 로컬의 캐시 정보를 이용함, 대개 gif 등은 웹 서버에 요청하지 않음)
클라이언트 요청에러	400	Bad Request (사용자의 잘못된 요청을 처리할 수 없음)
	401	Unauthorized (인증이 필요한 페이지를 요청한 경우)
	402	Payment required(예약됨)
	403	Forbidden (접근 금지, 디렉터리 리스팅 요청 및 관리자 페이지 접근 등을 차단)
	404	Not found, (요청한 페이지 없음)
	405	Method not allowed (허용되지 않는 http method 사용함)
	407	Proxy authentication required (프락시 인증 요구됨)
	408	Request timeout (요청 시간 초과)
	410	Gone (영구적으로 사용 금지)
	412	Precondition failed (전체 조건 실패)
	414	Request-URI too long (요청 URL 길이가 긴 경우임)
서버에러	500	Internal server error (내부 서버 오류)
	501	Not implemented (웹 서버가 처리할 수 없음)
	503	Service unavailable (서비스 제공 불가)
	504	Gateway timeout (게이트웨이 시간 초과)
	505	HTTP version not supported (해당 http 버전 지원되지 않음)

HTTP 메시지

HTTP 메시지는 클라이언트가 서버에 보내는 요청(request)와 서버가 클라이언트에 보내는 응답(response) 두 종류가 있다. 기능과 세부 구조에서는 차이가 있지만 크게 보면 이들은 HTTP 헤더와 바디로 구성된다는 공통점이 있다.



메시지는 시작 줄(Start Line)으로 시작하며, 그 뒤를 헤더들(header)이 따라온다. 헤더가 끝나면 빈 줄(Empty Line)이 나오고, 그 다음 본문(body)이 이어진다. 즉, 헤더와 본문은 이 빈 줄을 기준으로 구분된다.

HTTP 헤더

HTTP 헤더(Headers) 각 줄은 CRLF로 구분되며, 첫 줄은 시작 줄(Start Line), 나머지 줄은 헤더(Header)라고 불린다. 헤더의 끝은 빈 줄로 표시가 된다.

이때 CRLF란 커서를 맨 앞으로 이동 시키는 문자인 Carriage Return(CR) 과 커서를 다음 줄로 이동시키는 문자인 Line Feed(LF)의 조합이다. 주로 텍스트 파일에서 줄 바꿈을 나타내는 데 사용되는 제어 문자열이다.

HTTP 바디

HTTP 바디(body)는 헤드의 끝을 나타내는 CRLF 뒤 모든 줄을 말한다. 실제 전송할 데이터이며, HTML 문서, 이미지, 영상, **JSON** 등등 byte로 표현할 수 있는 모든 데이터가 전송 가능하다.

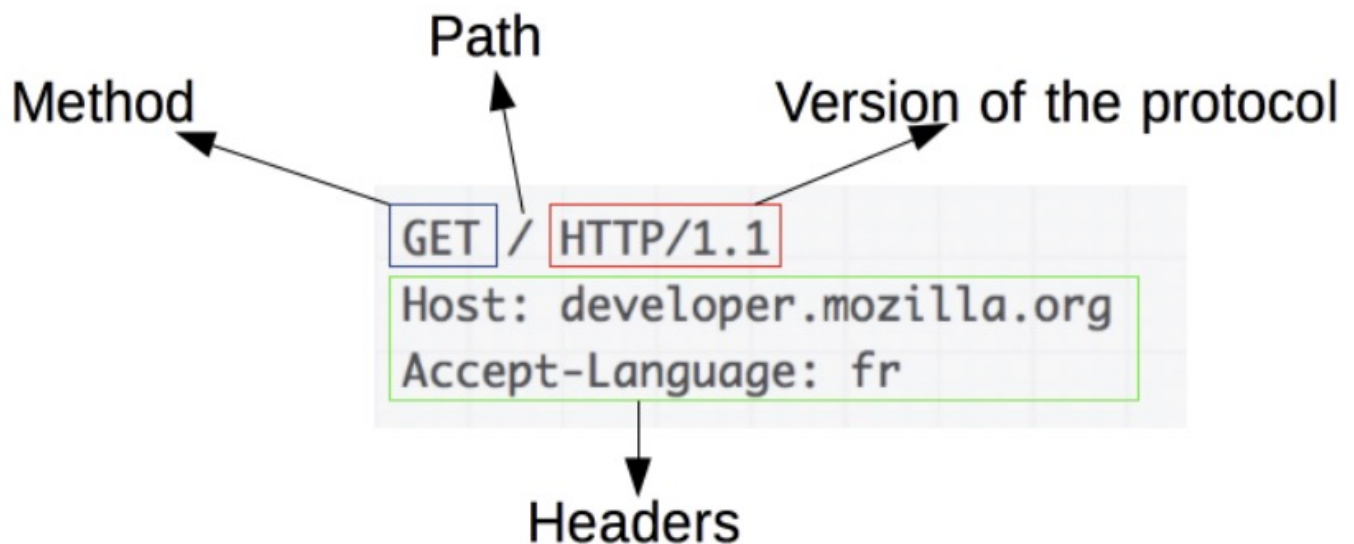
JSON(JavaScript Object Notation)

JSON은 데이터를 표현하고 교환하기 위한 경량의 텍스트 기반 포맷이다. 사람이 읽고 쓰기 쉽고, 기계가 파싱하고 생성하기 쉬운 구조를 가지고 있어 웹과 서버 간 데이터 통신에 널리 사용된다. JSON의 주요 특징은 다음과 같다.

- 키-값 쌍으로 이루어진 데이터 구조
- 데이터 타입 지원
- 언어 독립적이면서도 대부분 프로그래밍 언어에서 쉽게 파싱/생성
- 웹 API와 RESTful 서비스에서 표준 데이터 형식으로 채택

```
{
  "name": "Alice",
  "age": 25,
  "hobbies": ["reading", "swimming", "coding"],
  "isStudent": true,
  "address": {
    "city": "Seoul",
    "zipcode": "12345"
  }
}
```

Request 메시지 구조



Method : 클라이언트가 수행하고자 하는 동작을 정의

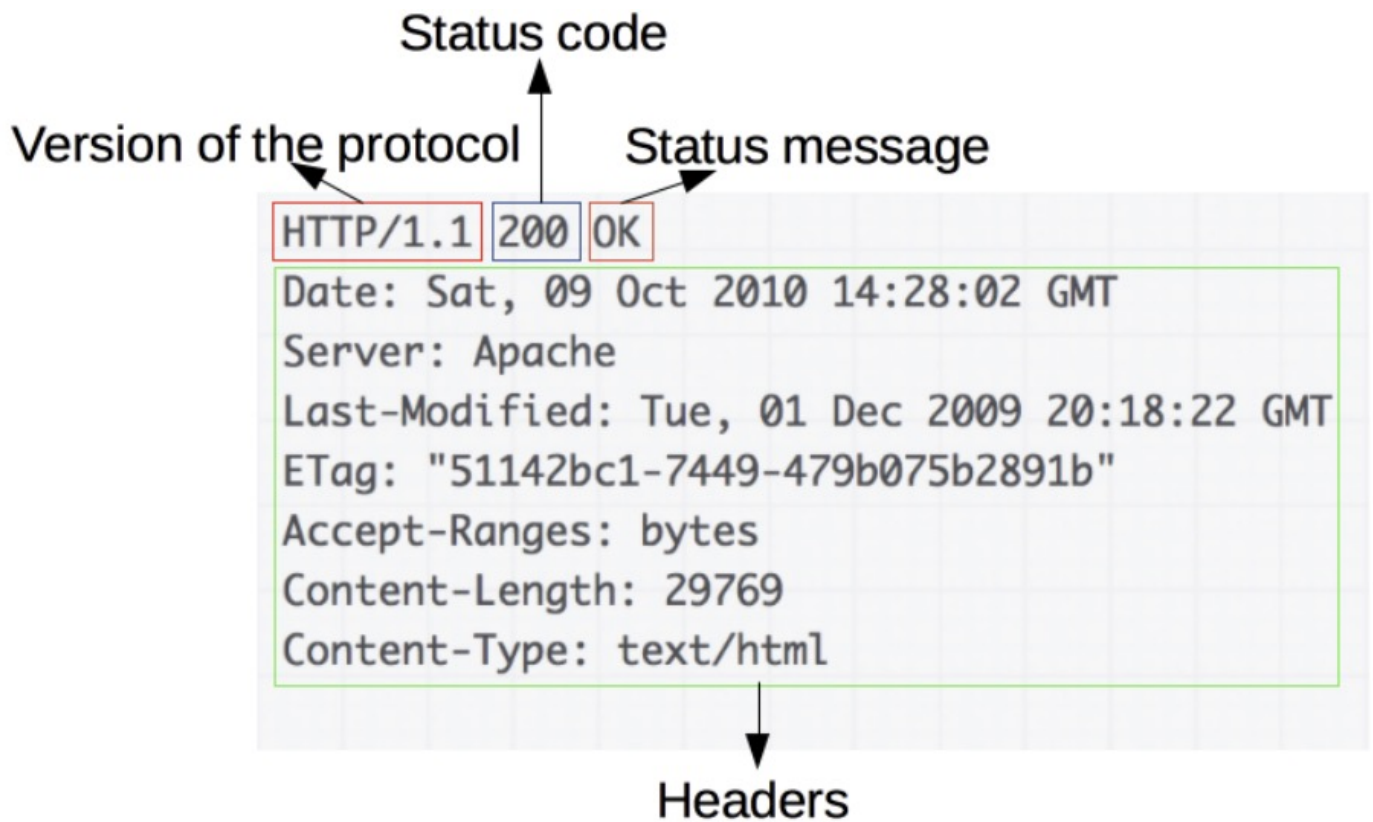
Path : 가져오려는 소스의 경로

Version of the Protocol : HTTP 프로토콜의 버전

Headers : 서버에 대한 추가 정보를 전달하는 선택적 헤더들

Etc : POST와 같은 몇 가지 메서드를 위한, 전송된 리소스를 포함하는 응답의 본문과 유사한 본문.

Response 메시지 구조



Version of the Protocol : HTTP 프로토콜의 버전

Status Code : 요청의 성공 여부와, 그 이유를 나타내는 상태코드

Status Message : 아무런 영향력이 없는, 상태 코드의 짧은 설명

Headers : 요청 헤더와 비슷한 HTTP 헤더들

Etc : 선택 사항으로, 가져온 리소스가 포함되는 본문

주요 HTTP 헤더 종류

키:값 쌍으로 이루어져 있으며 메시지의 성격과 형식을 알린다. 주요 헤더로는 다음과 같다.

Content-Type

전송하는 데이터의 타입을 지정한다(application/json, text/html)

Authorization

인증 정보를 담는다(토큰, 기본 인증)

Cookie

클라이언트가 서버에 저장된 쿠키를 전달한다

Accept

클라이언트가 처리할 수 있는 미디어 타입을 서버에 알린다

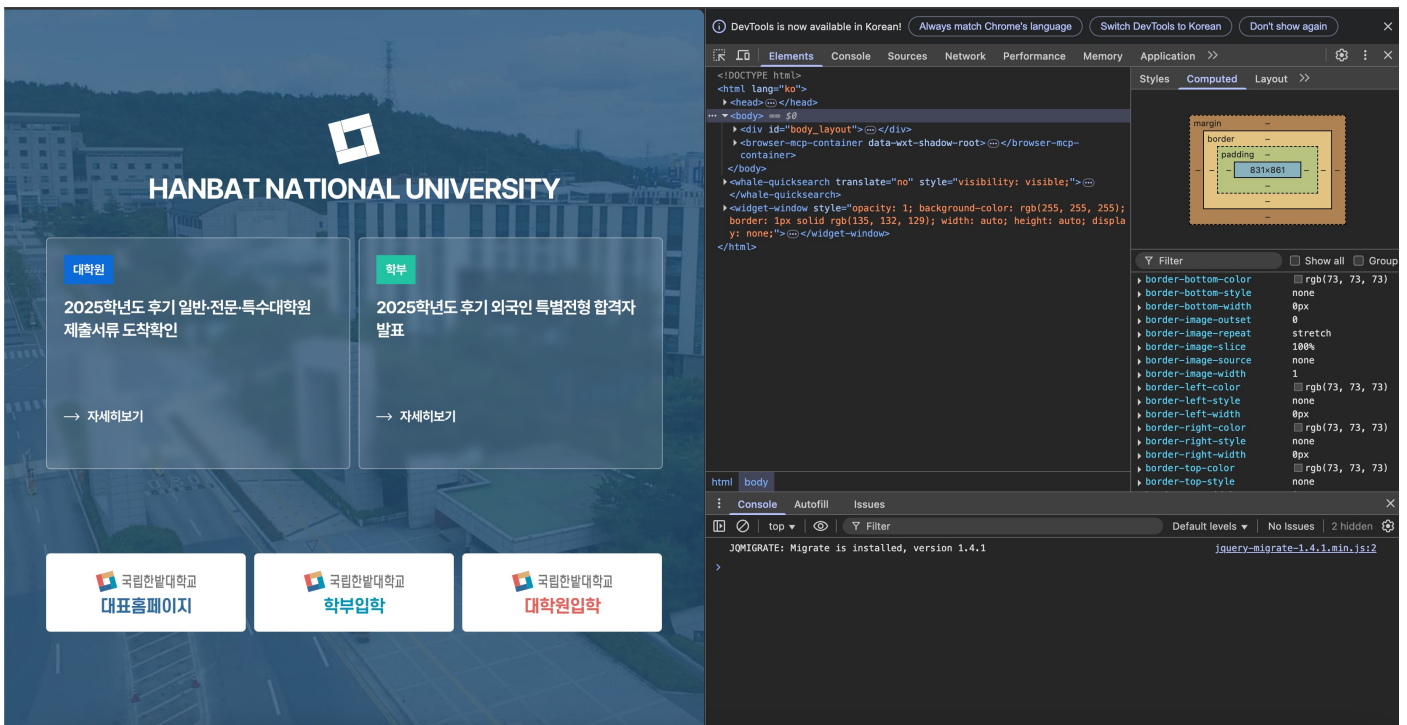
User-Agent

클라이언트 소프트웨어 정보(브라우저 종류 등)

1-4. 개발자 도구

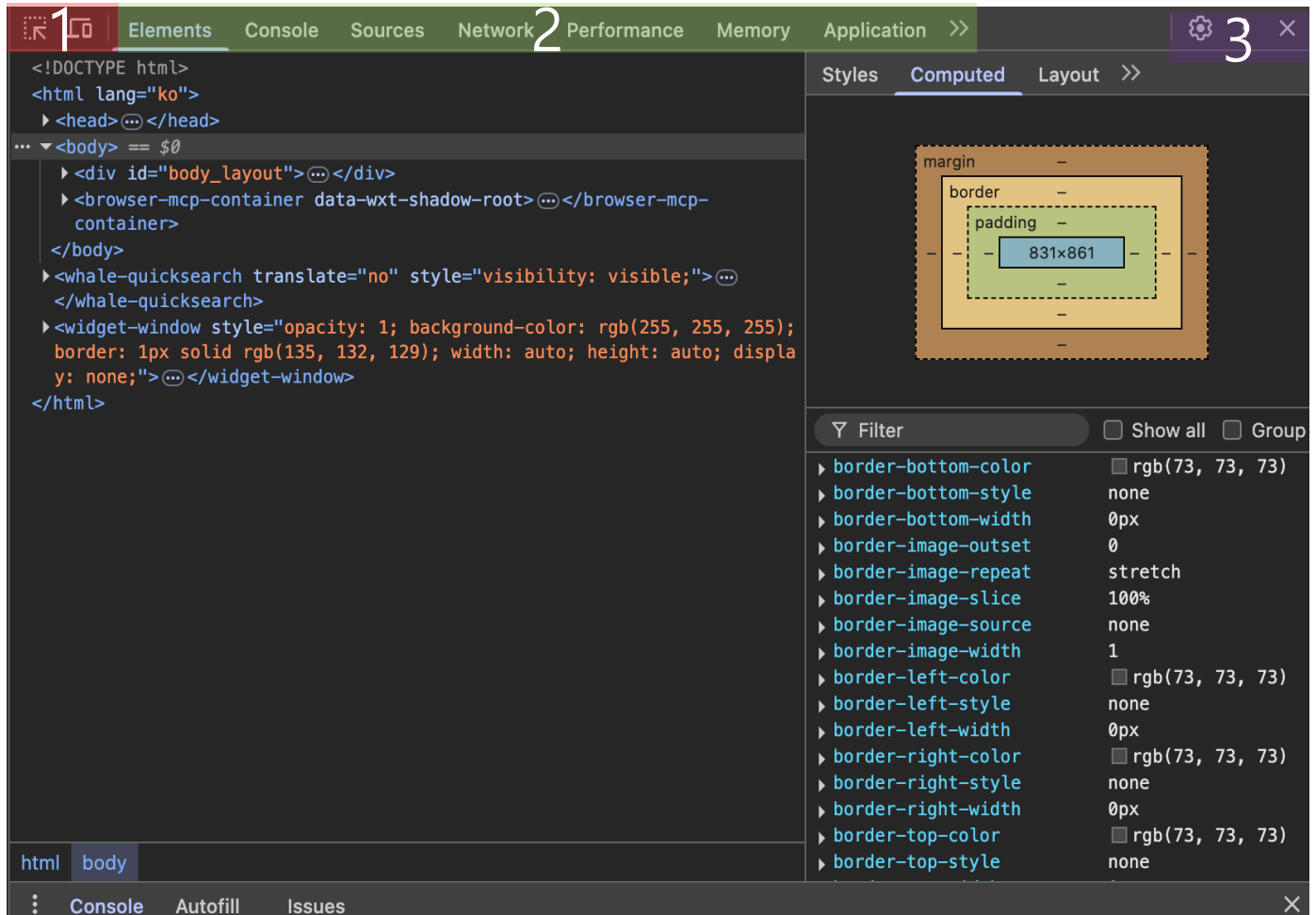
개발자 도구(Developer Tools)

개발자 도구는 웹 브라우저에 내장된 다양한 기능들을 모아 놓은 도구 모음이다. 웹 페이지를 개발, 디버깅, 분석하는 데 사용되며, 주로 웹 개발자들이 페이지의 구조, 스타일, 동작을 검사하고 문제를 찾아내어 성능을 최적화하는데 활용한다. 개발자 도구는 F12 키를 눌러서 들어갈 수 있다.



개발자 도구가 열려 있는 상태에서 다시 한 번 F12 키를 누르면 개발자 도구를 닫을 수 있다. 또한, 화면과 개발자 도구 사이의 경계선에 마우스를 올리면 화살표 모양이 나타나며, 이를 드래그하여 개발자 도구 창의 크기를 조절할 수 있다.

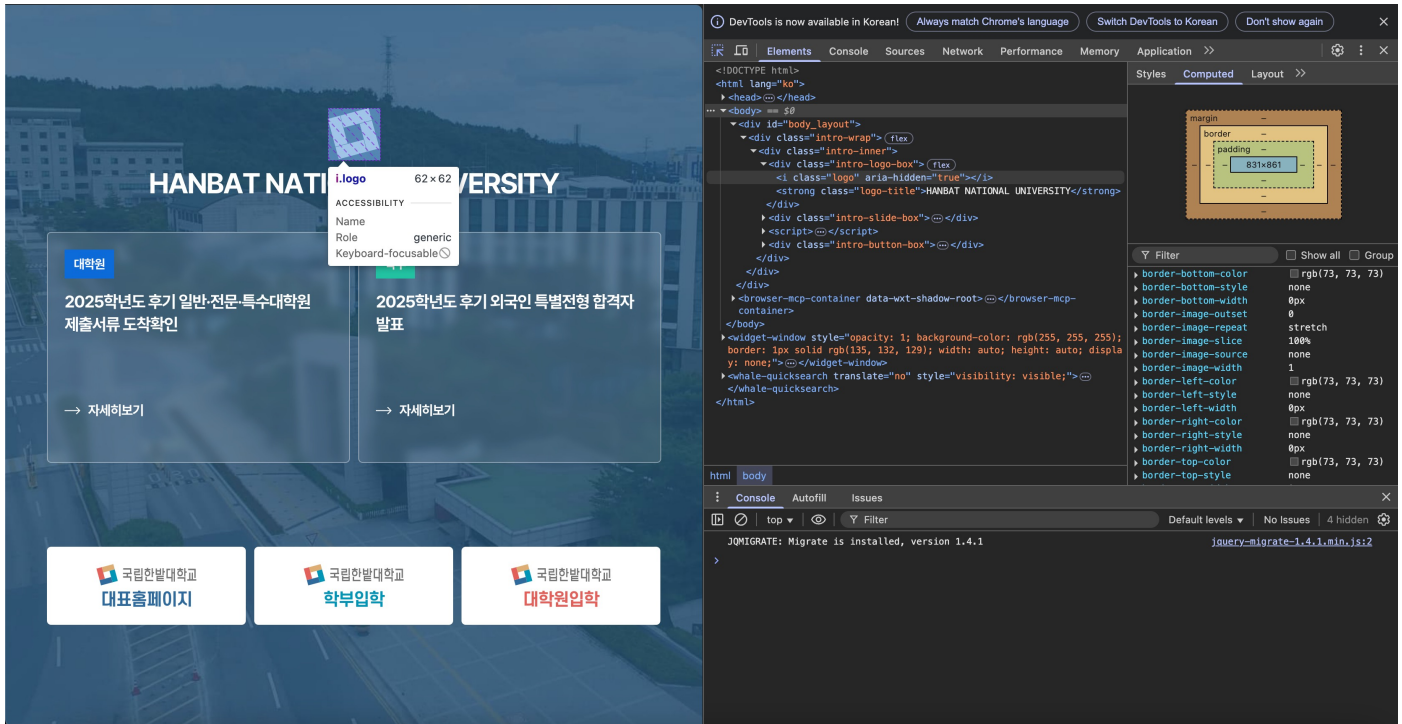
개발자 도구 레이아웃



1. 요소 검사(Inspect) 및 디바이스 툴바(Device Toolbar)
2. 기능을 선택하는 패널
3. 개발자 도구 설정

대표적으로 크게 위의 3가지 영역으로 나뉜다.(숫자는 순서와 상관X)

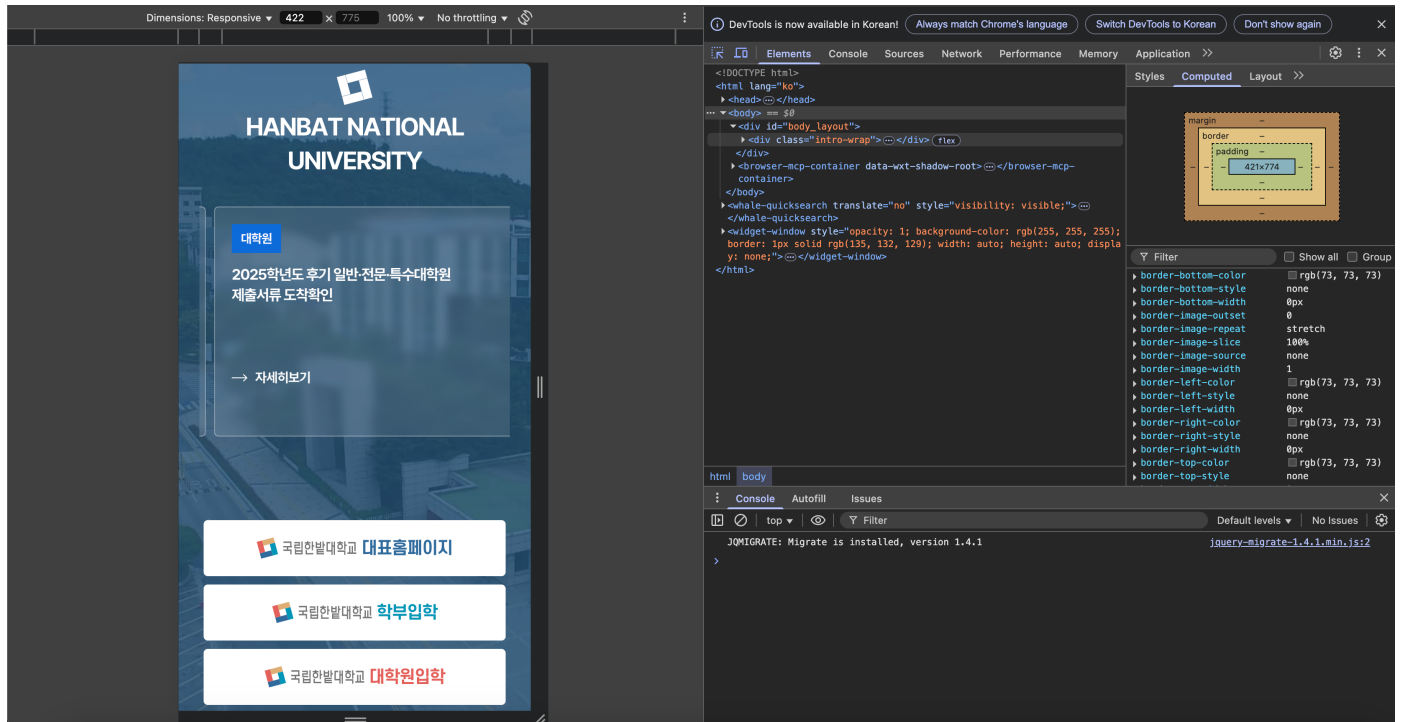
요소 검사(Inspect)



요소 검사를 활용하면 특정 요소의 개괄적인 정보를 파악하고, 관련 코드를 쉽게 찾을 수 있다. 요소 검사 버튼을 클릭한 뒤 웹 페이지에서 원하는 요소 위에 마우스를 올리면 해당 요소의 정보가 화면에 표시된다. 이 상태에서 클릭하면 해당 요소와 관련된 HTML 코드가 강조되어 나타난다.

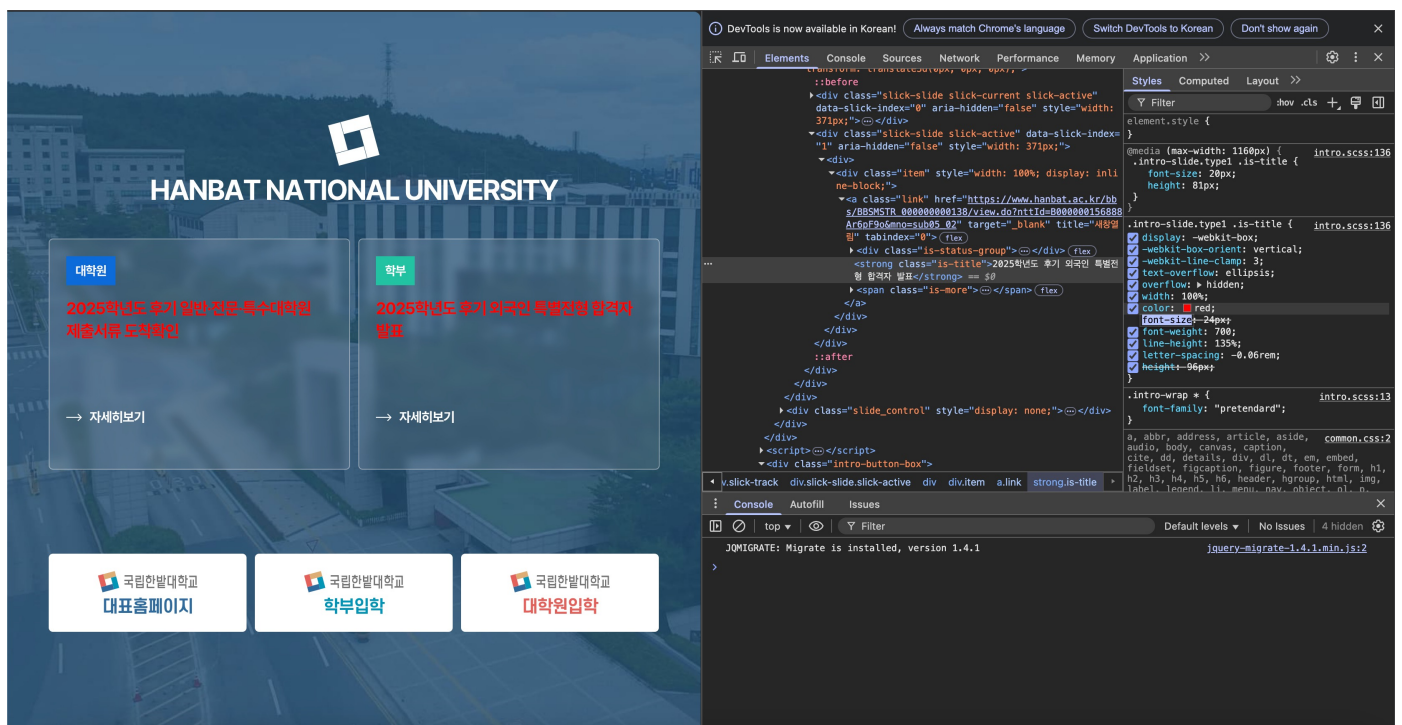
디바이스 툴바(Device Toolbar)

디바이스 툴바를 활용하면 현재 브라우저의 화면 비율 및 User-Agent값을 원하는 값으로 변경할 수 있다. 이를 통해 데스크톱, 태블릿, 스마트폰, 노트북 등 다양한 기기에 맞춰 웹 페이지가 어떻게 렌더링 되는지 테스트할 수 있다.



Elements panel

개발자 도구를 키면 나오는 기본으로 표시되는 탭으로, 웹 페이지의 구성(DOM)과 CSS를 확인할 수 있다.



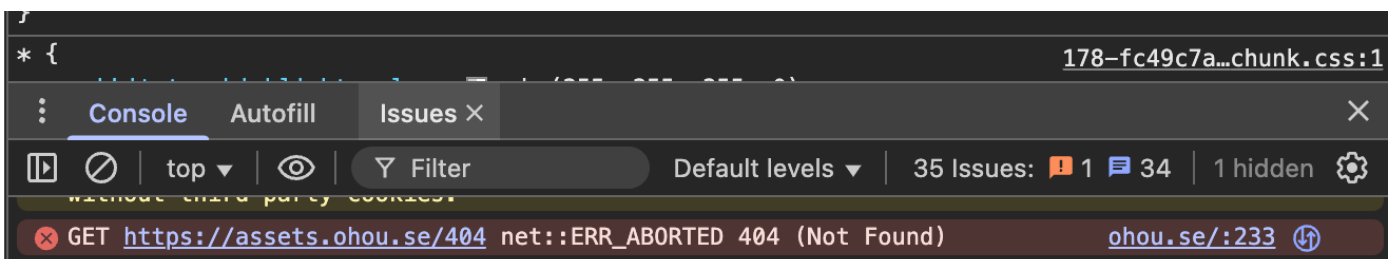
HTML코드를 분석하고 실시간으로 수정할 수 있는 공간이며, 예를 들어 특정 요소의 스타일에서 color 값을 빨간색(red)으로 변경하면 즉시 화면에 반영되는 것을 확인할 수 있다. 단, 페이지를 새로고침하면 변경 내용은 원래 상태로 돌아간다.

Console panel

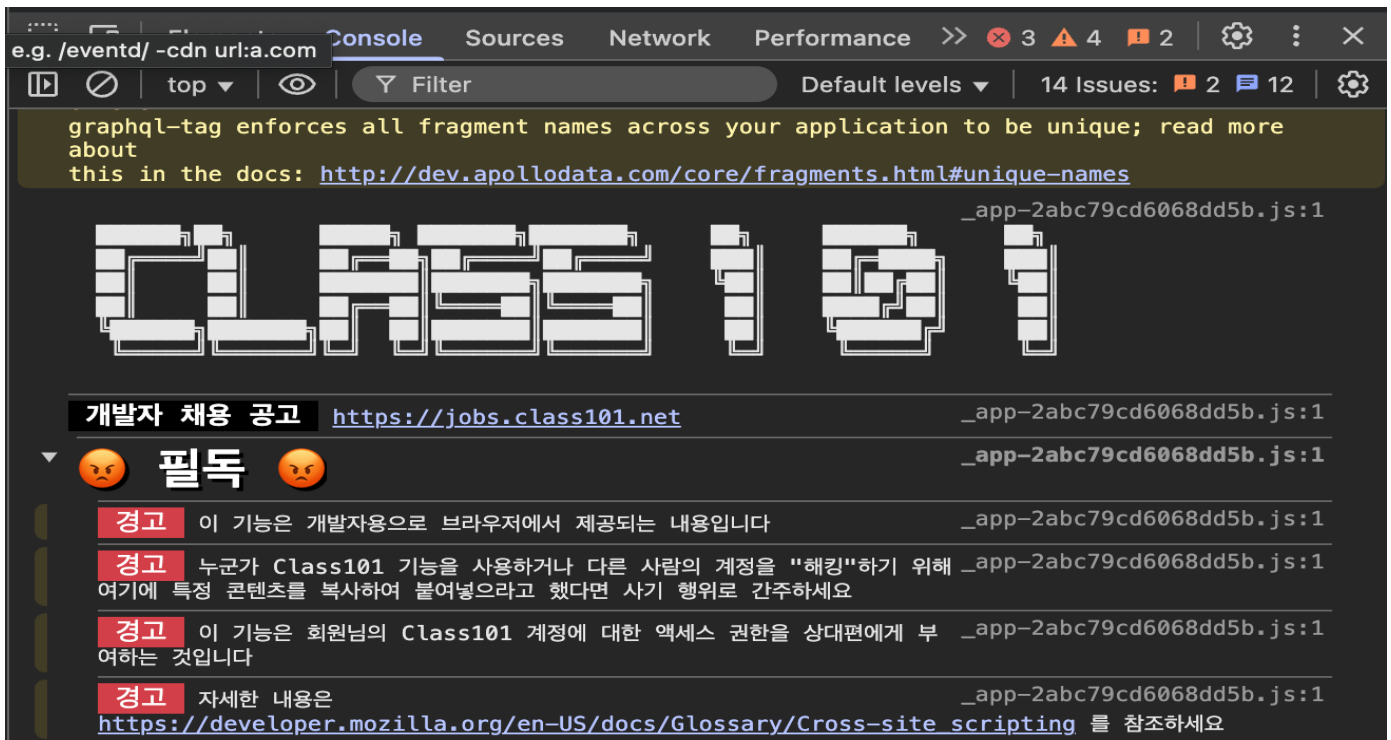
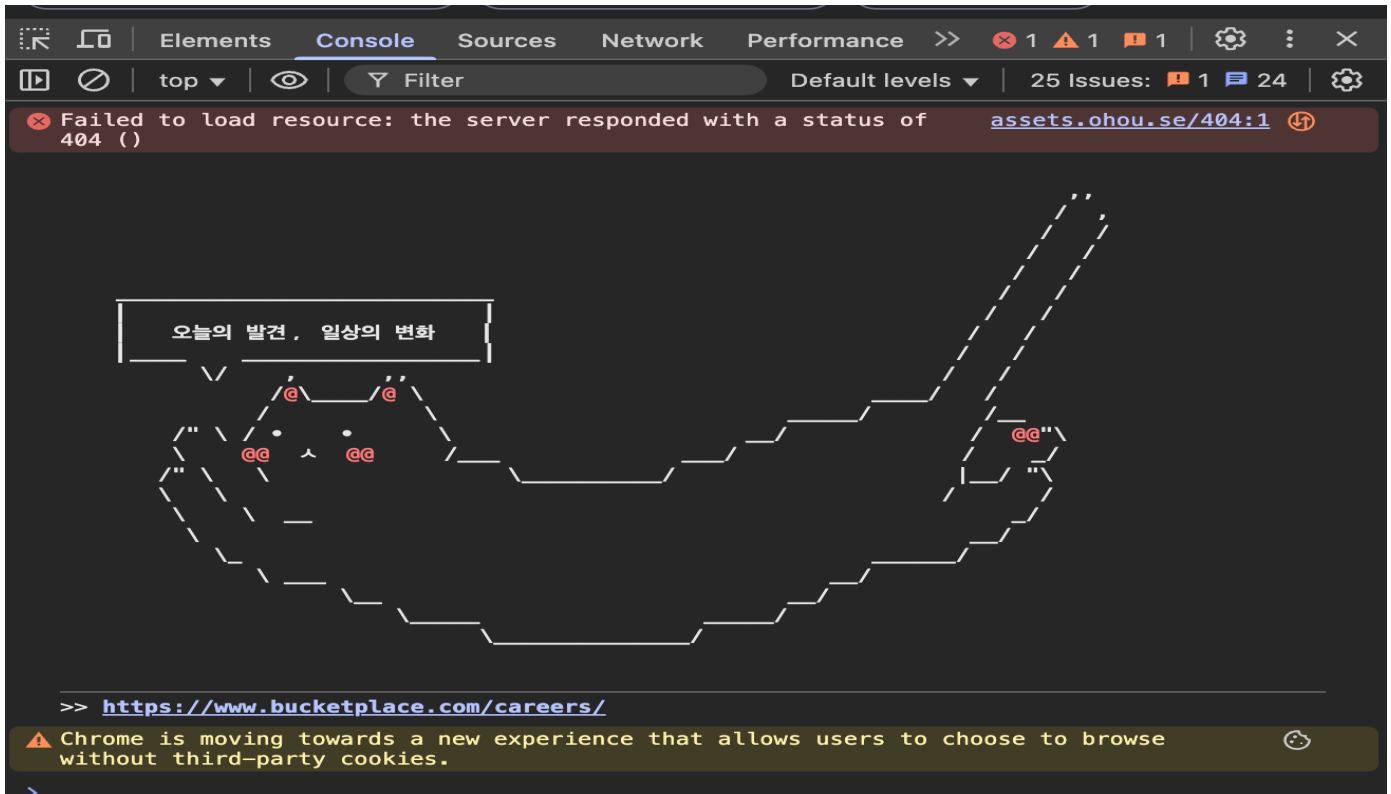
Console 패널은 개발자 도구에서 자바스크립트 코드 실행 결과와 오류 메시지를 확인할 수 있는 탭이다. 웹 페이지에서 발생하는 자바스크립트 에러를 실시간으로 보여주며, 개발자가 직접 자바스크립트 코드를 입력해 테스트하거나 디버깅할 수 있다. `console.log()` 같은 명령어로 출력된 로그 메시지도 여기서 확인할 수 있다.

```
> console.log("안녕");
안녕 VM462:1
< undefined
> document.cookie;
'bp-m-ver=1.0.2; bp-p-type=Web; bp-p-ver=Mozilla/5.0 (Macintosh, Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/134.0.0.0 Whale/4.31.304.16 Safari/537.36; bp'
```

다른 패널에 있었을 때 콘솔창을 보고 싶으면, ESC버튼을 누르면 아래쪽에 작은 콘솔창이 뜨는 것을 확인할 수 있다. 일반 콘솔창처럼 사용이 가능하다. 오류 발생시 상태코드와 메시지가 같이 출력되는 것도 확인할 수 있다.

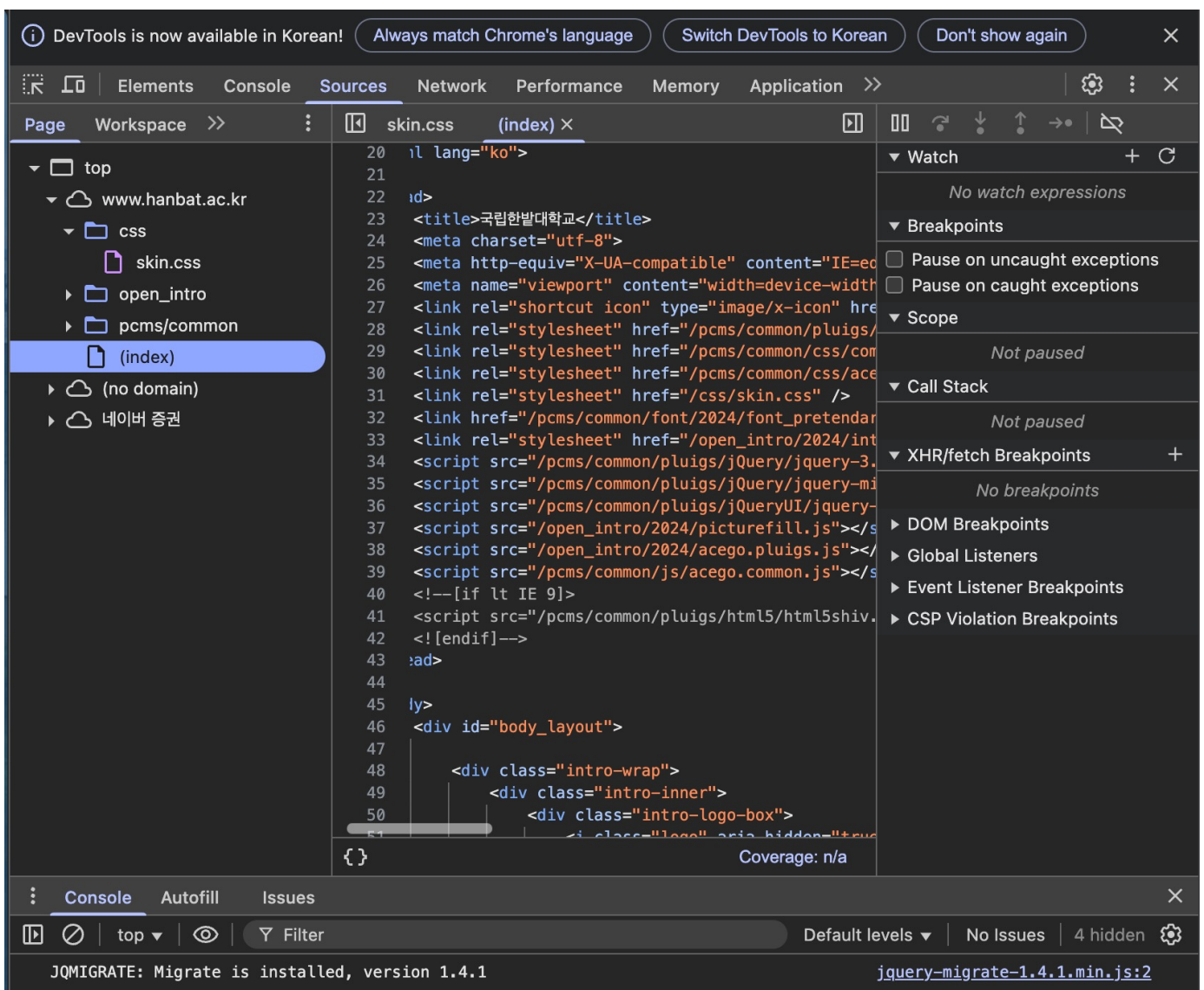


‘오늘의 집’과 ‘클래스 101’ 웹사이트에서 개발자 도구 console창을 키면 재미난 로그를 확인할 수 있다.



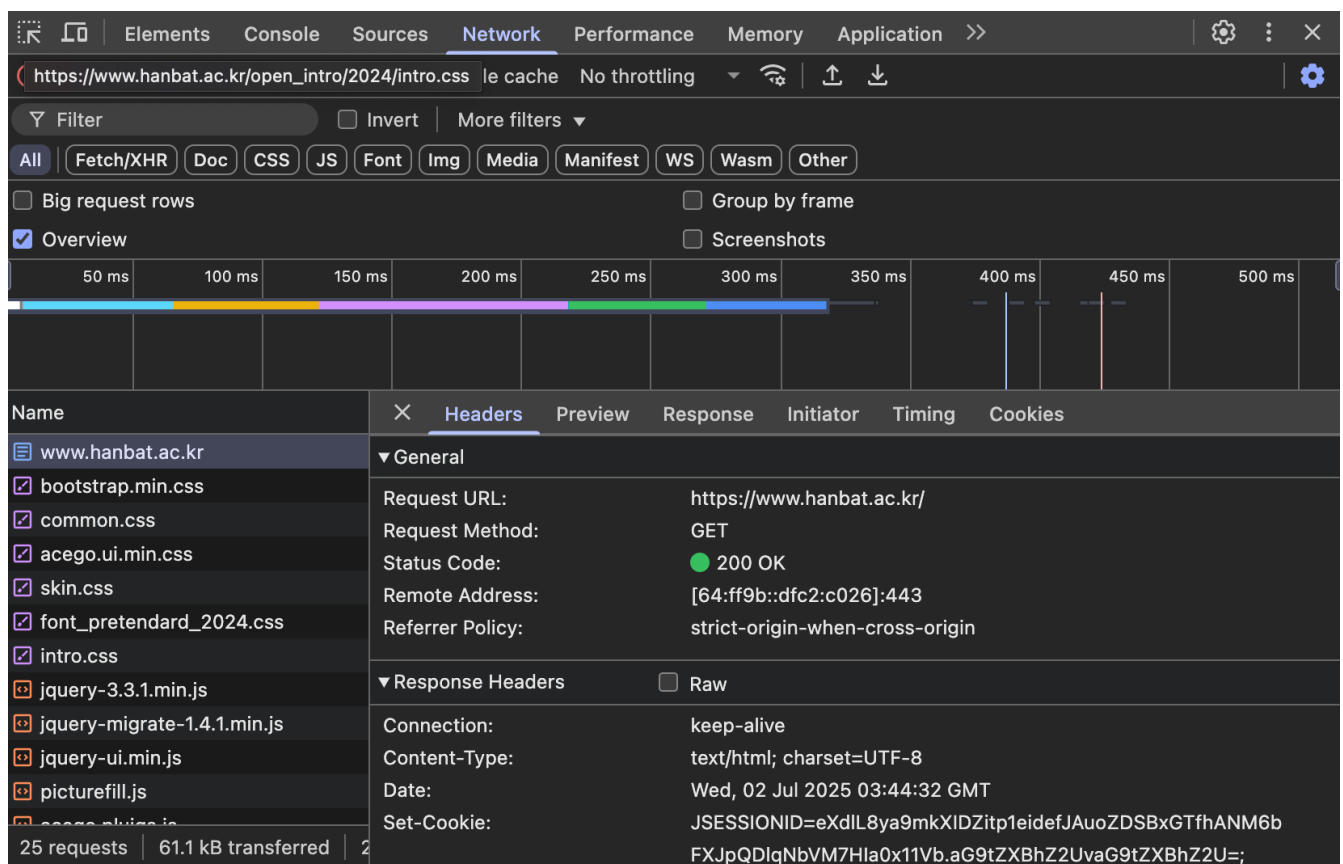
Source panel

Source 패널은 개발자 도구에서 웹 페이지를 구성하는 모든 소스 파일(HTML, CSS, JavaScript 등)을 확인하고 디버깅할 수 있는 탭이다. 이곳에서 자바스크립트 코드를 직접 보고, 중단점(breakpoint)을 설정하여 코드 실행을 단계별로 추적할 수 있다. 또한, 소스 파일을 실시간으로 수정해 페이지에 적용해 볼 수 있으며, 파일을 탐색하거나 검색하는 기능도 제공한다.



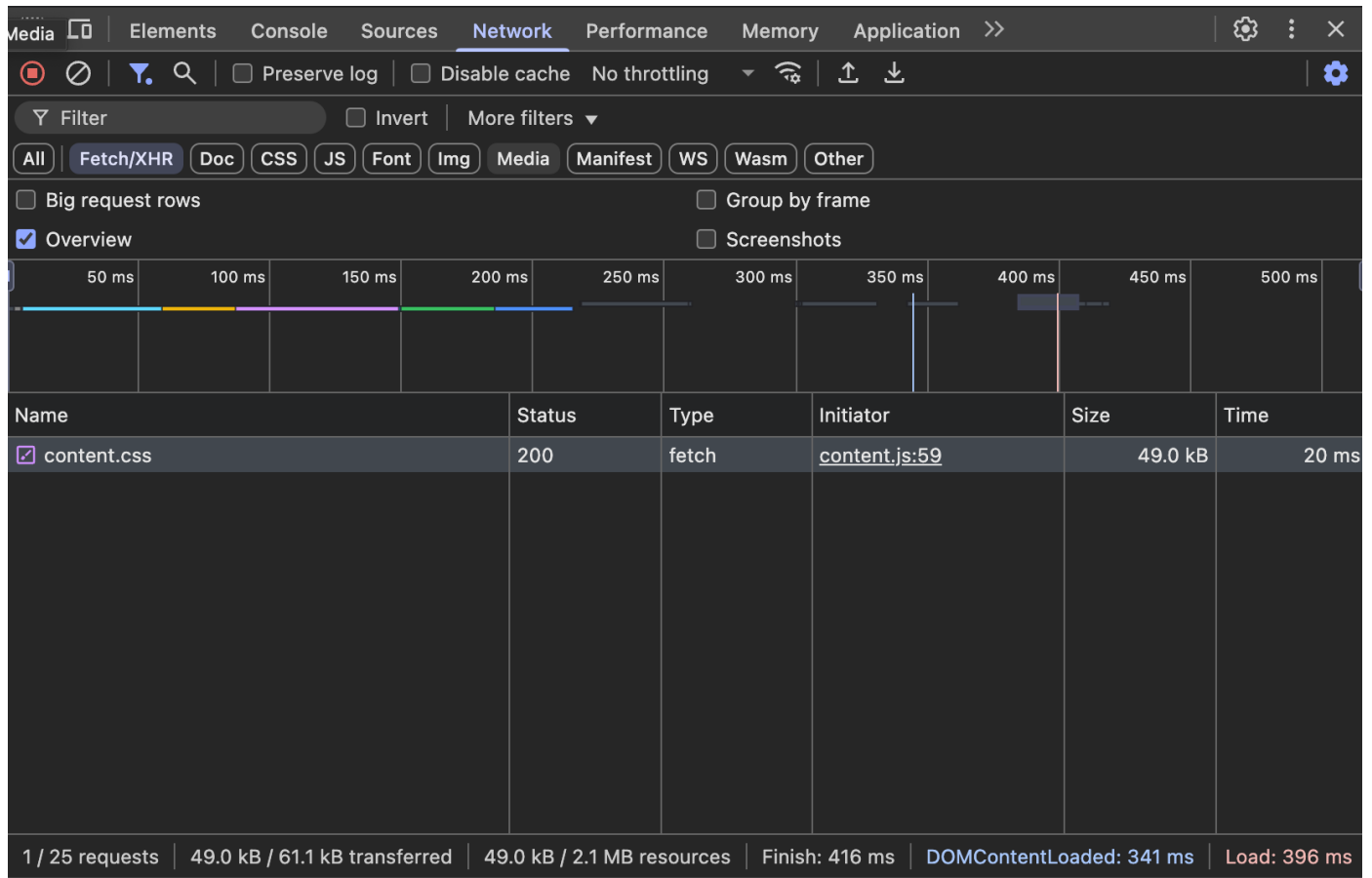
Network panel

매우 중요하다. 웹 사이트에서 통신하고 있는 모든 정보 목록을 보여준다. 리소스들이 예상대로 다운로드 되고 업데이트 되는지 확인할 때 사용한다. 웹 페이지와 서버 사이에서 데이터 흐름을 분석하고, 문제 발생 시 해결책을 찾도록 개발자를 돕는 역할을 한다.



어떤 리소스들이 업로드 및 다운로드 되는지 내역을 목록 형태로 확인할 수 있다. 각 항목을 클릭하면 요청 및 응답에 포함된 헤더 정보, 전송된 데이터 내용, 상태 코드 등 상세한 정보를 확인할 수 있다.

네트워크 패널에도 많은 탭들이 존재한다.



이중, Fetch/XHR 탭에서는 API 정보(request, response)를 확인할 때 사용한다. 여기서 중요하게 알아야 된다.

- Name : 리소스의 이름과 URL 표시
- Type : 파일의 형식
- Initiator : 요청을 발생시킨 코드 표시(클릭하면 코드로 이동)
- Size : 리소스들의 파일 사이즈
- Time : 요청과 응답까지 걸린 시간 확인
- Waterfall : 타임라인 세부 정보

Application Panel

Application 패널은 브라우저에 저장된 데이터를 관리하는 공간이다. 주요 저장소(Storage)로는 총 세가지가 있다.

Local Storage

클라이언트에 저장되며, 사용자가 직접 삭제하지 않는 한 브라우저에 계속 남아있다. 그렇기에 장기간 보관해야 하는 데이터를 저장할 때 사용된다.

예) 자동로그인 기능, 비회원 장바구니 등

Session Storage

클라이언트에 저장되며, 브라우저 탭이 닫힐 때 까지 유지된다. 잠깐 동안만 필요한 정보를 저장하는 데 적합하다.

예) 일시적인 사용자 입력 데이터, 페이지 상태 정보 등

Cookie

클라이언트에 저장되며, 서버가 응답을 통해 설정할 수 있다. 저장 용량이 작고, 만료시간을 설정할 수 있다. 프론트엔드와 백엔드 간 통신에 자주 사용되며, 요청 시 자동으로 서버에 전송된다. 보통 서비스와 직접적으로 연관되지 않은 정보를 저장하는 데 사용된다.

예) 서비스 약관 동의 여부, 일일 이벤트 팝업 표시 여부 등

DevTools is now available in Korean! [Always match Chrome's language](#) [Switch DevTools to Korean](#) [Don't show again](#) ✕

Elements Console Sources Network Performance Memory Application >> ⚙️ ⋮ ✕

Application

- Manifest
- Service workers
- Storage
 - Local storage
 - Session storage
 - Extension storage
 - IndexedDB
 - Cookies
 - https://www.hanbat... **Clear**
 - Private
 - Interest groups
 - Shared storage
 - Cache storage
 - Storage buckets
- Background services
 - Back/forward cache
 - Background fetch
 - Background sync

Name	Value	D..	P...	Ex...	Size	Ht...	Se...	Sa...	Pa...	Cr...	Pri...
FLOWKAT	x57kcj2phjpvur	w...	/	20...	21						M...
JSESSIONID	u2UpP3Kqv0m9k2d4aXBGg...	w...	/	Se...	99	✓					M...
_ga	GA1.1.1290832781.17140319...		/	20...	30						M...
_ga_0E5GPYT...	GS2.1.s1751450317\$o64\$g1...		/	20...	60						M...
_ga_30RHD4...	GS1.1.1733191078.25.0.1733...		/	20...	52						M...
_ga_CN39T42...	GS1.1.1741675936.2.0.17416...		/	20...	52						M...
_ga_DWYKZT...	GS2.1.s1750828546\$o51\$g...		/	20...	60						M...
_ga_G44MDS...	GS1.1.1730174259.7.1.17301...		/	20...	52						M...
_ga_MW27H...	GS2.1.s1751256840\$o1\$g0...		/	20...	59						M...
_ga_V8J0Y78...	GS1.1.1742618208.2.0.17426...		/	20...	51						M...
_ga_WJW6YC...	GS1.1.1722952243.5.1.17229...		/	20...	60						M...
_gid	GA1.3.1442538472.1751450...		/	20...	31						M...

No cookie selected
Select a cookie to preview its value

Console Autofill Issues ✕

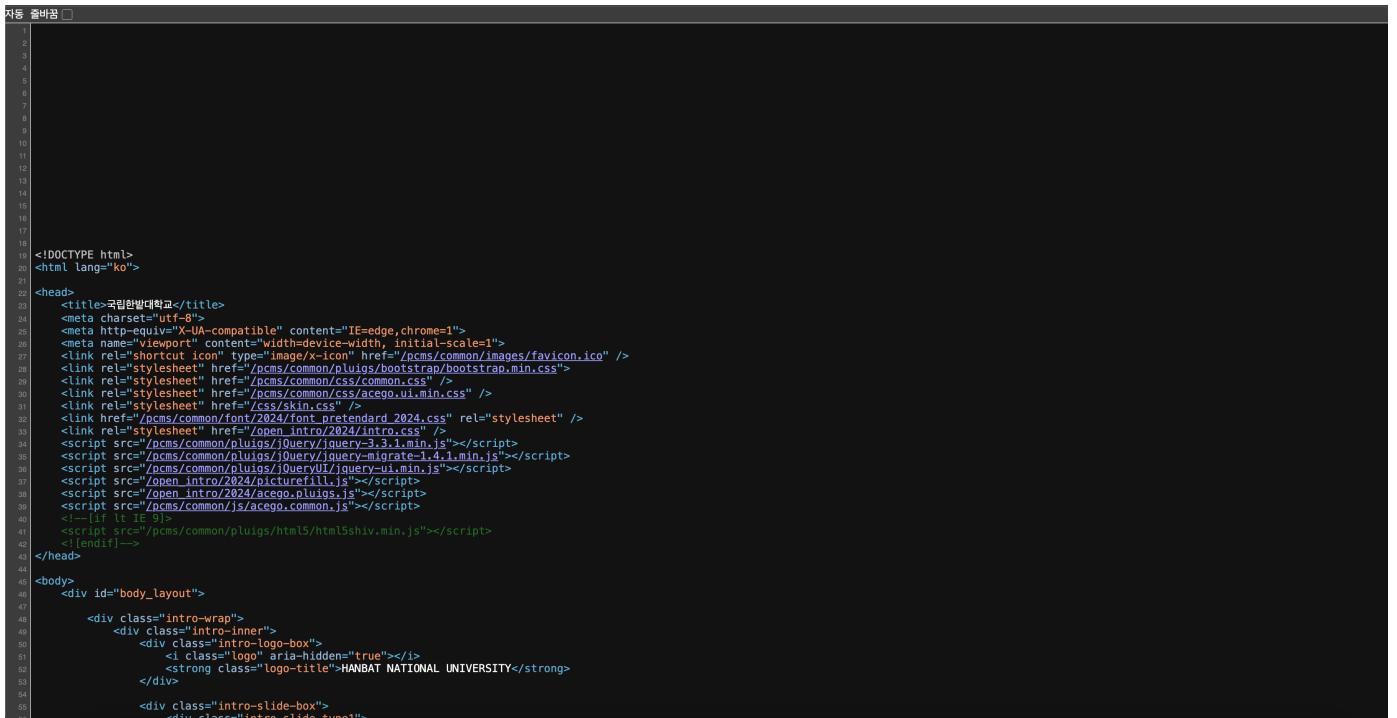
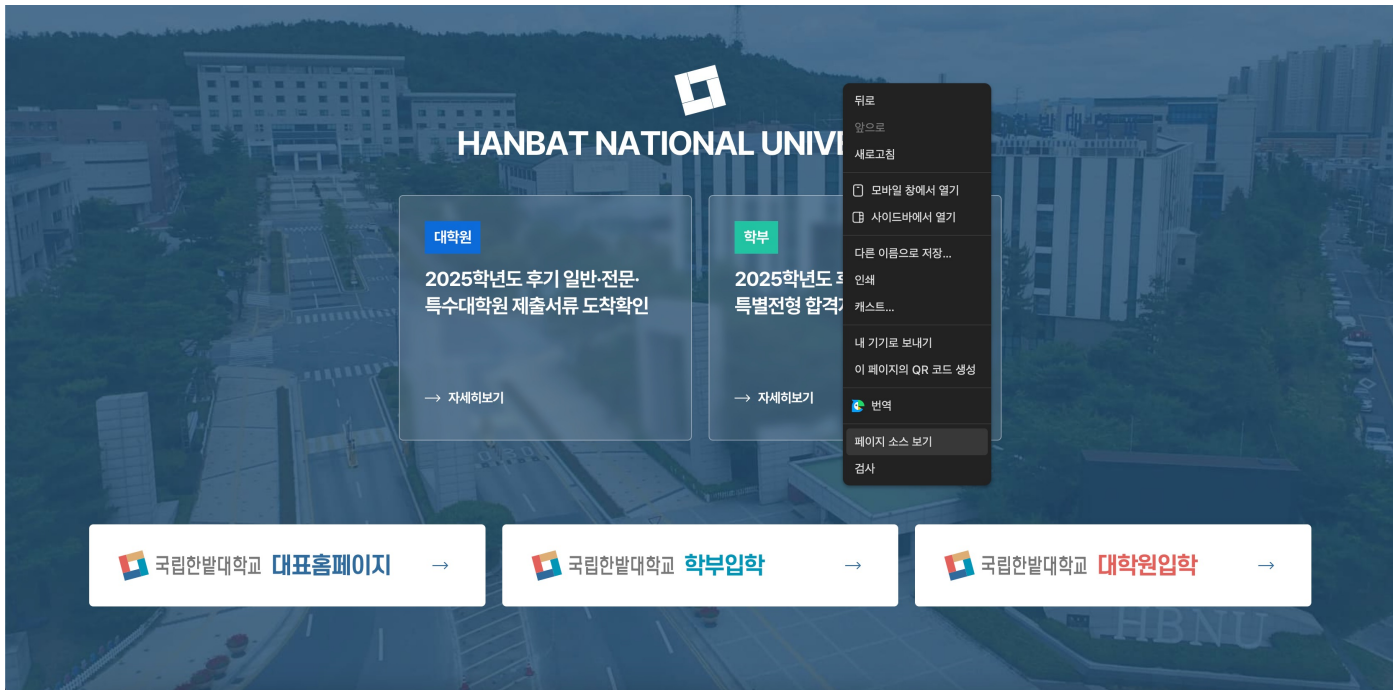
top ▼ 🔍 Filter Default levels ▼ No Issues 4 hidden ⚙️

Application �에서 저장소에서 볼 수 있는 화면이다. 해당 저장소의 값을 지우기 위해서 우클릭을 누른 후 Clear 버튼을 누르면 해당 데이터가 전부 지워진다.

🔄 Filter ✕ Only show cookies with an issue											
Name	Value	D..	P...	Ex...	Size	Ht...	Se...	Sa...	Pa...	Cr...	Pri...

기타 브라우저 기능

현재 보고 있는 웹 페이지의 소스 코드를 확인할 수 있다. 웹 페이지에서 마우스 오른쪽 버튼을 클릭한 후, '페이지 소스 보기' 메뉴를 선택하면 볼 수 있다.



제 2 장

REACT

- 1) 개발 환경 설정
- 2) React 기초
- 3) 스타일링
- 4) REACT HOOKS
- 5) 데이터 통신 및 CRUD

2-1. 맥북 환경 준비 (Homebrew, Node.js, npm 설치)

Homebrew는 맥에서 필요한 프로그램을 쉽게 설치할 수 있는 패키지 관리자입니다. Node.js는 자바스크립트 실행 환경이며, React 프로젝트 실행에 필수적입니다. npm(Node Package Manager)**은 Node.js에서 사용하는 패키지 관리 도구입니다.

- Homebrew: Mac 패키지 관리자
- Node.js: JS 실행 환경
- npm: Node 패키지 관리자

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
brew install node
node -v
npm -v
```

2-1. Homebrew 소개

Homebrew 소개

- Homebrew는 맥에서 필요한 프로그램을 쉽게 설치할 수 있는 패키지 관리자입니다.
- Homebrew를 사용하면 Node.js, Git, Python 등 다양한 개발 도구를 간단한 명령어로 설치할 수 있습니다.
- 즉, 개발자의 “앱스토어”와 같은 역할을 하는 도구입니다.

💡 Tip: Homebrew는 `/opt/homebrew` 경로에 설치되며, M1/M2 맥북은 이 경로가 기본입니다.

2-1. Homebrew 설치 과정

아래 명령어를 터미널에 입력하면 Homebrew가 자동으로 설치됩니다.

- 설치가 완료되면 `brew -v` 명령으로 버전을 확인할 수 있습니다.
- 설치 후 `brew doctor` 명령으로 정상 설치 여부를 확인하는 것이 좋습니다.

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

2-1. PATH 환경 변수 설정

- 설치 직후 brew 명령어가 작동하지 않는 경우가 있습니다.
- 이는 **PATH 환경 변수**가 올바르게 등록되지 않았기 때문입니다.
- 아래 명령어로 환경 변수를 설정합니다.

```
bash
```

```
echo 'eval "$(/opt/homebrew/bin/brew shellenv)"' >> ~/.zprofile  
source ~/.zprofile
```

2-1. Node.js와 npm 소개

- Node.js는 자바스크립트를 브라우저 밖에서 실행할 수 있게 해주는 런타임입니다.
- ****npm(Node Package Manager)****은 Node.js에서 사용하는 패키지 관리자입니다.
- React 프로젝트를 생성하고 실행하는 데 필수적인 도구입니다.

💡 Tip: Node.js는 LTS(Long Term Support) 버전을 설치하는 것이 안정적입니다.



2-1. Node.js와 npm 설치

Homebrew로 Node.js를 설치할 수 있습니다.

```
bash  
  
brew install node
```

설치 후 아래 명령어로 정상 동작 여부를 확인합니다.

```
bash  
  
node -v  
npm -v
```

2-1. nvm(Node Version Manager) 소개

- 프로젝트마다 Node.js 버전이 달라질 수 있습니다.
- nvm을 사용하면 여러 버전을 설치하고 필요할 때마다 전환할 수 있습니다.
- 팀 프로젝트에서는 nvm을 통해 동일한 버전을 맞추는 것이 좋습니다.

2-1. nvm 설치 및 사용법

nvm 설치 및 기본 사용법은 아래와 같습니다.

```
bash

brew install nvm
mkdir ~/.nvm
echo 'export NVM_DIR="$HOME/.nvm"' >> ~/.zshrc
source ~/.zshrc
nvm install --lts
nvm use --lts
```

설치 후 `node -v`로 버전을 확인하면 정상 동작합니다.

2-1. npm과 npx의 차이

- npm install: 패키지를 프로젝트에 설치합니다.
- npm run: package.json에 정의된 스크립트를 실행합니다.
- npx: 패키지를 설치하지 않고도 바로 실행할 수 있습니다.

```
bash
```

```
npm install axios
```

```
npm run start
```

```
npx create-react-app my-app
```

2-1. VSCode 소개

- Visual Studio Code(VSCode)는 무료이면서 강력한 기능을 가진 코드 에디터입니다.
- React 개발에 최적화된 환경을 제공합니다.
- 확장 프로그램과 테마를 이용해 맞춤형 개발 환경을 구축할 수 있습니다.



2-1. VSCode 확장 프로그램 설치

React 개발 시 유용한 확장 프로그램은 다음과 같습니다.

- ES7+ React Snippets: 컴포넌트 자동 생성 단축키 제공
- Prettier: 코드 자동 정렬
- React Developer Tools: 컴포넌트 상태와 구조 확인

```
jsonc

// .prettierrc 예시
{
  "singleQuote": true,
  "semi": true
}
```

2-1. 터미널 기본 명령어

- React 프로젝트 실행 전 기본 터미널 명령어를 알아야 합니다.

```
bash
```

```
pwd    # 현재 위치 확인
```

```
ls      # 파일 목록 보기
```

```
cd      # 디렉토리 이동
```

```
mkdir my-app # 새 폴더 생성
```

2-1. Git 초기화

- 프로젝트는 보통 Git으로 관리합니다.
- 다음 명령으로 저장소를 초기화합니다.

```
bash

git init
echo "node_modules/" >> .gitignore
git add .
git commit -m "프로젝트 초기 설정"
```

2-1. React 프로젝트 생성

- 프로젝트는 CRA(Create React App) 명령어로 React 프로젝트를 생성합니다.

```
bash

npx create-react-app my-app
cd my-app
npm start
```

- 실행하면 <http://localhost:3000> 에서 React 앱이 열립니다.

2-1. 폴더 구조 이해

- 프로젝트 생성 후 구조는 아래와 같습니다.

💡 Tip: React 앱은 `src/index.js` → `App.js` 순서로 실행됩니다.

```
my-app/  
├─ public/      # index.html 포함  
├─ src/         # 컴포넌트 소스코드  
└─ package.json # 의존성 관리
```

2-2. React란?

- React는 페이스북(현 메타)에서 개발한 자바스크립트 UI 라이브러리입니다. 기존 웹 개발 방식에서는 사용자가 페이지의 일부만 바뀌도 전체 페이지가 다시 로드되어 비효율적이었습니다. 그러나 React는 Virtual DOM을 도입하여 필요한 부분만 빠르게 갱신합니다. 이로 인해 화면 전환이 부드럽고 사용자 경험이 크게 개선됩니다.
- 또한 React는 컴포넌트 기반 구조를 채택합니다. 하나의 큰 페이지를 여러 개의 작은 블록(컴포넌트)으로 나누어 개발할 수 있기 때문에, 유지보수가 쉽고 코드 재사용성이 높습니다. 예를 들어, 버튼 컴포넌트를 만들어 두면 로그인, 회원가입, 결제 등 다양한 화면에서 동일한 버튼을 불러와 사용할 수 있습니다.

```
jsx

// 가장 기본적인 React 컴포넌트
function Hello() {
  return <h1>Hello React!</h1>;
}
```

2-2. React의 장점

React가 전 세계적으로 널리 쓰이는 이유는 분명합니다

1. Virtual DOM 기술로 불필요한 렌더링을 줄여 빠른 성능을 보장합니다. 기존 DOM은 크고 무겁기 때문에 작은 변화에도 전체를 다시 그리지만, React는 메모리 속 가상의 DOM을 이용해 차이를 계산한 뒤 실제 DOM에 최소한만 반영합니다.
2. 컴포넌트 기반 개발입니다. UI를 작게 나누어 개발하면 코드의 가독성이 높아지고, 재사용성이 커집니다. 같은 버튼·입력창을 여러 화면에서 다시 만들 필요 없이 한 번 만들어두고 가져와 쓰면 됩니다. 이는 팀 단위 개발에서도 큰 장점으로 작용합니다.
3. 강력한 생태계입니다. React에는 라우팅을 담당하는 react-router, 상태 관리를 위한 redux·recoil, 스타일링을 위한 styled-components, 데이터 통신을 단순화하는 axios 등 수많은 라이브러리가 존재합니다. React 하나를 배우면 주변 생태계 도구까지 활용할 수 있어 생산성이 비약적으로 올라갑니다.

2-2. 함수형 컴포넌트와 클래스형 컴포넌트

- React 컴포넌트는 크게 함수형(Function Component)과 클래스형(Class Component)으로 나뉩니다.
- 함수형 컴포넌트는 단순한 자바스크립트 함수 형태로 작성되며, React 16.8부터 도입된 Hooks(useState, useEffect 등)을 활용하여 상태 관리와 생명주기 기능을 구현합니다. 코드가 간결하고 직관적이어서 현재 대부분의 프로젝트에서 표준으로 사용됩니다.

```
jsx

// 함수형 컴포넌트
function Greeting() {
  return <p>안녕하세요, React입니다!</p>;
}
```

2-2. 함수형 컴포넌트와 클래스형 컴포넌트

- **클래스형 컴포넌트**는 ES6 클래스 문법을 이용하여 작성합니다.
render() 메서드를 통해 UI를 반환하고, this.state, this.setState를 사용해 상태를 관리합니다.

이전에는 복잡한 기능 구현에 주로 쓰였지만, Hooks의 등장 이후 점차 사용이 줄어들고 있습니다. 다만 오래된 코드베이스에서는 여전히 클래스형 컴포넌트를 접할 수 있습니다.

```
jsx

// 클래스형 컴포넌트
import React, { Component } from "react";

class Greeting extends Component {
  render() {
    return <p>안녕하세요, 클래스형 컴포넌트입니다!</p>;
  }
}
```

2-2. Props란?

- Props는 부모 컴포넌트가 자식 컴포넌트에 값을 전달할 때 사용하는 속성입니다.
읽기 전용 데이터이므로 자식 컴포넌트에서 직접 수정할 수는 없습니다.
화면에 반복적으로 같은 UI를 그릴 때 Props를 사용하면 코드 재사용성이 높아집니다.

```
jsx
```

```
function Welcome(props) {  
  return <h1>안녕하세요, {props.name}님!</h1>;  
}
```

```
// 사용 예시
```

```
<Welcome name="민재" />
```

```
<Welcome name="지훈" />
```

2-2. State란?

- State는 컴포넌트 내부에서 관리하는 데이터입니다.
버튼 클릭, 입력값 변경 등 사용자 동작에 따라 State가 바뀌면 화면이 자동으로 다시 렌더링됩니다.
State는 useState Hook을 사용하여 정의합니다.

jsx

```
import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>
      현재 카운트: {count}
    </button>
  );
}
```

2-2. Props와 State 비교

- Props: 부모에서 자식으로 전달되는 값, 읽기 전용입니다.
- State: 컴포넌트 안에서 관리되는 값, 직접 업데이트할 수 있습니다.

즉, Props는 “외부에서 주어지는 값”, State는 “내부에서 변하는 값”으로 이해하면 됩니다.

2-2. 이벤트 처리 (onClick)

- React에서는 HTML과 달리 이벤트를 camelCase 형태로 작성합니다. 이벤트 핸들러는 보통 화살표 함수나 별도의 함수로 정의합니다.

jsx

```
function Button() {  
  const handleClick = () => {  
    alert("버튼이 클릭되었습니다!");  
  };  
  return <button onClick={handleClick}>클릭</button>;  
}
```

2-2. 이벤트 처리 (onChange)

- 입력값을 실시간으로 관리하려면 onChange 이벤트를 사용합니다. 입력된 값은 e.target.value로 가져올 수 있으며, 보통 State와 함께 사용합니다.

```
function InputBox() {  
  const [text, setText] = useState("");  
  return (  
    <>  
      <input value={text} onChange={(e) => setText(e.target.value)} />  
      <p>입력한 값: {text}</p>  
    </>  
  );  
}
```

2-2. 조건부 렌더링

React 에서는 특정 조건에 따라 다른 UI를 보여줄 수 있습니다. 대표적으로 삼항 연산자(? :)와 AND 연산자(&&)를 많이 사용합니다.

```
function LoginStatus({ isLogin }) {  
  return (  
    <div>  
      {isLogin ? <p>환영합니다!</p> : <p>로그인이 필요합니다.</p>}  
      {isLogin && <button>로그아웃</button>}  
    </div>  
  );  
}
```

2-2. 리스트 렌더링

배열 데이터를 JSX 로 출력 할 때는 `map()` 을 사용합니다 .
각 요소에는 반드시 `key` 속성을 부여해야 효율적인 업데이트가 가능합니다.

```
const fruits = ["사과", "바나나", "포도"];

function FruitList() {
  return (
    <ul>
      {fruits.map((fruit, idx) => (
        <li key={idx}>{fruit}</li>
      ))}
    </ul>
  );
}
```

2-2. JSX 문법 기본

JSX는 자바스크립트 안에서 HTML 태그를 사용할 수 있는 문법입니다.
class 대신 className, for 대신 htmlFor를 사용해야 합니다.

```
jsx
```

```
const element = <h1 className="title">Hello JSX</h1>;
```

2-2. JSX 표현식

JSX 안에서 중괄호 {}를 사용하면 자바스크립트 값을 출력할 수 있습니다. 숫자, 문자열, 함수 호출 결과 등을 넣을 수 있습니다.

```
jsx
```

```
const name = "React";  
const element = <p>나는 {name}를 배우고 있습니다.</p>;
```

2-2. JSX 주의사항

- 반드시 하나의 루트 요소만 반환해야 합니다.
- 여러 요소를 반환하려면 <div>로 감싸거나, 빈 태그 <> </>(Fragment)를 사용합니다.
- 조건문은 JSX 안에서 직접 쓸 수 없고, 삼항 연산자를 사용합니다.

```
jsx

return (
  <>
    <h1>첫 번째 요소</h1>
    <p>두 번째 요소</p>
  </>
);
```

2-2. Props 활용 예시

- Props를 활용하면 같은 컴포넌트를 다양한 방식으로 재사용할 수 있습니다.

```
function Button({ text, color }) {  
  return <button style={{ backgroundColor: color }}>{text}</button>;  
}
```

// 사용 예시

```
<Button text="저장" color="blue" />  
<Button text="삭제" color="red" />
```

2-2. Axios란?

Axios 는 HTTP 요청을 쉽게 보낼 수 있는 라이브러리입니다.
REST API와 연동할 때 자주 사용되며, fetch보다 간단한 문법을 제공합니다.

```
bash
```

```
npm install axios
```

2-2. Axios GET 요청

- GET 요청으로 서버에서 데이터를 가져올 수 있습니다.

```
import axios from "axios";

axios.get("https://jsonplaceholder.typicode.com/posts")
  .then(res => console.log(res.data));
```

2-2. Axios POST 요청

- POST 요청으로 서버에 데이터를 보낼 수 있습니다.

```
axios.post("https://jsonplaceholder.typicode.com/posts", {  
  title: "새 글",  
  body: "내용입니다",  
})  
.then(res => console.log(res.data));
```

2-2. 컴포넌트 트리 구조

React 앱은 트리 구조로 동작합니다.

최상위 App 컴포넌트가 있고, 그 아래에 여러 자식 컴포넌트가 배치됩니다.

데이터는 보통 부모에서 자식으로 전달됩니다.

jsx

```
function App() {  
  return (  
    <div>  
      <Header />  
      <Main />  
      <Footer />  
    </div>  
  );  
}
```

2-2. React DevTools 사용

React Developer Tools 는 크롬 확장 프로그램 입니다 .
브라우저에서 컴포넌트 구조를 트리 형태로 확인할 수 있으며,
props와 state 값을 실시간으로 살펴볼 수 있습니다.

2-2. 요약

- React는 컴포넌트 기반 UI 라이브러리입니다.
- Props와 State로 데이터를 관리하며, JSX 문법을 사용합니다.
- 조건부 렌더링과 리스트 렌더링으로 동적 UI를 구현할 수 있습니다.
- Axios를 사용하여 서버와 쉽게 통신할 수 있습니다.

2-3. CSS 기본 문법 소개

React에서도 일반 웹 개발과 동일하게 CSS를 사용할 수 있습니다. 선택자와 속성을 이용하여 HTML 요소에 스타일을 적용합니다.

코드에서는 h1 요소의 글자 색상을 파란색으로, 글자 크기를 24px로 지정했습니다.

```
CSS

h1 {
  color: blue;
  font-size: 24px;
  margin-bottom: 10px;
}
```

2-3. 인라인 스타일링

React에서는 `style` 속성에 객체 형태로 스타일을 적용할 수 있습니다. 속성 이름은 카멜케이스(camelCase)로 작성해야 합니다.

```
function InlineExample() {  
  return <h1 style={{ color: "green", fontSize: "20px" }}>인라인 스타일</h1>;  
}
```

2-3. 외부 CSS 파일 적용

컴포넌트와 같은 폴더에 .css 파일을 만들고 import하여 사용할 수 있습니다.

CSS

```
/* App.css */
.title {
  color: red;
  font-weight: bold;
}
```

jsx

```
// App.js
import './App.css';

function App() {
  return <h1 className="title">외부 CSS 적용</h1>;
}
```

2-3. CSS 모듈 소개

CSS 파일 이름을 컴포넌트명.module.css 형태로 작성하면, 클래스 이름이 자동으로 고유하게 변환됩니다.

프로젝트 규모가 커질수록 CSS 충돌을 방지할 수 있습니다.

다음은 CSS 모듈 예제입니다.

2-3. CSS 모듈 예제

css

```
/* Button.module.css */  
.btn {  
  background-color: blue;  
  color: white;  
  padding: 10px;  
  border-radius: 8px;  
}
```

jsx

```
import styles from "./Button.module.css";  
  
function Button() {  
  return <button className={styles.btn}>모듈 버튼</button>;  
}
```

2-3. Styled-components 소개

Styled-components는 CSS를 자바스크립트 파일 안에 작성할 수 있게 해주는 라이브러리입니다.

컴포넌트 단위로 스타일을 관리할 수 있어 유지보수성이 높습니다.

```
bash
```

```
npm install styled-components
```

2-3. Styled-components 기본 사용

jsx

```
import styled from "styled-components";

const Btn = styled.button`
  background: tomato;
  color: white;
  padding: 10px 15px;
  border-radius: 6px;
`;

function App() {
  return <Btn>Styled 버튼</Btn>;
}
```

2-3. Styled-components와 Props

Styled-components는 Props를 전달받아 조건부 스타일을 적용할 수 있습니다.

jsx

```
const Btn = styled.button`
  background: ${props => (props.primary ? "blue" : "gray")};
  color: white;
`;

<Btn primary>Primary</Btn>
<Btn>Default</Btn>
```

2-3. CSS-in-JS의 장점

- 컴포넌트 단위로 스타일을 관리할 수 있습니다.
- 클래스 이름 충돌을 피할 수 있습니다.
- JS 문법을 그대로 활용해 동적인 스타일링이 가능합니다.

2-3. 요약

- React 스타일링 방법은 크게 일반 CSS, CSS 모듈, Styled-components로 나눌 수 있습니다.
- CSS 모듈은 클래스 충돌을 방지하고, Styled-components는 JS와 CSS를 통합 관리할 수 있습니다.
- 프로젝트 성격에 따라 적합한 방법을 선택하는 것이 중요합니다.

2-4. React Hooks 개요

- Hooks는 함수형 컴포넌트에서 상태 관리와 생명주기 기능을 사용할 수 있도록 만든 기능입니다.

대표적으로 `useState`, `useEffect`, `useRef`, `useContext`가 있으며, 직접

커스텀 훅을 만들어 재사용할 수도 있습니다.

2-4. useState 소개

- useState는 컴포넌트 안에서 값을 저장하고 변경할 수 있게 해주는 훅입니다.

상태가 바뀌면 해당 컴포넌트가 다시 렌더링됩니다.

```
jsx

import { useState } from "react";

function Counter() {
  const [count, setCount] = useState(0);
  return (
    <button onClick={() => setCount(count + 1)}>
      현재 카운트: {count}
    </button>
  );
}
```

2-4. useState 배열·객체 상태 관리

- 배열이나 객체를 상태로 다룰 때는 불변성을 유지해야 합니다.

기존 값을 복사한 뒤 수정하여 새로운 객체를 반환합니다.

```
jsx
```

```
const [user, setUser] = useState({ name: "민재", age: 25 });

const updateAge = () => {
  setUser({ ...user, age: user.age + 1 });
};
```

2-4. useEffect 소개

- useEffect는 컴포넌트가 렌더링될 때 특정 동작을 수행하도록 합니다.

주로 API 호출, 이벤트 등록, 구독 등 사이드 이펙트 작업에 사용됩니다.

jsx

```
import { useEffect } from "react";

useEffect(() => {
  console.log("컴포넌트가 마운트되었습니다.");
}, []);
```

2-4. useEffect 의존성 배열

- 의존성 배열에 상태나 변수를 넣으면 해당 값이 바뀔 때만 실행됩니다.

빈 배열([])을 넣으면 처음 한 번만 실행됩니다.

jsx

```
useEffect(() => {  
  document.title = `현재 카운트: ${count}`;  
}, [count]);
```

2-4. useEffect 클린업 함수

- 컴포넌트가 사라질 때 실행해야 하는 정리 작업을 클린업 함수로 작성할 수 있습니다.

```
jsx

useEffect(() => {
  const timer = setInterval(() => console.log("실행 중"), 1000);
  return () => clearInterval(timer); // 언마운트 시 실행
}, []);
```

2-4. useRef 소개

- useRef는 특정 DOM 요소를 직접 제어하거나, 렌더링과 관계없는 값을 저장할 때 사용합니다.

```
import { useRef } from "react";

function InputFocus() {
  const inputRef = useRef();
  return (
    <>
      <input ref={inputRef} />
      <button onClick={() => inputRef.current.focus()}>포커스</button>
    </>
  );
}
```

2-4. useRef 값 저장하기

- useRef는 렌더링이 일어나도 값이 초기화되지 않습니다.

이 특징을 이용해 이전 상태를 저장하거나 특정 값을 기억할 수 있습니다.

```
jsx
```

```
const countRef = useRef(0);

const increase = () => {
  countRef.current += 1;
  console.log(countRef.current);
};
```

2-4. useContext 소개

- useContext는 전역 데이터를 컴포넌트 트리 전체에서 공유할 수 있게 합니다.
Props를 여러 단계 전달하지 않고도 하위 컴포넌트가 값을 사용할 수 있습니다.

```
jsx

const ThemeContext = React.createContext("light");

function App() {
  return (
    <ThemeContext.Provider value="dark">
      <Child />
    </ThemeContext.Provider>
  );
}
```

2-4. useContext 사용 예시

jsx

```
function Child() {  
  const theme = React.useContext(ThemeContext);  
  return <p>현재 테마: {theme}</p>;  
}
```

2-4. 커스텀 훅 소개

커스텀 훅은 여러 컴포넌트에서 반복되는 로직을 함수로 분리해 재사용하는 방법입니다.

반드시 `use`로 시작하는 이름을 사용해야 합니다.

다음은 예제입니다.

2-4. 커스텀 훅 예제 (useToggle)

jsx

```
function useToggle(initial = false) {  
  const [value, setValue] = React.useState(initial);  
  const toggle = () => setValue(!value);  
  return [value, toggle];  
}
```

// 사용 예시

```
const [isOpen, toggleOpen] = useToggle();
```

2-4. 커스텀 훅 예제 (useFetch)

jsx

```
function useFetch(url) {  
  const [data, setData] = React.useState(null);  
  
  React.useEffect(() => {  
    fetch(url)  
      .then(res => res.json())  
      .then(json => setData(json));  
  }, [url]);  
  
  return data;  
}
```

2-4. 주요 Hooks 정리

- `useState`: 컴포넌트 내부 상태 관리
- `useEffect`: 사이드 이펙트 처리
- `useRef`: DOM 제어 및 값 저장
- `useContext`: 전역 데이터 공유
- 커스텀 훅: 로직 재사용

2-4. 요약

- React Hooks는 함수형 컴포넌트에서 상태와 로직을 다룰 수 있게 해주는 핵심 기능입니다.

가장 자주 사용하는 `useState`와 `useEffect`를 먼저 익히고, 상황에 맞게 다른 훅들을 조합해 사용하는 것이 중요합니다.

2-5. 회원가입 화면 구현 (JSX)

회원가입 기능은 사용자가 아이디, 비밀번호, 이메일 등의 정보를 입력하고 서버에 전달하여 계정을 생성하는 과정입니다.

아래 코드는 회원가입 폼의 JSX 구조 예시입니다.

```
import { useState } from "react";

function SignUp() {
  const [form, setForm] = useState({ username: "", password: "", email: "" });

  const handleChange = (e) => {
    setForm({ ...form, [e.target.name]: e.target.value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log(form); // 실제로는 axios.post() 요청
  };

  return (
    <form onSubmit={handleSubmit} className="signup-form">
      <input type="text" name="username" placeholder="아이디" onChange={handleChange} />
      <input type="password" name="password" placeholder="비밀번호" onChange={handleChange} />
      <input type="email" name="email" placeholder="이메일" onChange={handleChange} />
      <button type="submit">회원가입</button>
    </form>
  );
}
```

2-5. 회원가입 화면 스타일 (CSS)

```
.signup-form {  
  display: flex;  
  flex-direction: column;  
  width: 300px;  
  margin: 0 auto;  
}  
  
.signup-form input {  
  margin: 8px 0;  
  padding: 10px;  
  border: 1px solid #ccc;  
  border-radius: 6px;  
}  
  
.signup-form button {  
  padding: 10px;  
  background-color: #4caf50;  
  color: white;  
  border: none;  
  border-radius: 6px;  
  cursor: pointer;  
}  
  
.signup-form button:hover {  
  background-color: #45a049;  
}
```

2-5. 로그인 화면 구현 (JSX)

로그인은 사용자가 입력한 아이디와 비밀번호를 서버에 전달하고, 서버에서 발급받은 JWT 토큰을 저장하는 과정입니다.

```
import { useState } from "react";

function Login() {
  const [form, setForm] = useState({ username: "", password: "" });

  const handleChange = (e) => {
    setForm({ ...form, [e.target.name]: e.target.value });
  };

  const handleSubmit = (e) => {
    e.preventDefault();
    console.log("로그인 요청:", form);
    // axios.post("/login", form).then(res => localStorage.setItem("token", res.data.token));
  };

  return (
    <form onSubmit={handleSubmit} className="login-form">
      <input type="text" name="username" placeholder="아이디" onChange={handleChange} />
      <input type="password" name="password" placeholder="비밀번호" onChange={handleChange} />
      <button type="submit">로그인</button>
    </form>
  );
}
```

2-5. 로그인 화면 스타일 (CSS)

```
.login-form {  
  display: flex;  
  flex-direction: column;  
  width: 300px;  
  margin: 50px auto;  
}  
  
.login-form input {  
  margin: 8px 0;  
  padding: 12px;  
  border: 1px solid #bbb;  
  border-radius: 5px;  
}  
  
.login-form button {  
  margin-top: 10px;  
  padding: 12px;  
  background: #2196f3;  
  color: white;  
  font-weight: bold;  
  border: none;  
  border-radius: 6px;  
}  
  
.login-form button:hover {  
  background: #1976d2;  
}
```

2-5. JWT 검증 (JSX)

JWT(JSON Web Token) 검증은 사용자가 로그인 후 받은 토큰이 유효한지 확인하는 과정입니다.

React에서는 서버로 요청을 보낼 때 Authorization 헤더에 토큰을 담아 전달합니다.

```
import axios from "axios";

function JwtCheck() {
  const checkToken = async () => {
    const token = localStorage.getItem("token");
    try {
      const res = await axios.get("/auth/verify", {
        headers: { Authorization: `Bearer ${token}` },
      });
      alert("토큰이 유효합니다.");
    } catch (e) {
      alert("토큰이 만료되었거나 잘못되었습니다.");
    }
  };

  return <button onClick={checkToken} className="jwt-btn">JWT 검증</button>;
}
```

2-5. JWT 검증 (CSS)

```
.jwt-btn {  
  padding: 10px 20px;  
  background: #673ab7;  
  color: white;  
  border: none;  
  border-radius: 6px;  
  cursor: pointer;  
}  
  
.jwt-btn:hover {  
  background: #512da8;  
}
```

2-5. 로그아웃 (JSX)

로그아웃은 저장된 토큰을 삭제하고 로그인 화면으로 리다이렉트하는 방식으로 구현합니다.

```
function Logout() {  
  const handleLogout = () => {  
    localStorage.removeItem("token");  
    alert("로그아웃 되었습니다.");  
    window.location.href = "/login";  
  };  
  
  return <button onClick={handleLogout} className="logout-btn">로그아웃</button>;  
}
```

2-5. 로그아웃 (CSS)

```
.logout-btn {  
  padding: 10px 20px;  
  background: #e53935;  
  color: white;  
  border: none;  
  border-radius: 6px;  
  cursor: pointer;  
}  
  
.logout-btn:hover {  
  background: #c62828;  
}
```

2-5. 토큰 재발급 (JSX)

토큰이 만료되었을 때, 서버에 리프레시 토큰을 보내 새로운 액세스 토큰을 발급받습니다.

```
import axios from "axios";

function RefreshToken() {
  const handleRefresh = async () => {
    try {
      const res = await axios.post("/auth/refresh", {
        refreshToken: localStorage.getItem("refreshToken"),
      });
      localStorage.setItem("token", res.data.accessToken);
      alert("토큰이 재발급되었습니다.");
    } catch {
      alert("재로그인이 필요합니다.");
    }
  };

  return <button onClick={handleRefresh} className="refresh-btn">토큰 재발급</button>;
}
```

2-5. 토큰 재발급 (CSS)

```
.refresh-btn {  
  padding: 10px 20px;  
  background: #009688;  
  color: white;  
  border: none;  
  border-radius: 6px;  
  cursor: pointer;  
}  
  
.refresh-btn:hover {  
  background: #00796b;  
}
```

2-5. 회원 정보 수정 (JSX)

회원 정보 수정은 사용자가 자신의 이름, 이메일 등을 바꿀 수 있도록 하는 기능입니다.

```
import { useState } from "react";
import axios from "axios";

function UpdateProfile() {
  const [form, setForm] = useState({ name: "", email: "" });

  const handleChange = (e) => {
    setForm({ ...form, [e.target.name]: e.target.value });
  };

  const handleSubmit = async (e) => {
    e.preventDefault();
    await axios.put("/users/me", form);
    alert("정보가 수정되었습니다.");
  };

  return (
    <form onSubmit={handleSubmit} className="update-form">
      <input type="text" name="name" placeholder="이름" onChange={handleChange} />
      <input type="email" name="email" placeholder="이메일" onChange={handleChange} />
      <button type="submit">수정하기</button>
    </form>
  );
}
```

2-5. 회원 정보 수정 (CSS)

```
.update-form {
  display: flex;
  flex-direction: column;
  width: 320px;
  margin: 0 auto;
}

.update-form input {
  margin: 6px 0;
  padding: 10px;
  border: 1px solid #aaa;
  border-radius: 6px;
}

.update-form button {
  padding: 12px;
  margin-top: 10px;
  background: #ff9800;
  border: none;
  color: white;
  font-weight: bold;
  border-radius: 6px;
  cursor: pointer;
}

.update-form button:hover {
  background: #f57c00;
}
```

2-5. 리스트 조회 (JSX)

리스트 조회는 서버에서 데이터를 가져와 화면에 반복적으로 출력하는 기능입니다.

```
import { useEffect, useState } from "react";
import axios from "axios";

function UserList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    axios.get("/users").then((res) => setUsers(res.data));
  }, []);

  return (
    <ul className="user-list">
      {users.map((user) => (
        <li key={user.id}>{user.name} ({user.email})</li>
      ))}
    </ul>
  );
}
```

2-5. 리스트 조회 (CSS)

```
.user-list {  
  list-style: none;  
  padding: 0;  
  margin: 20px auto;  
  width: 300px;  
}  
  
.user-list li {  
  padding: 10px;  
  border-bottom: 1px solid #ddd;  
}
```


2-5. 게시물 목록 조회 (JSX)

게시글 목록을 서버에서 가져와 리스트로 렌더링합니다. 로딩과 오류 상태를 분리하여 사용자에게 명확히 알립니다.

```
import { useEffect, useState } from "react";
import axios from "axios";

export default function PostList() {
  const [data, setData] = useState([]);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState("");

  useEffect(() => {
    const fetchPosts = async () => {
      try {
        setLoading(true);
        const res = await axios.get("/api/posts");
        setData(res.data);
      } catch (e) {
        setError("목록을 불러오지 못했습니다.");
      } finally {
        setLoading(false);
      }
    };
    fetchPosts();
  }, []);

  if (loading) return <p>불러오는 중입니다...</p>;
  if (error) return <p className="error">{error}</p>;

  return (
    <ul className="post-list">
      {data.map((p) => (
        <li key={p.id} className="post-item">
          <a href={`/${posts}/${p.id}`}>{p.title}</a>
          <span className="meta">{p.author} · {new Date(p.createdAt).toLocaleDateString()}</span>
        </li>
      ))}
    </ul>
  );
}
```

2-5. 게시물 목록 조회 (CSS)

다음은 게시물 목록 조회의 일부 예시입니다.

리스트 간격 및 정보 정리까지 다루어 가독성을 높이는 작업이 이루어집니다.

```
.post-list { list-style: none; padding: 0; margin: 24px auto; max-width: 720px; }
.post-item { padding: 12px 0; border-bottom: 1px solid #eee; display: flex; justify-content: space-between; align-items: baseline; gap: 16px; }
.post-item a { font-weight: 600; text-decoration: none; color: #222; }
.post-item a:hover { text-decoration: underline; }
.meta { color: #888; font-size: 12px; }
.error { color: #d32f2f; }
```

2-5. 카테고리별 목록 조회 (JSX)

쿼리스트링으로 카테고리를 받아 해당 글만 필터링합니다. 카테고리 변경 시 목록을 다시 불러옵니다.

```
import { useEffect, useState } from "react";
import { useSearchParams } from "react-router-dom";
import axios from "axios";

export default function CategoryList() {
  const [params] = useSearchParams();
  const category = params.get("category") || "all";
  const [items, setItems] = useState([]);

  useEffect(() => {
    axios.get("/api/posts", { params: { category } }).then((res) => setItems(res.data));
  }, [category]);

  return (
    <>
      <h2>카테고리: {category}</h2>
      <ul className="post-list">
        {items.map((p) => <li key={p.id} className="post-item"><a href={` /posts/${p.id}`}>{p.title}</a></li>)}
      </ul>
    </>
  );
}
```

2-5. 카테고리별 목록 조회 (CSS)

아래 코드는 현재 선택된 카테고리를 제목으로 강조합니다.

```
h2 { margin: 16px 0; font-size: 20px; }
```

2-5. 게시물 상세 조회 (JSX)

게시글 본문과 부가 정보를 표시합니다. 없는 글 접근 시 사용자에게 안내합니다.

```
import { useEffect, useState } from "react";
import { useParams } from "react-router-dom";
import axios from "axios";

export default function PostDetail() {
  const { id } = useParams();
  const [post, setPost] = useState(null);
  const [err, setErr] = useState("");

  useEffect(() => {
    axios.get(`/api/posts/${id}`)
      .then(res => setPost(res.data))
      .catch(() => setErr("게시글을 찾을 수 없습니다."));
  }, [id]);

  if (err) return <p className="error">{err}</p>;
  if (!post) return <p>불러오는 중입니다...</p>;

  return (
    <article className="post-detail">
      <h1>{post.title}</h1>
      <p className="meta">{post.author} · {new Date(post.createdAt).toLocaleString()}</p>
      <div className="content">{post.content}</div>
    </article>
  );
}
```

2-5. 게시물 상세 조회 (CSS)

본문 가독성을 높이고 제목과 메타 영역을 구분합니다.

```
.post-detail { max-width: 720px; margin: 24px auto; }  
.post-detail h1 { margin: 0 0 8px; font-size: 28px; }  
.post-detail .meta { color: #777; margin-bottom: 16px; }  
.post-detail .content { line-height: 1.75; white-space: pre-wrap; }
```

2-5. 게시물 작성 (JSX)

인증이 필요한 요청입니다. 토큰을 헤더에 포함하고 성공 후 상세 페이지로 이동합니다.

```
import { useState } from "react";
import axios from "axios";
import { useNavigate } from "react-router-dom";

export default function PostCreate() {
  const nav = useNavigate();
  const [form, setForm] = useState({ title: "", content: "" });
  const onChange = (e) => setForm({ ...form, [e.target.name]: e.target.value });

  const onSubmit = async (e) => {
    e.preventDefault();
    const token = localStorage.getItem("token");
    const { data } = await axios.post("/api/posts", form, {
      headers: { Authorization: `Bearer ${token}` },
    });
    nav(`/posts/${data.id}`);
  };

  return (
    <form onSubmit={onSubmit} className="post-form">
      <input name="title" placeholder="제목" value={form.title} onChange={onChange} />
      <textarea name="content" placeholder="내용" value={form.content} onChange={onChange} rows={10} />
      <button type="submit">작성하기</button>
    </form>
  );
}
```

2-5. 게시물 작성 (CSS)

입력 영역 크기를 충분히 주고 버튼을 고정 높이로 통일합니다.

```
.post-form { max-width: 720px; margin: 24px auto; display: grid; gap: 12px; }  
.post-form input, .post-form textarea { padding: 12px; border: 1px solid #ccc; border-radius: 8px; }  
.post-form button { height: 44px; background: #1976d2; color: #fff; border: 0; border-radius: 8px; cursor: pointer; }  
.post-form button:hover { background: #1565c0; }
```

2-5. 게시물 수정 (JSX)

기존 값을 불러와 폼에 채우고 저장합니다. 백엔드가 PUT 또는 PATCH를 사용하는지 확인하여 메서드를 맞춥니다.

```
import { useEffect, useState } from "react";
import { useParams, useNavigate } from "react-router-dom";
import axios from "axios";

export default function PostEdit() {
  const { id } = useParams();
  const nav = useNavigate();
  const [form, setForm] = useState({ title: "", content: "" });

  useEffect(() => {
    axios.get(`/api/posts/${id}`).then(res => setForm({ title: res.data.title, content: res.data.content }));
  }, [id]);

  const onChange = (e) => setForm({ ...form, [e.target.name]: e.target.value });
  const onSubmit = async (e) => {
    e.preventDefault();
    const token = localStorage.getItem("token");
    await axios.put(`/api/posts/${id}`, form, { headers: { Authorization: `Bearer ${token}` } });
    nav(`/posts/${id}`);
  };

  return (
    <form onSubmit={onSubmit} className="post-form">
      <input name="title" value={form.title} onChange={onChange} />
      <textarea name="content" rows={10} value={form.content} onChange={onChange} />
      <button type="submit">수정하기</button>
    </form>
  );
}
```

2-5. 게시물 수정 (CSS)

입력 영역 크기를 충분히 주고 버튼을 고정 높이로 통일합니다.

게시글 작성과 동일합니다.

```
.post-form { max-width: 720px; margin: 24px auto; display: grid; gap: 12px; }  
.post-form input, .post-form textarea { padding: 12px; border: 1px solid #ccc; border-radius: 8px; }  
.post-form button { height: 44px; background: #1976d2; color: #fff; border: 0; border-radius: 8px; cursor: pointer; }  
.post-form button:hover { background: #1565c0; }
```

2-5. 게시물 삭제 (JSX)

삭제는 사용자 확인 후 진행합니다. 성공 시 목록으로 이동합니다.

```
import axios from "axios";
import { useNavigate, useParams } from "react-router-dom";

export default function PostDeleteButton() {
  const { id } = useParams();
  const nav = useNavigate();

  const onDelete = async () => {
    if (!window.confirm("정말 삭제하시겠습니까?")) return;
    const token = localStorage.getItem("token");
    await axios.delete(`/api/posts/${id}`, { headers: { Authorization: `Bearer ${token}` } });
    nav("/");
  };

  return <button className="danger" onClick={onDelete}>삭제하기</button>;
}
```

2-5. 게시물 삭제 (CSS)

위험 동작은 명확한 색상으로 표시합니다.

```
.danger { padding: 10px 16px; background: #e53935; color: #fff; border: 0; border-radius: 8px; cursor: pointer; }  
.danger:hover { background: #c62828; }
```

2-5. 좋아요 수/상태 별도 조회 (JSX)

상태와 수치를 분리해 사용할 수 있습니다. 다른 컴포넌트에서도 재사용하기
좋습니다.

```
import { useEffect, useState } from "react";
import axios from "axios";

export function useLikeInfo(postId) {
  const [info, setInfo] = useState({ liked: false, count: 0 });
  useEffect(() => {
    Promise.all([
      axios.get(`/api/posts/${postId}/likes/count`),
      axios.get(`/api/posts/${postId}/likes/me`)
    ]).then(([c, m]) => setInfo({ count: c.data.count, liked: m.data.liked }));
  }, [postId]);
  return info;
}
```

2-5. 좋아요 수/상태 표시 (CSS)

작은 배지 형태로 재사용합니다.

```
.badge { font-size: 12px; padding: 2px 8px; border-radius: 10px; background: ■ #f5f5f5; color: □ #444; }
```

2-5. 댓글 작성 (JSX)

댓글 입력 후 서버에 저장합니다. 성공 시 입력값을 초기화하고 목록을 갱신합니다.

```
import { useState } from "react";
import axios from "axios";

export default function CommentForm({ postId, onCreate }) {
  const [text, setText] = useState("");

  const submit = async (e) => {
    e.preventDefault();
    const token = localStorage.getItem("token");
    await axios.post(`/api/posts/${postId}/comments`, { content: text }, {
      headers: { Authorization: `Bearer ${token}` },
    });
    setText("");
    onCreate?.();
  };

  return (
    <form onSubmit={submit} className="comment-form">
      <textarea value={text} onChange={(e) => setText(e.target.value)} placeholder="댓글을 입력하세요" rows={3}/>
      <button type="submit">등록</button>
    </form>
  );
}
```

2-5. 댓글 작성 (CSS)

입력 영역은 최소 높이를 유지하고 버튼을 우측 정렬한 코드입니다.

```
.comment-form { display: grid; gap: 8px; max-width: 720px; margin: 12px auto; }  
.comment-form textarea { resize: vertical; padding: 10px; border: 1px solid #ccc; border-radius: 6px; }  
.comment-form button { justify-self: end; height: 36px; padding: 0 16px; background: #4caf50; color: #fff; border: 0; border-radius: 6px; }  
.comment-form button:hover { background: #43a047; }
```

2-5. 댓글 목록 조회 (JSX)

게시글 상세 화면에서 댓글을 함께 출력합니다. 최신 순 정렬을 적용합니다.

```
import { useEffect, useState } from "react";
import axios from "axios";

export default function CommentList({ postId }) {
  const [list, setList] = useState([]);

  const load = () => axios.get(`/api/posts/${postId}/comments`)
    .then((res) => setList(res.data));

  useEffect(() => { load(); }, [postId]);

  return (
    <ul className="comment-list">
      {list.map(c => (
        <li key={c.id} className="comment-item">
          <b>{c.author}</b>
          <span className="date">{new Date(c.createdAt).toLocaleString()}</span>
          <p>{c.content}</p>
        </li>
      ))}
    </ul>
  );
}
```

2-5. 댓글 수정 (JSX)

자신의 댓글만 수정하도록 합니다. 간단한 인라인 편집 UI를 제공합니다.

```
import { useState } from "react";
import axios from "axios";

export function CommentEditable({ comment, onUpdated }) {
  const [edit, setEdit] = useState(false);
  const [text, setText] = useState(comment.content);

  const save = async () => {
    const token = localStorage.getItem("token");
    await axios.put(`/api/comments/${comment.id}`, { content: text }, {
      headers: { Authorization: `Bearer ${token}` },
    });
    setEdit(false);
    onUpdated?.();
  };

  return (
    <div className="comment-edit">
      {edit ? (
        <>
          <textarea value={text} onChange={(e) => setText(e.target.value)} rows={3} />
          <div className="row">
            <button onClick={save}>저장</button>
            <button onClick={() => setEdit(false)} className="ghost">취소</button>
          </div>
        </>
      ) : (
        <>
          <p>{comment.content}</p>
          <button onClick={() => setEdit(true)}>수정</button>
        </>
      )}
    </div>
  );
}
```

2-5. 댓글 삭제 (JSX)

삭제 전 확인을 요청하고, 성공 시 목록을 새로고침합니다..

```
import axios from "axios";

export function CommentDelete({ id, onDelete }) {
  const remove = async () => {
    if (!window.confirm("댓글을 삭제하시겠습니까?")) return;
    const token = localStorage.getItem("token");
    await axios.delete(`/api/comments/${id}`, { headers: { Authorization:
      onDelete?.() };
    };
    return <button className="danger" onClick={remove}>삭제</button>;
  }
}
```

2-5. 댓글 좋아요 토글 (JSX)

댓글 단위로 좋아요 상태를 전환하고 카운트를 업데이트합니다.

```
import { useState, useEffect } from "react";
import axios from "axios";

export default function CommentLike({ commentId }) {
  const [liked, setLiked] = useState(false);
  const [count, setCount] = useState(0);

  useEffect(() => {
    axios.get(`/api/comments/${commentId}/likes/count`).then(r => setCount(r.data.count));
    axios.get(`/api/comments/${commentId}/likes/me`).then(r => setLiked(r.data.liked));
  }, [commentId]);

  const toggle = async () => {
    const token = localStorage.getItem("token");
    const { data } = await axios.post(`/api/comments/${commentId}/likes/toggle`, {}, {
      headers: { Authorization: `Bearer ${token}` },
    });
    setLiked(data.liked);
    setCount(c => data.liked ? c + 1 : Math.max(0, c - 1));
  };

  return <button className={`like-btn ${liked ? "on" : ""}`} onClick={toggle}>❤️ {count}</button>;
}
```

2-5. 좋아요 수 별도 표시 (JSX)

댓글 영역 우측에 작은 배치로 표시한 코드입니다.

```
export function LikeBadge({ count }) {  
  return <span className="badge">♥ {count}</span>;  
}
```

2-5. 게시물 핀 고정 (JSX, CSS)

관리자 또는 권한 사용자만 상단 고정합니다. 성공 시 목록을 새로고침합니다.

```
import axios from "axios";

export function PinPost({ postId, onPinned }) {
  const pin = async () => {
    const token = localStorage.getItem("token");
    await axios.post(`/api/posts/${postId}/pin`, {}, { headers: { Authorization: `Bearer ${token}` } });
    onPinned?.();
  };
  return <button className="pin-btn" onClick={pin}>상단 고정</button>;
}
```

```
.pin-btn { padding: 8px 12px; border: 1px dashed #1976d2; color: #1976d2; background: #e3f2fd; border-radius: 8px; cursor: pointer; }
```

2-5. 게시물 핀 해제 (JSX, CSS)

이미 고정된 게시글을 목록에서 일반 글로 되돌립니다.

```
import axios from "axios";

export function UnpinPost({ postId, onUnpinned }) {
  const unpin = async () => {
    const token = localStorage.getItem("token");
    await axios.delete(`/api/posts/${postId}/pin`, { headers: { Authorization: `Bearer ${token}` }});
    onUnpinned?.();
  };
  return <button className="pin-off" onClick={unpin}>고정 해제</button>;
}
```

```
.pin-off { padding: 8px 12px; background: #fafafa; border: 1px solid #bbb; border-radius: 8px; }
```

2-5. 상단 고정 목록 조회 (JSX, CSS)

상단 고정된 게시글만 별도 영역에 표시합니다.

```
import { useEffect, useState } from "react";
import axios from "axios";

export default function PinnedList() {
  const [pins, setPins] = useState([]);

  useEffect(() => {
    axios.get("/api/posts/pinned").then(res => setPins(res.data));
  }, []);

  if (!pins.length) return null;

  return (
    <section className="pinned">
      <h3>📌 상단 고정</h3>
      <ul>
        {pins.map(p => <li key={p.id}><a href={`/${posts}/${p.id}`}>{p.title}</a></li>)}
      </ul>
    </section>
  );
}
```

```
.pinned { max-width: 720px; margin: 16px auto 24px; padding: 12px 16px; background: #fff8e1; border: 1px solid #ffe082; border-radius: 8px; }
.pinned h3 { margin: 0 0 8px; font-size: 16px; }
.pinned ul { list-style: none; padding: 0; margin: 0; display: grid; gap: 6px; }
.pinned a { color: #333; text-decoration: none; }
.pinned a:hover { text-decoration: underline; }
```


제 3 장

SPRING BOOT

- 1) 들어가며
- 2) 개발환경 세팅
- 3) 핵심 메커니즘
- 4) REST API 설계
- 5) 데이터 액세스와 트랜잭션 관리
- 6) 인증/인가
- 7) 설정 전략
- 8) 관측성
- 9) 테스트
- 10) 패키징/배포
- 11) 품질·성능·보안 체크리스트
- 12) 미니 프로젝트: 게시판 API 설계·구현·테스트

3-1. 들어가며: 스프링 부트가 푸는 문제

웹 개발 과정에서의 구조적 난점

웹 개발 과정에서 개발자는 코드 작성보다 환경 설정과 운영 환경 대응에 더 많은 시간을 소비한다. 새로운 기능 구현보다 빌드 오류, 라이브러리 버전 충돌, 배포 스크립트 작성 같은 작업이 더 큰 부담으로 다가온다. 이러한 보이지 않는 비용이 프로젝트의 생산성을 저해한다.

- **설정 지옥(Configuration Hell)**: 프로젝트마다 프레임워크와 서버 설정 방식이 달라 표준화가 어렵다.
- **환경 불일치(Environment Drift)**: 개발, 테스트, 운영 환경이 다르게 구성되어 장애가 발생한다.
- **운영 가시성 부족(Observability Gap)**: 로그와 메트릭이 흩어져 있어 문제 해결이 지연된다.
- **보안 기본선 부재(Security Baseline Missing)**: 인증과 인가 기능을 프로젝트마다 새롭게 구현해야



스프링 부트의 해법 – 구성의 단순화와 실행 환경의 일원화

스프링 부트는 단순한 프레임워크가 아니라 플랫폼으로 기능한다. 개발과 운영의 간극을 줄이는 것을 목표로 하며, 합리적인 기본값을 제공하여 개발자가 본질적인 기능 구현에 집중할 수 있게 한다.

- **Starter**: 호환성이 검증된 의존성 모음을 제공한다.
- **Auto-Configuration**: 클래스패스와 설정을 감지하여 자동으로 Bean을 구성한다.
- **임베디드 서버**: Tomcat이나 Jetty를 내장하여 `java -jar` 명령만으로 애플리케이션을 실행할 수 있다.
- **외부화된 설정**: `application.yml`, 환경 변수, 프로파일을 통해 설정을 분리한다.
- **운영 가시성**: Actuator와 Micrometer로 운영 지표와 상태를 즉시 확인할 수 있다.



전통적 스프링 VS 스프링 부트

항목	전통적 스프링	스프링 부트
웹 서버	외부 Tomcat + WAR 배포	내장 Tomcat + 실행형 JAR
의존성 관리	수동으로 버전 호환성 확인	Starter/BOM 제공
설정	XML, 자바 외부 conf 분산	<code>application.yml</code> 로 일원화
가시성	별도의 도구 설치 필요	Actuator와 Micrometer 내장
보안	프로젝트마다 개별 구현	Spring Security 기본 제공

이 장에서 다룰 여정

1. 개발환경 세팅: Initializr, Gradle, 프로파일 구분.
2. API 설계: DTO, Validation, 표준 에러 응답.
3. 데이터 액세스: JPA, 트랜잭션, N+1 문제 해결.
4. 보안: JWT 인증 및 인가.
5. 운영: Actuator와 Micrometer를 통한 관측성.
6. 배포: Docker와 Jib을 활용한 패키징.

주의점

- Auto-Configuration이 생성하는 Bean을 이해하지 못하면 충돌이 발생한다.
- JPA는 편리하지만 성능 문제 해결은 여전히 개발자의 몫이다.
- 운영 환경은 반드시 개발/테스트/운영으로 분리해야 한다.
- 보안은 기본 제공 기능 위에 별도의 정책과 검증을 더해야 한다.

3-2. 개발환경 세팅(IntelliJ, Initializr, 프로파일 구분)

스프링 부트를 학습하기 위해서는 먼저 표준화된 개발환경을 마련해야 한다. 환경이 일관되지 않으면 빌드 재현성이 떨어지고, 운영 환경과 개발 환경의 차이로 인해 예기치 못한 오류가 발생한다. 따라서 JDK, 빌드 도구, 프로젝트 초기화 도구를 명확히 지정하고, 환경별 설정 분리를 습관화하는 것이 중요하다.

IDE - 인텔리제이 vs 이클립스

- JetBrains 공식 웹사이트에서 IntelliJ IDEA Community(무료) 또는 Ultimate(유료, 강력한 스프링 부트 지원) 버전을 설치한다.
- 설치 후 JDK 21을 IntelliJ에 등록하고, 프로젝트 기본 SDK로 지정한다.
- Gradle은 IntelliJ에서 자동으로 통합되어 관리된다.

참고: 자바를 별도로 설치하지 않아도 된다. 인텔리제이 안에서 자바 설치도 함께 진행

인텔리제이(IntelliJ) 설치하기

- 다운로드 링크: <https://www.jetbrains.com/ko-kr/idea/download/?section=mac>
- IntelliJ IDEA Community Edition (무료 버전)
 - OS 선택: Windows, macOS, Linux
 - Windows: .exe 선택

Windows macOS Linux

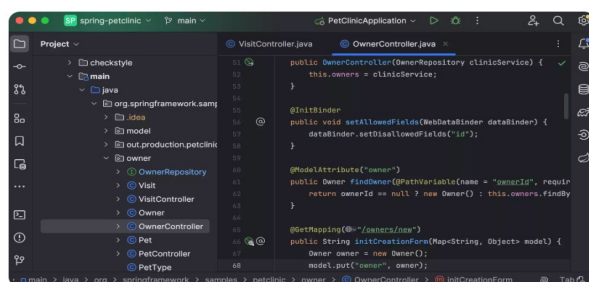


IntelliJ IDEA Ultimate

전문적 Java 및 Kotlin 개발을 위한 선도적 IDE

다운로드

.dmg (Apple Silicon) ▼



버전: 2025.2
빌드: 252.23892.409
2025년 8월 4일

시스템 요구 사항
설치 안내

기타 버전
타사 소프트웨어

145

Spring Initializr 프로젝트 생성

- 링크: <https://start.spring.io/>
- 화면에서 다음과 같이 선택한다.
 - **Project:** Gradle - Groovy
 - **Language:** Java
 - **Spring Boot:** 안정화된 최신 버전 (예: 3.5.5)
 - **Metadata:** Group = `com.example`, Artifact = `demo`
 - **Packaging:** Jar
 - **Java Version:** 17 또는 21 (교재에서는 17 기준)

Project		Language	
☒ Gradle - Groovy		☒ Java	☐ Kotlin ☐ Groovy
☐ Gradle - Kotlin	☐ Maven		
Spring Boot			
☐ 4.0.0 (SNAPSHOT)	☐ 4.0.0 (M2)	☐ 3.5.6 (SNAPSHOT)	☒ 3.5.5
☐ 3.4.10 (SNAPSHOT)	☐ 3.4.9		
Project Metadata			
Group	com.example		
Artifact	demo		
Name	demo		
Description	Demo project for Spring Boot		
Package name	com.example.demo		
Packaging	☒ Jar	☐ War	
Java	☐ 24	☐ 21	☒ 17

의존성 선택

초기 프로젝트를 학습 목적에 맞게 구성하기 위해 다음을 추가한다.

- Spring Web: REST API 개발용
- Validation: 입력 검증
- Spring Data JPA: 데이터 액세스
- MySQL Driver: DB 연결
- Spring Security: 보안
- Actuator: 관측성

Dependencies ADD ...

Spring Web WEB	Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.	—
Validation I/O	Bean Validation with Hibernate validator.	—
Spring Data JPA SQL	Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.	—
MySQL Driver SQL	MySQL JDBC driver.	—
Spring Security SECURITY	Highly customizable authentication and access-control framework for Spring applications.	—
Spring Boot Actuator OPS	Supports built in (or custom) endpoints that let you monitor and manage your application - such as application health, metrics, sessions, etc.	—

GENERATE EXPLORE ...

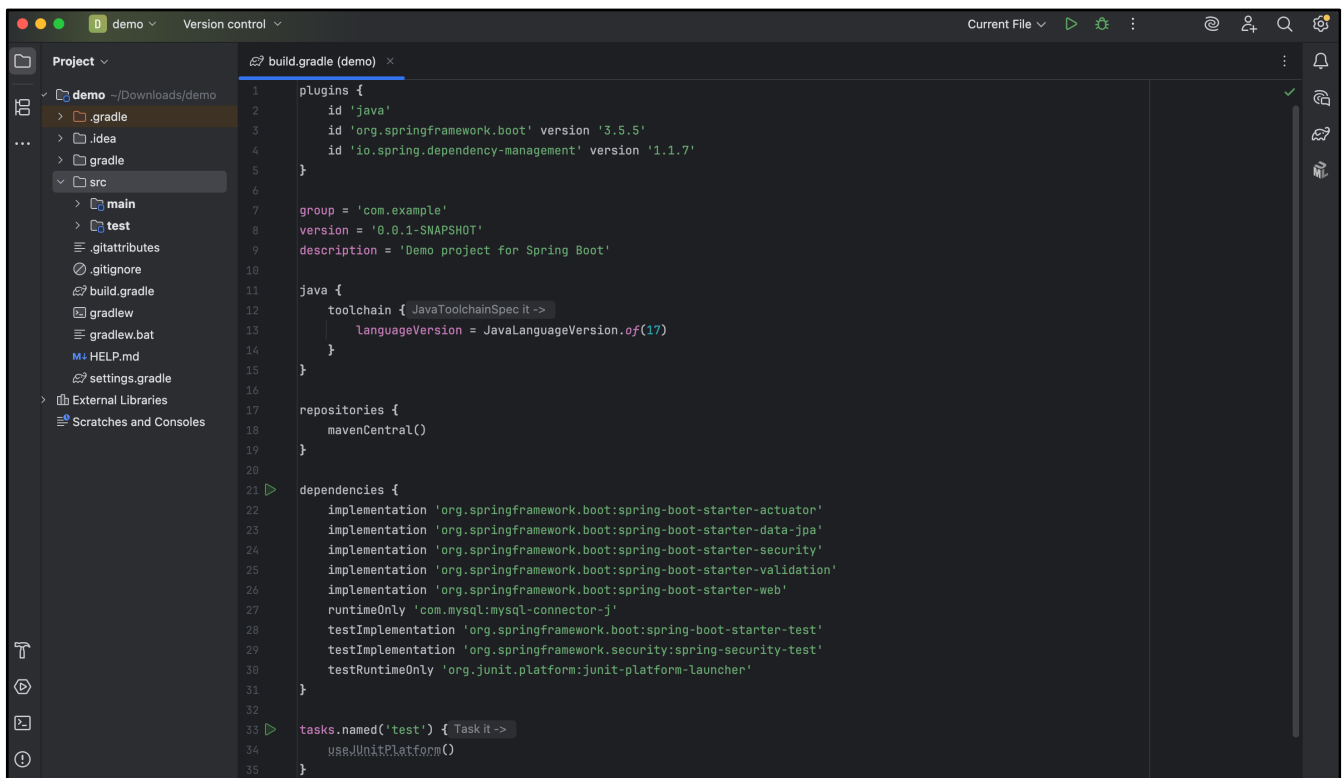
IntelliJ에서 프로젝트 열기

1. 다운로드한 ZIP 파일을 압축 해제한다.
2. IntelliJ IDEA → Open → 프로젝트 루트 폴더 선택.
3. Gradle 동기화가 자동으로 실행된다. (하단 Progress 확인)
4. SDK를 JDK 17(또는 21)로 설정한다.

프로파일 구분

환경별 설정은 application.yml에서 분리한다.

- 기본: application.yml (공통 설정)
- 개발: application-dev.yml
- 운영: application-prod.yml
- IntelliJ 실행 설정에서 --spring.profiles.active=dev 옵션을 부여한다.

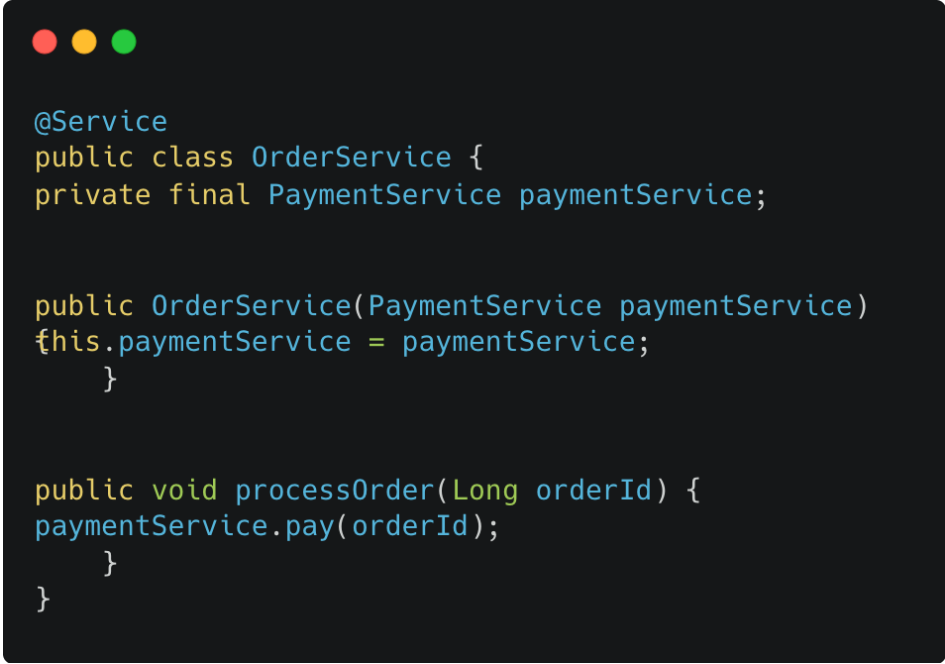


3-3. 핵심 메커니즘(IOC/DI, Auto-Config, Profiles)

제어의 역전(IOC)과 의존성 주입(DI)

스프링 프레임워크의 핵심은 제어의 역전(Inversion of Control, IOC)이다. 전통적인 자바 애플리케이션에서는 객체 생성과 생명주기 관리가 개발자의 코드에 직접 포함된다. 그러나 스프링에서는 이 제어 권한을 프레임워크가 가져가며, 개발자는 필요한 객체를 요청(주입)받아 사용한다.

- IOC 컨테이너: 애플리케이션 구동 시 Bean을 생성하고 관리한다.
- 의존성 주입(DI): 객체 간 결합도를 낮추고, 테스트와 유지보수를 용이하게 한다.



```
@Service
public class OrderService {
    private final PaymentService paymentService;

    public OrderService(PaymentService paymentService) {
        this.paymentService = paymentService;
    }

    public void processOrder(Long orderId) {
        paymentService.pay(orderId);
    }
}
```

위 코드에서 `OrderService`는 `PaymentService` 구현체를 직접 생성하지 않고, 프레임워크가 주입해 준다.

자동 설정(Auto-Configuration)

스프링 부트는 **Auto-Configuration**을 통해 클래스패스와 환경 설정을 분석하고, 적절한 Bean을 자동으로 등록한다. 예를 들어, 클래스패스에 **spring-boot-starter-web**이 있으면 **DispatcherServlet**과 관련 빈들이 자동으로 등록된다.

- **@SpringBootApplication**은 내부적으로 **@EnableAutoConfiguration**을 포함한다.
- 자동 설정은 **spring.factories** 또는 **spring-autoconfigure-metadata.properties**에 정의된 규칙에 따라 실행된다.
- 불필요하거나 충돌하는 설정은 **@ConditionalOnMissingBean** 등의 조건을 통해 조정된다.

예시: 데이터 소스 자동 설정

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/demo
    username: root
    password: root
```

위 설정만 제공하면, 스프링 부트가 자동으로 **DataSource** Bean을 생성한다.

프로파일(Profiles)

현실적인 애플리케이션은 개발/테스트/운영 환경에서 서로 다른 설정을 요구한다. 스프링 부트는 Profiles 기능을 통해 환경별 설정을 분리할 수 있게 한다.

- `application.yml`에 여러 프로파일을 정의한다.

```
spring:
  config:
    activate:
      on-profile: dev
  datasource:
    url: jdbc:mysql://localhost:3306/devdb
    username: dev
    password: dev123
---
spring:
  config:
    activate:
      on-profile: prod
  datasource:
    url: jdbc:mysql://db.prod.example:3306/proddb
    username: admin
    password: securepwd
```

- 실행 시 `--spring.profiles.active=dev` 옵션을 부여하면 개발 환경 설정이 적용된다.
- IntelliJ 실행 환경에서 VM Options에 추가하거나, 환경 변수로 지정할 수 있다.

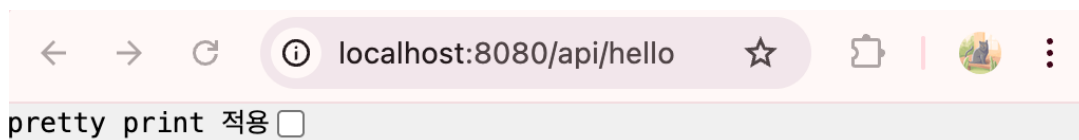
HelloController 예시

스프링 부트의 동작을 확인하기 위해 간단한 REST 컨트롤러를 작성한다.

```
@RestController
@RequestMapping("/api/hello")
public class HelloController {
    @GetMapping
    public Map<String, String> hello() {
        return Map.of("message", "Spring Boot is working");
    }
}
```

위 클래스를 추가한 뒤 애플리케이션을 실행하고,

<http://localhost:8080/api/hello>에 접속하면 JSON 응답을 확인할 수 있다.



```
{"message": "Spring Boot is working"}
```

세 가지

IOC/DI는 객체 관리의 기본 원칙이다.

- Auto-Configuration은 IOC 컨테이너가 제공할 Bean을 합리적으로 자동 등록한다.
- Profiles는 동일한 구조를 유지하면서 환경별 차이를 유연하게 반영한다.

세 가지가 결합되면, 개발자는 최소한의 설정만으로 환경별 일관된 애플리케이션을 손쉽게 구축할 수 있다.

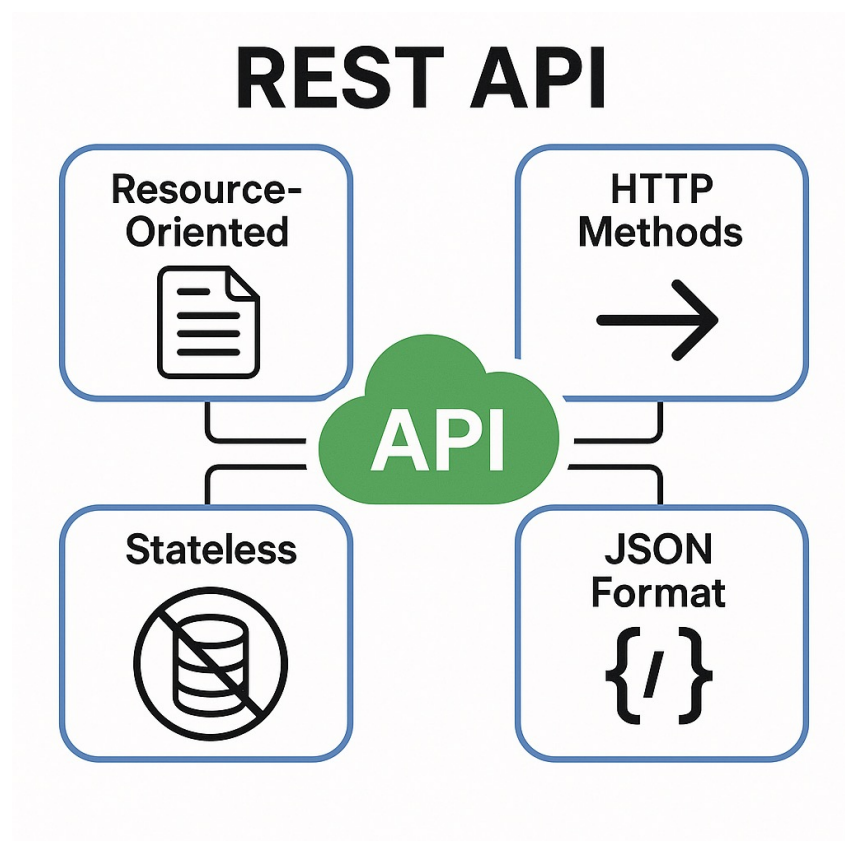
3-4. REST API 설계(레이어드, DTO/Validation)

REST API란 무엇인가?

REST API는 Representational State Transfer 아키텍처 스타일을 기반으로 하는 웹 API이다. 클라이언트와 서버가 HTTP 프로토콜을 통해 리소스(데이터)에 접근하는 방식을 정의한다.

- 리소스 지향적: URL은 리소스를 표현한다. 예: /api/posts/1
- 표준 메서드 활용: HTTP 메서드(GET, POST, PUT, DELETE)를 의미에 맞게 사용한다.
- 무상태성(Stateless): 서버는 클라이언트의 상태를 세션에 저장하지 않는다.
- 표현(Representation): 리소스는 JSON, XML 등 다양한 표현 형식으로 전달될 수 있다. 일반적으로 JSON을 사용한다.

예를 들어, /api/posts에 대한 GET 요청은 게시글 목록을 반환하고, POST 요청은 새로운 게시글을 생성한다.



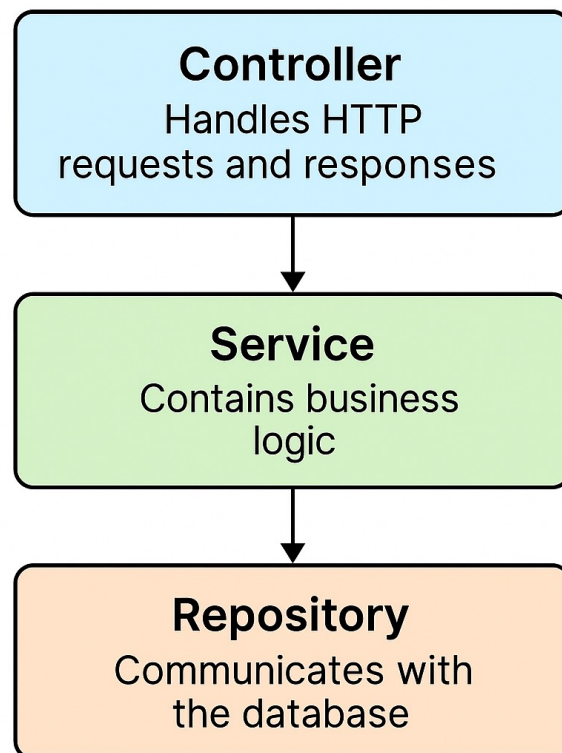
레이어드 아키텍처 개요

스프링 부트 애플리케이션은 보통 Controller → Service → Repository 계층으로 나뉜다.

- **Controller**: HTTP 요청과 응답을 처리한다.
- **Service**: 비즈니스 로직을 수행한다.
- **Repository**: 데이터베이스와 직접 통신한다.

이 계층 분리를 통해 코드의 책임이 명확해지고 테스트 용이성이 확보된다.

Layered Architecture Overview



엔티티와 DTO의 분리

1) 엔티티(Entity)의 역할

엔티티는 데이터베이스 테이블과 직접 매핑되는 클래스이다.

- @Entity 애너테이션으로 정의한다.
- 컬럼 제약 조건, 관계 매핑, 영속성 관리와 같은 DB 관련 속성을 포함한다.
- 외부 API와 직접적으로 교환하지 않고, 내부 비즈니스 로직과 데이터 저장소 사이의 다리 역할을 한다.

```
@Entity
@Table(name = "posts")
public class Post {
    @Id @GeneratedValue(strategy =
    GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false, length = 120)
    private String title;

    @Column(nullable = false, columnDefinition =
    "TEXT")
    private String content;

    // Getter/Setter 생략
}
```

엔티티와 DTO의 분리

2) DTO(Data Transfer Object)의 역할

DTO는 계층 간 데이터 전달을 위한 객체이다.

- 엔티티를 외부로 그대로 노출하지 않고, 필요한 필드만 추려낸다.
- 보안: 민감한 컬럼(비밀번호, 내부 식별자 등)을 숨길 수 있다.
- 유연성: API 요구사항에 맞는 구조를 설계할 수 있다.
- 검증: DTO에 `@NotBlank`, `@Size` 등의 애너테이션을 부여해 입력 유효성을 보장한다.

```
public record PostCreateRequest(
    @NotBlank @Size(max = 120) String title,
    @NotBlank String content
) {}

public record PostResponse(Long id, String title,
String content) {
    public static PostResponse from(Post post) {
        return new PostResponse(post.getId(),
post.getTitle(), post.getContent());
    }
}
```


엔티티와 DTO의 분리

3) 변환 과정 예시 Service & Controller

서비스 계층에서 엔티티 ↔ DTO 변환을 수행하고, 컨트롤러 계층은 이를 클라이언트에 반환한다.

```
@Service
public class PostService {
    private final PostRepository postRepository;

    // 생성자 주입을 통한 Repository 의존성 주입
    public PostService(PostRepository postRepository) {
        this.postRepository = postRepository;
    }

    // 게시물 생성: DTO(PostCreateRequest) → 엔티티(Post) → 저장 후
    // DTO(PostResponse) 반환
    public PostResponse create(PostCreateRequest request) {
        // 1. DTO를 엔티티로 변환
        Post post = new Post();
        post.setTitle(request.title());
        post.setContent(request.content());

        // 2. Repository를 통해 DB에 저장
        Post saved = postRepository.save(post);

        // 3. 엔티티를 응답용 DTO로 변환 후 반환
        return PostResponse.from(saved);
    }

    // 게시물 목록 조회: 엔티티 리스트 → DTO 리스트 변환
    public List<PostResponse> findAll() {
        return postRepository.findAll()
            .stream()
            .map(PostResponse::from) // 각 엔티티를 DTO로 변환
            .toList();
    }
}
```

- create() 메서드는 입력 **DTO** → **엔티티** → **저장** → **응답 DTO**라는 흐름을 따른다.
- findAll() 메서드는 DB에서 가져온 **엔티티 리스트**를 **DTO 리스트**로 변환한다.

엔티티와 DTO의 분리

3) 변환 과정 예시 Service & Controller

```
@RestController
@RequestMapping("/api/posts")
public class PostController {
    private final PostService postService;

    // 생성자 주입으로 서비스 의존성 주입
    public PostController(PostService postService) {
        this.postService = postService;
    }

    // 게시물 생성 API
    @PostMapping
    public ResponseEntity<PostResponse> create(@Valid @RequestBody
    PostCreateRequest request) {
        // Service 호출 → DTO 반환
        PostResponse response = postService.create(request);
        // HTTP 상태코드 201(Created)와 함께 결과 반환
        return
        ResponseEntity.status(HttpStatus.CREATED).body(response);
    }

    // 게시물 목록 조회 API
    @GetMapping
    public List<PostResponse> list() {
        // Service 호출 결과를 그대로 반환
        return postService.findAll();
    }
}
```

- `@RestController`는 JSON 응답을 반환하는 컨트롤러임을 의미한다.
- `@PostMapping`은 **POST** 요청을 처리하고, 입력 DTO를 `@RequestBody`로 받는다.
- `@Valid`는 DTO의 유효성 검증을 수행한다.
- `ResponseEntity`를 사용해 HTTP 상태 코드와 응답 본문을 함께 반환한다.
- `@GetMapping`은 **GET** 요청을 처리하고, 게시물 리스트를 반환한다.

입력 검증과 예외 처리

1) 입력 검증

- `@Valid` 애너테이션으로 DTO의 제약 조건을 검사한다.
- 검증 실패 시 `MethodArgumentNotValidException`이 발생한다.

```
public record UserSignupRequest(  
    @NotBlank @Size(min = 3, max = 20) String username,  
    @Email String email,  
    @NotBlank @Size(min = 8) String password  
) {}
```

2) 예외 처리 ControllerAdvice

전역 예외 처리기를 사용하면 일관된 에러 응답을 반환할 수 있다.

```
@RestControllerAdvice  
public class GlobalExceptionHandler {  
    @ExceptionHandler(MethodArgumentNotValidException.class)  
    public ResponseEntity<Map<String, String>>  
    handleValidationErrors(MethodArgumentNotValidException ex) {  
        Map<String, String> errors = new HashMap<>();  
        ex.getBindingResult().getFieldErrors().forEach(err ->  
            errors.put(err.getField(), err.getDefaultMessage()));  
        );  
        return ResponseEntity.badRequest().body(errors);  
    }  
}
```

표준 에러 응답 구조

실무에서는 클라이언트가 에러를 일관되게 처리할 수 있도록 표준 에러 포맷을 제공한다.

```
{
  "timestamp": "2025-08-29T14:30:00",
  "status": 400,
  "error": "Bad Request",
  "message": "Title must not be blank",
  "path": "/api/posts"
}
```

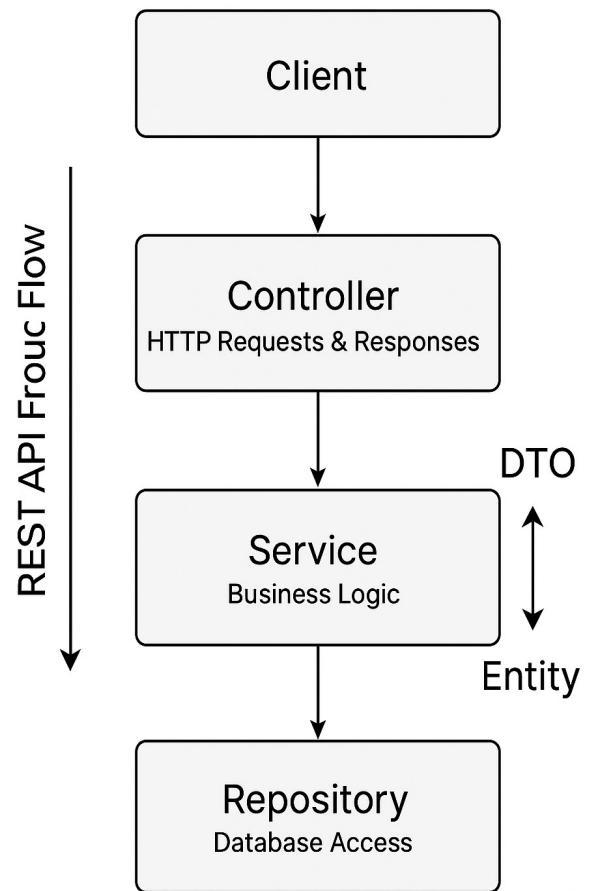
- **timestamp**: 오류 발생 시각
- **status**: HTTP 상태 코드
- **error**: 상태 코드에 대응하는 에러 문자열
- **message**: 사용자에게 전달할 구체적 메시지
- **path**: 요청 경로

Quick Review

1. REST API의 핵심 특징은 무엇인가?
2. 엔티티와 DTO를 구분하는 이유는 무엇인가?
3. 입력 검증 실패 시 어떤 예외가 발생하는가?
4. 전역 예외 처리기의 장점은 무엇인가?

Checklist

- REST API 엔드포인트가 리소스 지향적으로 설계되었는가?
- DTO를 사용해 민감한 데이터 노출을 방지했는가?
- 서비스 계층에서 트랜잭션 경계를 명확히 했는가?
- 표준 에러 응답을 적용했는가?



REST API request flow

Pitfalls

- REST 원칙을 따르지 않고 URL에 동작을 표현하는 실수(/getPostList 등).
- 엔티티를 그대로 API 응답으로 반환하여 개인정보가 노출되는 실수.
- 입력 검증을 소홀히 하여 SQL 오류나 보안 취약점이 발생하는 문제.
- 예외 처리를 각 컨트롤러에 중복 작성하는 비효율.

3-5. 데이터 액세스와 트랜잭션 관리 - JPA, N+1 문제)

JPA와 스프링 데이터 JPA

스프링 부트에서 가장 많이 쓰이는 데이터 접근 방법은 Spring Data JPA이다. 복잡한 SQL을 일일이 작성하지 않고도, 간단한 메서드 정의만으로 데이터를 저장하고 조회할 수 있다.

```
@Entity
public class Member {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;
    private int age;
    // Getter, Setter 생략
}

--
public interface MemberRepository extends JpaRepository<Member, Long>
{
    List<Member> findByName(String name);
}
```

- JpaRepository를 상속하면 save, findAll, delete 같은 CRUD 메서드가 자동 제공된다.
- findByName처럼 메서드 이름만 정의하면 쿼리가 자동 생성된다.
- 👉 복잡한 DAO 클래스 작성이 필요 없어지고, 개발자는 비즈니스 로직에 집중할 수 있다.

트랜잭션 관리

데이터베이스 작업은 트랜잭션 단위로 이루어진다. “모두 성공하거나, 모두 실패해야 한다”는 원칙이다. 스프링에서는 `@Transactional`을 붙이면 이 과정을 자동으로 처리한다.

```
@Service
public class MemberService {
    private final MemberRepository memberRepository;

    public MemberService(MemberRepository memberRepository) {
        this.memberRepository = memberRepository;
    }

    @Transactional
    public Member createMember(String name, int age) {
        Member member = new Member();
        member.setName(name);
        member.setAge(age);
        return memberRepository.save(member);
    }
}
```

- `@Transactional`이 붙은 메서드는 실행 시 트랜잭션이 열리고, 성공하면 커밋, 실패하면 롤백된다.
 - 단순 조회만 한다면 `@Transactional(readOnly = true)`로 최적화할 수 있다.
- 👉 개발자가 따로 커밋/롤백 코드를 작성할 필요가 없다.

N+1 문제

JPA를 쓰다 보면 성능 이슈로 자주 언급되는 것이 바로 **N+1** 문제이다.

예를 들어, 회원(Member)과 주문(Order)이 연결된 상황에서, 주문 목록을 조회하면서 각 주문에 연결된 회원 이름을 출력한다고 하자.

```
@Entity
public class Order {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @ManyToOne(fetch = FetchType.LAZY)
    private Member member;
}

--
List<Order> orders = orderRepository.findAll();
for (Order o : orders) {
    System.out.println(o.getMember().getName());
}
```

- `JpaRepository`를 상속하면 `save`, `findAll`, `delete` 같은 CRUD 메서드가 자동 제공된다.
 - `findByName`처럼 메서드 이름만 정의하면 쿼리가 자동 생성된다.
- 👉 복잡한 DAO 클래스 작성이 필요 없어지고, 개발자는 비즈니스 로직에 집중할 수 있다.

해결 방법

N+1 문제를 막기 위해 JPA는 여러 방법을 제공한다.

1. Fetch Join 사용

```
@Query("SELECT o FROM Order o JOIN FETCH o.member")
List<Order> findAllWithMember();
```

→ 주문과 회원을 한 번에 가져온다.

2. 엔티티 그래프(EntityGraph)

```
@EntityGraph(attributePaths = {"member"})
@Query("SELECT o FROM Order o")
List<Order> findAllEntityGraph();
```

→ 쿼리 실행 시 특정 연관 엔티티를 함께 로딩한다.

3. Batch Size 조정

```
spring:
  jpa:
    properties:
      hibernate.default_batch_fetch_size: 100
```

→ 여러 건의 연관 엔티티를 묶어서(in 절) 가져온다.

👉 상황에 따라 적절한 방법을 조합하는 것이 중요하다.

정리

- Spring Data JPA는 DAO 코드를 대폭 줄여준다.
- 트랜잭션은 서비스 계층에 선언해 데이터 일관성을 보장한다.
- N+1 문제는 성능 저하의 원인이 되므로 반드시 해결 전략을 마련해야 한다.

Quick Review

1. Spring Data JPA를 쓰면 어떤 이점이 있는가?
2. 트랜잭션을 서비스 계층에 두는 이유는 무엇인가?
3. N+1 문제가 발생하는 원리를 설명해 보라.
4. Fetch Join과 EntityGraph는 어떤 차이가 있는가?

Checklist

- Repository가 JpaRepository를 확장했는가?
- 서비스 계층에 @Transactional이 선언되어 있는가?
- N+1 문제가 발생하지 않도록 쿼리를 점검했는가?
- Batch Size 등 성능 최적화 설정을 적용했는가?

Pitfalls

- 트랜잭션 없이 엔티티를 조작하다 예외가 발생하는 실수
- 연관관계를 EAGER로 설정해 불필요한 쿼리를 남발하는 실수
- N+1 문제를 인식하지 못해 성능 저하를 겪는 실수
- Repository 계층에 트랜잭션 로직을 넣어 아키텍처가 무너지는 실수

3-6. 인증/인가(Spring Security, JWT/OAuth2)

보안의 필요성

웹 애플리케이션에서 보안은 선택이 아니라 필수이다. 사용자가 로그인한 뒤 자신의 자원만 접근할 수 있도록 제어하지 않으면 개인정보 유출이나 데이터 변조가 쉽게 발생한다. 특히 REST API 기반 서버에서는 세션(Session) 대신 토큰(Token)을 이용한 인증이 주로 사용된다.

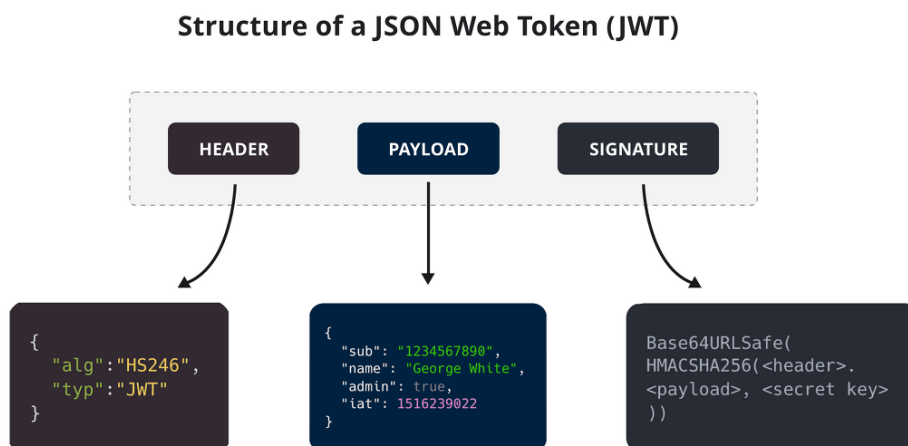
💡 여기서 가장 많이 쓰이는 방식이 바로 JWT(JSON Web Token)이다. 또한, OAuth2와 같은 표준 인증 프로토콜도 외부 서비스 연동에 널리 사용된다.

JWT란 무엇인가?

JWT는 JSON 포맷으로 사용자 인증 정보를 담은 토큰이다. 구조는 세 부분으로 나뉜다.

- **Header:** 토큰 타입(JWT)과 서명 알고리즘(예: HS256) 정보가 들어간다.
- **Payload:** 사용자 ID, 권한(Role) 등 인증 정보가 들어간다.
- **Signature:** 비밀키로 Header와 Payload를 서명한 값으로, 위변조 여부를 검증한다.

이 구조 덕분에 서버는 토큰만 보고도 사용자 식별과 권한을 확인할 수 있다.



인증(Authentication)과 인가(Authorization)

- 인증(Authentication): 사용자가 누구인지 확인하는 과정 (예: 로그인)
- 인가(Authorization): 인증된 사용자가 무엇을 할 수 있는지 권한을 확인하는 과정 (예: 관리자만 접근 가능)

JWT는 이 두 과정을 단일 토큰으로 처리할 수 있게 해준다.

Authentication



Who are you? →

Authorization



What can you do?

Spring Security와의 통합

스프링 부트에서 JWT를 실무적으로 적용할 때는 보통 Spring Security와 함께 사용한다.

- **필터 기반 보안**: Spring Security는 서블릿 필터 체인을 통해 인증/인가를 수행한다. JWT 검증 로직은 `OncePerRequestFilter`를 구현해 필터 체인에 등록한다.
- **Authentication 객체**: 토큰 검증이 끝나면 `UsernamePasswordAuthenticationToken` 같은 Authentication 객체를 생성하여 `SecurityContext`에 저장한다.
- **권한(Role) 기반 인가**: 컨트롤러나 서비스 메서드에 `@PreAuthorize("hasRole('ADMIN')")` 같은 애너테이션을 사용하여 접근을 제어한다.

OAuth2 개요

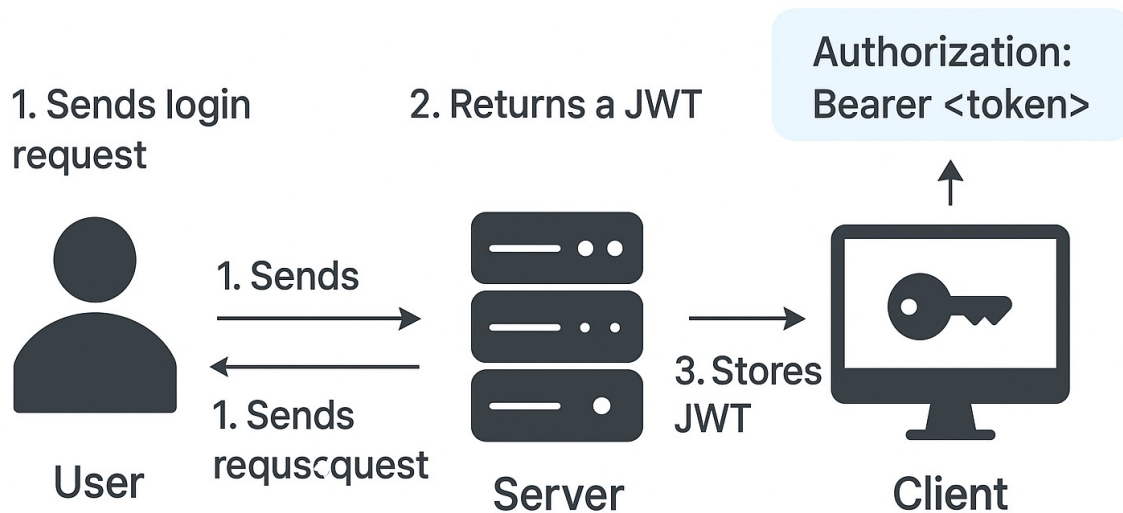
JWT 외에도 OAuth2는 외부 서비스(Google, Facebook, GitHub 등)와 연동할 때 널리 사용된다.

- 사용자가 직접 아이디/비밀번호를 입력하지 않고, 외부 서비스의 인증을 위임받아 로그인할 수 있다.
- Spring Security는 `spring-boot-starter-oauth2-client` 의존성을 통해 소셜 로그인과 같은 기능을 쉽게 구현할 수 있다.
- OAuth2는 내부 서비스의 JWT 기반 인증과 함께 사용하여, 자체 회원 + 소셜 로그인 회원을 동시에 지원할 수 있다.



JWT 인증 흐름

1. 사용자가 로그인 요청을 보낸다 (아이디/비밀번호).
2. 서버는 로그인 정보를 검증한 뒤, 사용자 정보를 담은 JWT를 생성하여 반환한다.
3. 클라이언트는 JWT를 로컬 스토리지나 쿠키에 저장한다.
4. 이후 요청 시 HTTP Header의 Authorization: Bearer <토큰> 형태로 전송한다.
5. 서버는 토큰의 유효성을 검증하고, 권한에 따라 요청을 처리한다.



스프링 부트에서 JWT 구현 예시

폴더 구조 권장

아래 예시는 로그인 → JWT 발급 → 요청마다 토큰 검증의 최소 구현이다. 각 단계에서 어떤 파일을 어디에 두고 무엇이 일어나는지를 함께 설명한다.

```
com.example.security
├─ JwtUtil.java // 토큰 생성·검증 유틸
├─ JwtFilter.java // 매 요청마다 토큰 검증 필터
├─ SecurityConfig.java // Spring Security 설정(필터 체인 등록)
└─ AuthController.java // 로그인(토큰 발급) API
```

1) 의존성 추가 – 왜 필요한가?

JWT 문자열을 만들고 파싱하려면 라이브러리가 필요하다. Spring Security는 필터 체인과 권한 모델을 제공한다.

```
dependencies {
    implementation 'io.jsonwebtoken:jjwt-api:0.11.5'
    runtimeOnly 'io.jsonwebtoken:jjwt-impl:0.11.5'
    runtimeOnly 'io.jsonwebtoken:jjwt-jackson:0.11.5'
    implementation 'org.springframework.boot:spring-boot-starter-security'
}
```

스프링 부트에서 JWT 구현 예시

2) JwtUtil - 무엇을 하나?

- 토큰 발급: 사용자 식별자(subject)와 만료 시간(exp)을 넣어 서명한다.
- 토큰 검증: 서명을 확인하고, 만료되지 않았는지 점검한 뒤 subject를 꺼낸다.

```
@Component
public class JwtUtil {
    // 실서비스에서는 환경변수/Secrets로 주입한다.
    private static final String SECRET = "change-me";

    public String generateToken(String username) {
        return Jwts.builder()
            .setSubject(username)
            .setIssuedAt(new Date())
            .setExpiration(new Date(System.currentTimeMillis() + 1000 * 60 * 60))
            .signWith(SignatureAlgorithm.HS256, SECRET)
            .compact();
    }

    public String validateAndGetUsername(String token) {
        return Jwts.parser().setSigningKey(SECRET)
            .parseClaimsJws(token)
            .getBody().getSubject();
    }
}
```


스프링 부트에서 JWT 구현 예시

3) AuthController 언제 호출되나?

- 클라이언트가 `/api/auth/login`으로 아이디/비밀번호를 보낸다.
- 서버가 검증 후 JWT를 만들어 반환한다.

```
@RestController
@RequestMapping("/api/auth")
public class AuthController {
    private final JwtUtil jwtUtil;
    public AuthController(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil; }

    @PostMapping("/login")
    public Map<String, String> login(@RequestBody Map<String, String> req) {
        String username = req.get("username");
        String password = req.get("password");
        // TODO: 실제로는 DB/PasswordEncoder로 검증한다.
        if ("user".equals(username) && "pass".equals(password)) {
            return Map.of("token", jwtUtil.generateToken(username));
        }
        throw new RuntimeException(HttpStatus.UNAUTHORIZED, "Invalid credentials");
    }
}
```

스프링 부트에서 JWT 구현 예시

4) JwtFilter 왜 필터인가?

- 모든 요청 전에 실행되어 Authorization 헤더의 JWT를 검증한다.
- 유효하면 `SecurityContext`에 인증 정보를 채워 이후 인가(@PreAuthorize 등)가 동작한다.

```
@Component
public class JwtFilter extends OncePerRequestFilter {
    private final JwtUtil jwtUtil;

    public JwtFilter(JwtUtil jwtUtil) {
        this.jwtUtil = jwtUtil;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain
fc)
        throws ServletException, IOException {
        String header = req.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            try {
                String username = jwtUtil.validateAndGetUsername(header.substring(7));
                var auth = new UsernamePasswordAuthenticationToken(username, null, List.of());
                SecurityContextHolder.getContext().setAuthentication(auth);
            } catch (JwtException e) {
                // 유효하지 않은 토큰: 컨텍스트를 비우고 다음 필터로 넘긴다(최종 인가에서 거부된다)
                SecurityContextHolder.clearContext();
            }
        }
        fc.doFilter(req, res);
    }
}
```

스프링 부트에서 JWT 구현 예시

5) SecurityConfig 어디에 끼우나?

- CSRF는 REST API에서는 보통 비활성화한다.
- 로그인/헬스체크 같은 공개 엔드포인트는 `permitAll`, 나머지는 인증 필요.
- 커스텀 `JwtFilter`를 `UsernamePasswordAuthenticationFilter` 앞에 배치한다.

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    SecurityFilterChain filter(HttpSecurity http, JwtFilter jwtFilter) throws Exception {
        http.csrf(AbstractHttpConfigurer::disable) // REST API에서는 보통 CSRF를 끄
        .sessionManagement(sm -> sm.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
        // 세션 대신 JWT 사용
        .authorizeHttpRequests(auth -> auth
            .requestMatchers("/api/auth/**", "/actuator/health").permitAll()
            // 인증 불필요한 엔드포인트
            .anyRequest().authenticated()) // 나머지는 인증 필요
        .addFilterBefore(jwtFilter, UsernamePasswordAuthenticationFilter.class); // JWT 필터 등록

        return http.build();
    }
}
```

스프링 부트에서 JWT 구현 예시

6) 실행·검증 어떻게 확인하나?

1. 로그인하여 토큰 발급

```
curl -s -X POST http://localhost:8080/api/auth/login \
-H 'Content-Type: application/json' \
-d '{"username":"user","password":"pass"}'

# {"token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
```

2. 보호된 API 호출

```
TOKEN=eyJ... # 위에서 받은 토큰
curl -H "Authorization: Bearer $TOKEN"
http://localhost:8080/api/hello
```

→ 200 응답이면 인증이 동작한다.

7) 자주 만나는 오류와 해결

- **403 Forbidden**: 필터가 인증을 채우지 못했다. Authorization 헤더 형식과 필터 등록 순서를 확인한다.
- **401 Unauthorized**: 토큰이 없거나 만료되었다. 만료 시간(exp)과 클라이언트 전송 여부를 확인한다.
- **서명 검증 실패**: 서버 시크릿이 변경되었거나 잘못 주입되었다. 환경변수로 주입하고 일관성 유지한다.

정리

- JWT는 세션을 대체할 수 있는 대표적인 인증 방식이다.
- Spring Security와 결합하면 필터 기반의 강력한 보안 체계를 구성할 수 있다.
- OAuth2는 외부 서비스와 연동된 인증을 구현할 수 있으며, JWT와 병행하여 사용할 수 있다.
- 클라이언트는 토큰을 저장하고, 요청마다 Authorization 헤더에 담아 보낸다.
- 서버는 토큰을 검증하고, 사용자 권한에 따라 접근을 허용한다.

Quick Review

1. JWT가 세션 기반 인증과 다른 점은 무엇인가?
2. JWT의 구성 요소 3가지를 설명하라.
3. 인증과 인가의 차이는 무엇인가?
4. Spring Security에서 JWT를 적용하기 위한 핵심 단계는 무엇인가?
5. OAuth2는 어떤 상황에서 유용한가?

Pitfalls

- 토큰 Payload에 민감한 정보를 직접 담는 실수
- 토큰 만료/재발급을 구현하지 않아 보안 구멍이 생기는 경우
- HTTPS 없이 토큰을 주고받아 탈취당하는 위험
- Spring Security의 필터 체인을 올바르게 이해하지 못해 인증/인가 로직이 꼬이는 경우
- OAuth2를 도입하면서 리다이렉션 URL과 권한 스코프를 잘못 설정하는 경우

3-7. 설정 전략(application.yml, Profiles, Secret 관리)

application.yml 구조 이해하기

스프링 부트는 모든 환경설정을 코드 안에 박아두지 않고, application.yml 파일을 통해 관리한다. 이 파일은 Key-Value 계층 구조로 되어 있어 직관적으로 읽을 수 있다.

예시:

```
server:
  port: 8080

spring:
  datasource:
    url: jdbc:mysql://localhost:3306/demo
    username: root
    password: root
  jpa:
    hibernate:
      ddl-auto: update
    show-sql: true
```

- `server.port`: 서버 실행 포트 지정 (기본값은 8080)
- `spring.datasource`: DB 연결 정보
- `spring.jpa`: 엔티티 자동 생성 여부, SQL 로그 출력 여부 등 JPA 관련 설정

👉 장점: 설정을 코드에서 분리하면 코드를 건드리지 않고도 실행 환경을 쉽게 바꿀 수 있다.

Profiles로 환경 분리하기

실무에서는 보통 개발(Dev) / 테스트(Test) / 운영(Prod) 환경이 따로 있다. 운영 DB와 개발용 로컬 DB의 접속 정보가 같은 파일에 섞여 있다면, 배포 시 실수하기 쉽다.

이를 해결하기 위해 Profiles 기능을 사용한다.

예시 (application-dev.yml):

```
spring:
  datasource:
    url: jdbc:mysql://localhost:3306/devdb
    username: dev
    password: dev123
```

예시 (application-prod.yml):

```
spring:
  datasource:
    url: jdbc:mysql://db.prod.example:3306/proddb
    username: admin
    password: securepwd
```

실행 시 환경을 지정:

```
java -jar app.jar --spring.profiles.active=dev
```

👉 이렇게 하면 같은 코드를 그대로 두고, 환경별 설정만 바꿔서 실행할 수 있다.

👉 교재 Tip: “운영 환경에서 절대 dev 설정이 켜지지 않도록 주의하라”

Secret 관리 전략

DB 비밀번호나 API 키 같은 민감한 값을 그대로 application.yml에 적어두는 것은 보안 사고로 직결된다.

따라서 반드시 외부 관리 방식으로 배야 한다.

1) 환경 변수 활용

```
spring:
  datasource:
    username: ${DB_USER}
    password: ${DB_PASS}
```

실행 환경에서 변수 주입:

```
export DB_USER=myuser
export DB_PASS=mypassword
java -jar app.jar --spring.profiles.active=prod
```

2) Spring Cloud Config

- 중앙 설정 서버를 두고, 각 애플리케이션이 실행될 때 설정과 비밀값을 받아온다.
- 여러 개의 마이크로서비스 환경에서 특히 유용하다.

3) 클라우드 Secret Manager

- AWS Systems Manager Parameter Store, AWS Secrets Manager, GCP Secret Manager 등 제공 서비스 활용
- 운영 환경에서는 비밀번호를 직접 yml에 넣지 않고, Secret Manager에서 가져오도록 한다.

👉 원칙: Git 저장소에는 비밀번호를 절대 남기지 않는다.

정리

- application.yml → 공통 설정 관리
- Profiles → 환경별 설정 분리 (dev, test, prod)
- Secret 관리 → 환경 변수 / Config 서버 / 클라우드 Secret Manager 활용

Quick Review

1. application.yml로 설정을 외부화하는 이유는 무엇인가?
2. Profiles를 쓰면 어떤 이점이 있는가?
3. Secret을 직접 yml에 적는 것이 왜 위험한가?

Checklist

- 공통 설정과 환경별 설정이 분리되어 있는가?
- 민감한 정보가 Git 저장소에 포함되지 않았는가?
- 운영 서버에서 올바른 Profile이 활성화되는가?
- Secret Manager나 환경 변수를 이용해 비밀값을 관리하고 있는가?
- 로깅 레벨이 환경에 맞게 조정되어 있는가?

Pitfalls

- 모든 환경 설정을 한 파일에 뒤섞어 관리하다가 운영 환경에서 장애 발생
- DB 비밀번호, API 키를 깃허브에 올려버리는 보안 사고
- 운영 서버에서 dev 프로파일이 활성화되는 치명적 실수
- Secret Manager를 쓰면서 권한 설정(IAM Role 등)을 빼먹어 접근 오류 발생

3-8. 관측성(Actuator, Micrometer, 로그 전략)

관측성이란 무엇인가

운영 환경에서는 애플리케이션이 잘 동작하는지만 보는 것이 아니라, 어떻게 동작하는지, 어디서 문제가 발생하는지, 어떤 지표가 중요한지를 알아야 한다. 이를 관측성(Observability)이라 한다.

관측성은 세 가지 축으로 구성된다.

- **Metrics**: CPU, 메모리, 요청 수, 지연 시간 등 수치 데이터
- **Logs**: 이벤트가 발생했을 때 남기는 텍스트 기록
- **Traces**: 요청이 여러 서비스를 거칠 때의 흐름 추적

스프링 부트는 Actuator와 Micrometer를 통해 관측성을 지원한다.



Metrics

CPU, memory, request count, latency, etc.



Logs

Text records of events



Traces

Tracking the request path

Spring Boot Actuator

Actuator는 애플리케이션 상태를 확인할 수 있는 다양한 엔드포인트를 제공한다.

1) 의존성 추가

```
dependencies {  
    implementation 'org.springframework.boot:spring-boot-starter-actuator'  
}
```

2) 대표 엔드 포인트

- 1) `/actuator/health`: 애플리케이션 상태 확인 (UP/DOWN)
- 2) `/actuator/metrics`: 기본 메트릭 확인
- 3) `/actuator/env`: 환경 변수와 설정 값 확인
- 4) `/actuator/loggers`: 로깅 레벨 동적 변경

👉 개발 단계에서는 유용하지만, 운영 환경에서는 접근 권한을 반드시 제한해야 한다.



Spring Boot Actuators

Micrometer와 모니터링 시스템 연동

Micrometer는 스프링 부트 애플리케이션 안에서 메트릭을 수집하는 역할을 한다. CPU 사용량, 메모리, 요청 수, 응답 시간 등을 자동으로 수집하고, Actuator와 함께 동작하여 외부 모니터링 시스템으로 보낼 수 있다.

1) Prometheus란 무엇인가?

Prometheus는 오픈소스 모니터링 시스템이자 시간 기반 데이터베이스(Time-series DB)이다. 쉽게 말해, 를 계속 기록하는 도구이다. 시간이 지남에 따라 변하는 숫자(메트릭)

- **Pull 방식:** Prometheus 서버가 애플리케이션에게 주기적으로 “지금 상태 알려줘” 하고 요청한다.
- **시계열 저장:** CPU, 메모리, 요청 수 같은 데이터를 시간축으로 저장해 추세를 분석할 수 있다.
- **PromQL:** 저장된 데이터를 조회하는 전용 쿼리 언어 제공
- **시각화 가능:** Grafana와 연동하면 대시보드 형태로 지표를 확인할 수 있다.

💡 정리하면, Micrometer가 애플리케이션 내부에서 데이터를 모으면, Prometheus가 그 데이터를 수집해 저장하고, Grafana 같은 도구가 예쁘게 시각화한다.



Micrometer와 모니터링 시스템 연동

2) Prometheus 연동 예시

build.gradle에 Prometheus 레지스트리 의존성 추가

```
implementation 'io.micrometer:micrometer-registry-prometheus'
```

application.yml 설정:

```
management:
  endpoints:
    web:
      exposure:
        include: health, metrics, prometheus
  metrics:
    tags:
      application: demo-app
```

- /actuator/prometheus 엔드포인트에서 Prometheus가 읽을 수 있는 형식으로 메트릭 제공
 - Prometheus 서버가 이 엔드포인트를 주기적으로 스크랩(scrape)하여 데이터 저장
 - Grafana를 Prometheus와 연결하면 CPU, 요청 수, 지연 시간 등을 차트로 시각화 가능
- 👉 이런 흐름으로 Spring Boot → Micrometer → Prometheus → Grafana 구조가 완성된다.

로그 전략

로그는 문제 해결의 핵심이다. 하지만 로그를 무분별하게 남기면 디스크가 가득 차거나, 중요한 정보가 묻히게 된다. 따라서 전략적인 로그 관리가 필요하다.

1) 로그 레벨 구분

- **TRACE**: 가장 상세한 디버깅용
- **DEBUG**: 개발 환경에서 동작 확인용
- **INFO**: 운영 환경에서 정상 동작을 알리는 수준
- **WARN**: 잠재적 문제
- **ERROR**: 즉각 조치가 필요한 오류

💡 운영 환경에서는 보통 INFO 이상을 기본으로 하고, 특정 패키지에 한해 DEBUG를 설정한다.

2) yml을 통한 설정 예시

```
logging:
  level:
    root: INFO
    com.example.demo: DEBUG
  file:
    name: logs/app.log
```

3) 중앙 집중 로그 관리

- ELK(Elasticsearch, Logstash, Kibana) 스택
- CloudWatch Logs (AWS)

→ 여러 서버에서 발생하는 로그를 한 곳에 모아 검색하고 분석할 수 있다.

🚫 **주의:** 비밀번호, 개인정보 등 민감한 값은 로그에 남기지 않는다.

정리

- Actuator는 애플리케이션 상태와 메트릭을 엔드포인트로 제공한다.
- Micrometer는 다양한 모니터링 시스템과 연동되어 시각화를 가능하게 한다.
- 로그는 레벨을 구분해 전략적으로 남기고, 중앙 집중 시스템으로 수집한다.

Quick Review

1. 관측성을 구성하는 세 가지 축은 무엇인가?
2. Actuator의 /actuator/health 엔드포인트는 어떤 역할을 하는가?
3. Micrometer와 Prometheus를 연동하면 무엇이 가능한가?
4. 로그 전략에서 로그 레벨을 구분하는 이유는 무엇인가?

Checklist

- Actuator 엔드포인트가 활성화되어 있는가?
- 운영 환경에서 민감한 Actuator 엔드포인트가 보호되고 있는가?
- Micrometer를 통해 주요 지표를 수집하고 있는가?
- 로그 레벨이 환경별로 적절하게 설정되어 있는가?
- 중앙 집중 로그 관리 시스템이 구축되어 있는가?

Pitfalls

- 운영 서버에서 모든 Actuator 엔드포인트를 공개해 보안 취약점 발생
- 로그 레벨을 DEBUG로 유지해 성능 저하 및 정보 유출
- 메트릭을 수집하면서 태그를 과도하게 붙여 Prometheus 부하 발생
- 로그 로테이션을 설정하지 않아 디스크 용량 초과 문제 발생

3-9. 테스트(Unit/Slice/Integration, Testcontainers)

테스트의 중요성

안정적인 애플리케이션 개발을 위해서는 코드가 예상대로 동작하는지 검증하는 테스트가 필요하다. 스프링 부트는 JUnit과 함께 다양한 테스트 방법을 제공하며, 단위(Unit), 슬라이스(Slice), 통합(Integration) 테스트로 구분할 수 있다.

단위(Unit) 테스트

- 정의: 가장 작은 코드 단위(메서드, 클래스)를 검증한다.
- 도구: JUnit, Mockito
- 특징: 빠르고 독립적이며, 외부 시스템(DB, 네트워크)에 의존하지 않는다.

예시:

```
@SpringBootTest
class CalculatorTest {

    @Test
    void addTest() {
        Calculator calc = new Calculator();
        assertEquals(4, calc.add(2, 2));
    }
}
```


슬라이스(Slice) 테스트

- **정의:** 애플리케이션 일부 계층만 잘라서 검증한다. 전체 애플리케이션을 띄우는 것이 아니라, 특정 레이어에 집중해 테스트를 수행한다.
- **예시:** `@WebMvcTest`로 컨트롤러 계층만 테스트, `@DataJpaTest`로 JPA 계층만 테스트
- **장점:** 불필요한 빈들을 모두 로딩하지 않으므로 빠르고 가볍다. 특정 계층의 기능에만 집중해 버그를 빠르게 찾을 수 있다.
- **주의점:** 실제 동작 전체가 보장되는 것은 아니므로, 통합 테스트와 함께 병행해야 한다.

예시:

```
@WebMvcTest(HelloController.class)
class HelloControllerTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void helloTest() throws Exception {
        mockMvc.perform(get("/api/hello"))
            .andExpect(status().isOk())
            .andExpect(jsonPath("$.message").value("Spring Boot is working"));
    }
}
```

통합(Integration) 테스트

- 정의: 애플리케이션을 실제 실행한 환경과 유사하게 전체적으로 검증한다. 스프링 컨텍스트를 모두 띄워 실제 빈들이 협력하는 과정을 확인한다.
- 특징: DB, 외부 API 등 실제 환경과 통합 테스트 가능하며, 단위/슬라이스 테스트로 잡히지 않는 문제를 발견할 수 있다.
- 장점: 실제 서비스 동작과 유사하여 배포 전 안정성을 높인다.
- 단점: 실행 속도가 느릴 수 있으며, 환경 준비(테스트 DB 등)가 필요하다.

예시:

```
@SpringBootTest
@AutoConfigureMockMvc
class IntegrationTest {

    @Autowired
    private MockMvc mockMvc;

    @Test
    void contextLoads() throws Exception {
        mockMvc.perform(get("/api/hello"))
            .andExpect(status().isOk());
    }
}
```

Testcontainers

- **정의:** Docker 컨테이너를 이용해 실제 DB, 메시지 브로커 등을 띄워 테스트하는 라이브러리
- **장점:** 로컬 환경과 테스트 환경 차이를 줄일 수 있다.
- **예시:** 실제 MySQL 컨테이너를 띄워 Repository 테스트 실행예시:

```
@Testcontainers
@SpringBootTest
class UserRepositoryTest {

    @Container
    static MySQLContainer<?> mysql = new MySQLContainer<>("mysql:8.0")
        .withDatabaseName("testdb")
        .withUsername("test")
        .withPassword("test");

    @Autowired
    private UserRepository userRepository;

    @Test
    void saveUserTest() {
        User user = new User("Alice");
        userRepository.save(user);
        assertNotNull(user.getId());
    }
}
```



Testcontainers는 클라우드 환경과 유사한 테스트 환경을 제공하여, “내 PC에서는 잘 되는데...” 문제를 줄여준다.

정리

- 단위(Unit) 테스트: 빠르고 독립적인 가장 작은 단위 검증
- 슬라이스(Slice) 테스트: 특정 계층만 잘라서 검증
- 통합(Integration) 테스트: 실제 구동 환경과 유사한 전체 검증
- Testcontainers: 실제 컨테이너 환경을 띄워 신뢰도 높은 테스트 수행

Quick Review

1. 단위(Unit) 테스트와 통합(Integration) 테스트의 차이는 무엇인가?
2. 슬라이스(Slice) 테스트는 어떤 상황에서 유용한가?
3. Testcontainers를 사용하면 어떤 장점이 있는가?

Checklist

- JUnit 기반 단위 테스트가 작성되어 있는가?
- 슬라이스 테스트로 계층별 검증이 이루어졌는가?
- 통합 테스트가 실제 환경을 충분히 반영하는가?
- Testcontainers로 외부 시스템 테스트를 수행했는가?

Pitfalls

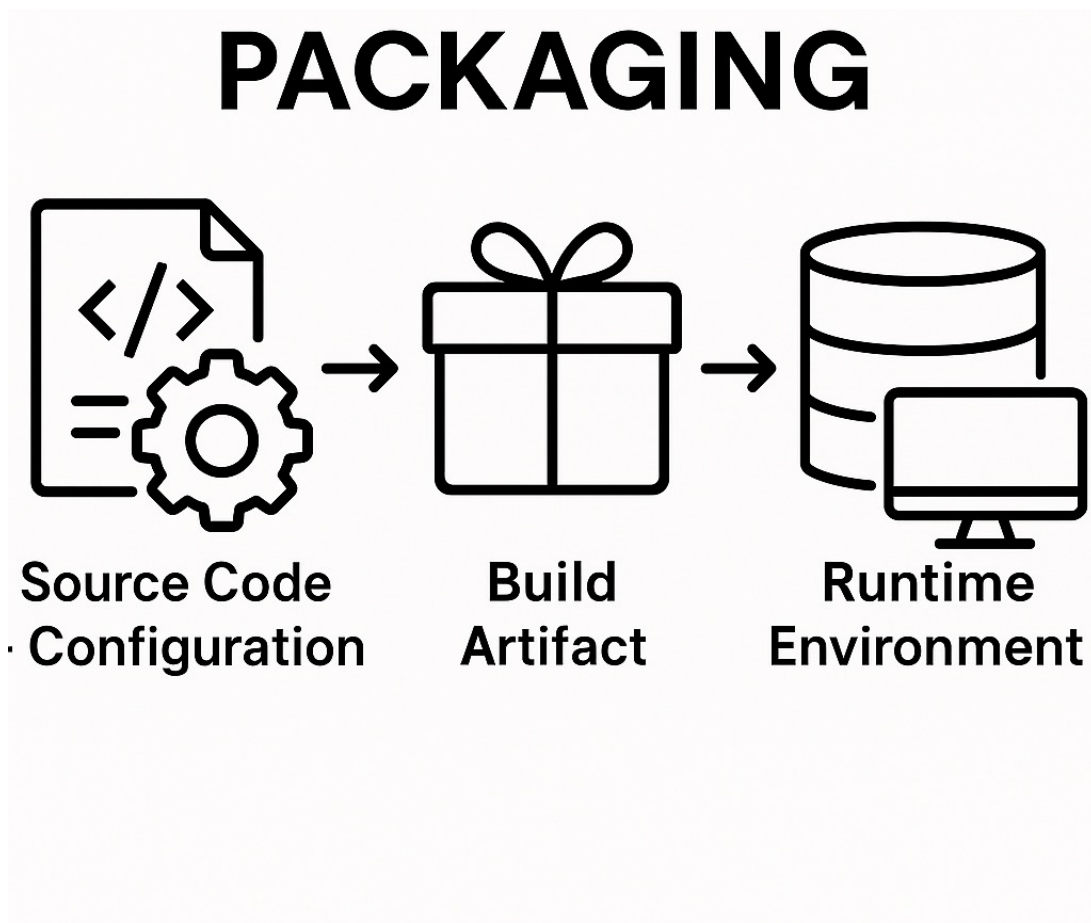
- 모든 테스트를 통합 테스트로만 작성해 실행 속도가 느려지는 실수
- Mock을 남발해 실제 동작과 괴리가 생기는 문제
- DB를 로컬에 직접 띄워 테스트하다가 환경 차이로 장애 발생
- Testcontainers 실행 속도를 고려하지 않아 CI 파이프라인에서 시간 초과 발생

3-10. 패키징/배포(Dockerfile, Jib, 레이어 최적화)

패키징의 필요성

개발자가 작성한 스프링 부트 애플리케이션은 소스 코드와 설정 파일, 빌드 산출물(JAR)로 구성된다. 이를 운영 환경에 배포하려면, 실행에 필요한 환경(JDK, 라이브러리, OS)을 함께 묶어야 한다.

이 과정을 패키징(packaging)이라 하고, 패키징된 애플리케이션을 실행 환경에 올리는 것을 배포(deployment)라 한다.



Dockerfile을 이용한 컨테이너 패키징

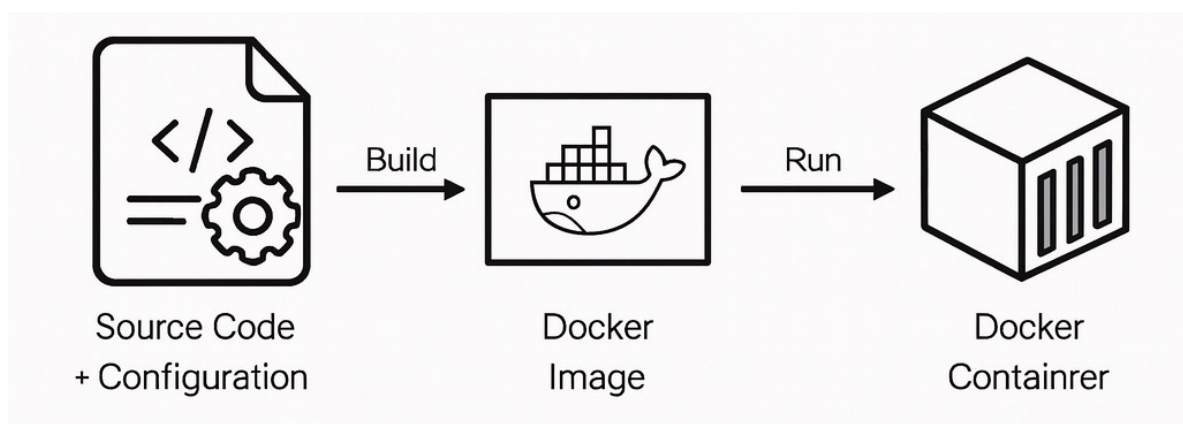
Docker는 애플리케이션과 실행 환경을 컨테이너라는 단위로 묶어 배포할 수 있게 한다.

```
# 1단계: 빌드 (Gradle/Maven으로 JAR 생성)
FROM gradle:8.5-jdk17 AS builder
WORKDIR /app
COPY . .
RUN gradle build -x test

# 2단계: 실행
FROM eclipse-temurin:17-jdk
WORKDIR /app
COPY --from=builder /app/build/libs/demo-0.0.1-SNAPSHOT.jar app.jar
ENTRYPOINT ["java", "-jar", "app.jar"]
```

- 멀티스테이지 빌드를 사용해 빌드 환경과 실행 환경을 분리한다.
- 실행 단계에는 빌드 도구(Gradle, Maven)가 포함되지 않아 이미지 크기를 줄일 수 있다.

💡 이미지가 크면 네트워크 전송과 배포 속도가 느려진다. 따라서 항상 최소한의 런타임 환경만 포함하는 것이 좋다.



Jib을 활용한 빌드 최적화

Google Jib는 Dockerfile 없이도 Java 애플리케이션을 컨테이너 이미지로 빌드할 수 있는 도구이다.

1) Gradle 설정

```
plugins {  
    id 'com.google.cloud.tools.jib' version '3.4.0'  
}  
  
jib {  
    to {  
        image = 'myrepo/demo-app:latest'  
    }  
}
```

2) 빌드 실행

```
./gradlew jibDockerBuild
```

- Jib은 클래스, 리소스, 의존성을 자동으로 구분하여 레이어별로 캐싱한다.
- 코드가 바뀌지 않았을 때는 레이어를 재사용하므로 빌드와 배포 속도가 빨라진다.

💡 운영 환경에서 자주 쓰이는 CI/CD 파이프라인(Jenkins, GitHub Actions)과 Jib을 연결하면, Dockerfile 없이도 자동 배포가 가능하다.

정리

- Dockerfile은 가장 전통적인 패키징 방식이다.
- Jib은 Dockerfile 없이도 레이어 최적화를 자동으로 지원한다.
- 레이어 최적화를 통해 빌드/배포 속도를 획기적으로 줄일 수 있다.

Quick Review

1. Dockerfile 멀티스테이지 빌드의 장점은 무엇인가?
2. Jib이 제공하는 레이어 최적화는 어떤 원리인가?
3. 레이어를 구성할 때 “자주 바뀌는 코드”와 “거의 안 바뀌는 의존성”은 각각 어디에 배치하는가?

Checklist

- 멀티스테이지 빌드를 사용해 이미지 크기를 줄였는가?
- Jib이나 Dockerfile 중 프로젝트에 맞는 방식을 선택했는가?
- 레이어 구성이 최적화되어 캐싱이 잘 동작하는가?
- CI/CD 파이프라인에서 자동 빌드·배포가 가능한가?

Pitfalls

- 빌드 도구나 테스트 파일까지 이미지에 포함해 이미지가 지나치게 커지는 문제
- 레이어 최적화를 고려하지 않아, 코드 한 줄 수정에도 모든 레이어가 재빌드되는 비효율
- 비밀 키나 설정 파일을 Docker 이미지에 하드코딩하는 보안 실수

3-11. 품질·성능·보안 체크리스트

스프링 부트 애플리케이션을 개발하고 배포하는 과정에서, 기능 구현만큼 중요한 것은 품질(Quality), 성능(Performance), 보안(Security)을 지속적으로 점검하는 것이다.

이 장에서는 운영 직전, 혹은 운영 중 점검해야 할 주요 항목들을 정리한다

품질(Quality)

- 코드 컨벤션: 팀 내 통일된 코드 스타일을 적용했는가? (Checkstyle, Spotless 등 도구 사용)
- 테스트 커버리지: 단위/슬라이스/통합 테스트가 핵심 기능을 충분히 커버하는가?
- 빌드 파이프라인: CI 환경에서 자동 빌드·테스트가 실행되는가?
- 문서화: API 문서(Swagger/OpenAPI)와 README가 최신 상태인가?
- 리뷰 프로세스: 코드 리뷰를 통해 품질을 상호 검증하는가?

성능(Performance)

- 프로파일링: JMeter, Gatling 등으로 부하 테스트를 수행했는가?
- 캐싱: Redis, Caffeine 등을 적절히 적용했는가?
- 쿼리 최적화: N+1 문제, 불필요한 JOIN을 점검했는가? (Hibernate log, p6spy 활용)
- 리소스 모니터링: CPU, 메모리, GC 로그를 수집하고 있는가?
- 스케일링 준비: 트래픽 급증 시 수평 확장이 가능한 구조인가?

보안(Security)

- 인증·인가: Spring Security와 JWT/OAuth2 기반의 보안 로직이 올바르게 적용되었는가?
- 민감정보 보호: 비밀번호, API 키, DB 접속 정보가 코드나 이미지에 포함되지 않았는가?
(Secret Manager, 환경 변수 사용)
- 입력 검증: SQL Injection, XSS, CSRF 등 입력 기반 공격에 대비했는가?
- HTTPS 적용: 운영 환경에서 TLS/SSL을 사용하고 있는가?
- 취약점 스캐닝: 의존성(Dependency) 보안 점검 도구(OWASP Dependency-Check, Snyk 등)를 사용했는가?

3-12. 미니 프로젝트: 게시판 API 설계·구현·테스트

1) 요구사항

본 프로젝트는 스프링 부트로 구현하는 게시판 API이다.

핵심 기능은 다음과 같다.

- 게시글: 목록 조회, 카테고리별 목록 조회, 상세 조회, 작성, 수정, 삭제
- 게시글 좋아요: 토글(등록/취소), 좋아요 수 조회, 좋아요 여부 확인
- 댓글: 작성, 목록 조회, 수정, 삭제
- 댓글 좋아요: 토글(등록/취소), 좋아요 수 조회

조회는 누구나 접근 가능하지만, 작성·수정·삭제 및 좋아요 토글은 JWT 기반 인증 사용자만 가능하다.

2) 데이터 모델

- **User**: id, username, password, roles
- **Category**: id, name
- **Post**: id, title, content, category, author, createdAt, updatedAt
- **PostLike**: id, post_id, user_id
- **Comment**: id, post_id, author, content, createdAt, updatedAt
- **CommentLike**: id, comment_id, user_id

Post는 Category와 User에 N:1 관계를 가진다.

Comment는 Post와 User에 N:1 관계를 가진다.

Like 엔티티는 Post/Comment와 User에 N:1 관계를 가진다.

3) 엔드포인트 설계

게시글(Post)

Aa 이름	≡ 태그	≡ API
📄 전체 게시글 목록 조회	GET	/posts
📄 카테고리별 게시글 목록 조회	GET	/posts?category={카테고리명}
📄 게시글 상세 조회	GET	/posts/{postId}
📄 게시글 작성	POST	/posts
📄 게시글 수정	PUT	/posts/{postId}
📄 게시글 삭제	DELETE	/posts/{postId}

게시글 좋아요(Post Like)

Aa 이름	≡ 태그	≡ API
📄 게시글 좋아요 등록/취소(Toggle)	POST	/posts/{postId}/like
📄 좋아요 수 조회	GET	/posts/{postId}/like
📄 좋아요 여부 확인	GET	/posts/{postId}/like/status

댓글(Comment)

Aa 이름	≡ 태그	≡ API
📄 댓글 작성	POST	/posts/{postId}/comments
📄 댓글 목록 조회	GET	/posts/{postId}/comment
📄 댓글 수정	PUT	/posts/{postId}/comments/{commentId}
📄 댓글 삭제	DELETE	/posts/{postId}/comments/{commentId}

댓글 좋아요(Comment Like)

Aa 이름	≡ 태그	≡ API
📄 댓글 좋아요 등록/취소	POST	/comments/{commentId}/like
📄 댓글 좋아요 수 조회	GET	/comments/{commentId}/like

4) DTO와 검증

입력 검증 실패 시에는 400 Bad Request와 함께 필드별 오류 메시지를 반환한다.

```
public record PostCreateRequest(  
    @NotBlank @Size(max = 120) String title,  
    @NotBlank String content,  
    @NotBlank String category  
) {}  
  
public record PostResponse(  
    Long id, String title, String content,  
    String category, String author, long  
    likeCount,  
    Instant createdAt, Instant updatedAt  
) {}
```

5) 서비스 로직 규칙

- 게시글/댓글의 수정·삭제는 작성자 또는 ADMIN만 수행할 수 있다.
- 좋아요 토글은 존재하면 삭제, 없으면 생성한다.
- 연관 데이터 삭제 시 학습 목적에서는 물리 삭제를 허용한다.

6) 컨트롤러 코드 예시

```
@PostMapping("/posts/{postId}/like")
public ResponseEntity<Map<String, Object>> toggle(@PathVariable Long postId,
                                                  @AuthenticationPrincipal UserPrincipal me)
{
    boolean liked = postLikeService.toggle(postId, me.id());
    long count = postLikeService.count(postId);
    return ResponseEntity.ok(Map.of("liked", liked, "count", count));
}
```

7) 테스트 전략

- 단위(Unit) 테스트: 서비스 로직(토글, 권한 검증)을 독립적으로 검증한다.
- 슬라이스(Slice) 테스트: `@WebMvcTest`로 컨트롤러 API 응답 스펙을 확인한다.
- 통합(Integration) 테스트: `@SpringBootTest`로 애플리케이션 전체 흐름을 검증한다.
- Testcontainers: MySQL/PostgreSQL 컨테이너를 사용하여 실제 DB 환경과 유사하게 테스트한다.

8) 완료 기준

- 모든 조회 API는 200 또는 404 상태 코드를 일관되게 반환한다.
- 인증이 필요한 API는 인증 실패 시 401, 권한 부족 시 403을 반환한다.
- 입력 검증 실패 시 400 + 오류 메시지를 반환한다.
- 테스트 커버리지가 60% 이상 확보된다.
- Dockerfile 또는 Jib으로 패키징 후 컨테이너 실행이 가능하다.

제 4 장

Docker

- 1) Docker란?
- 2) WSL2 설치하기
- 3) Docker 설치하기
- 4) Docker 실행 흐름
- 5) Docker 생명 주기와 명령어
- 6) Docker 이미지 파일 만들기

4-1. Docker란?

운영체제란?

운영체제(Operating System) 컴퓨터 하드웨어와 소프트웨어 자원을 관리하고, 사용자가 컴퓨터를 편리하게 사용할 수 있도록 환경을 제공하는 기본 소프트웨어이다. 운영체제는 사용자와 컴퓨터 하드웨어 사이에서 중개자 역할을 하며, 프로그램 실행, 파일 관리, 메모리 관리, 입출력 제어, 보안 관리 등의 핵심 기능을 수행한다.

PC와 서버 등 다양한 환경에서 사용되며 대표적인 운영체제로는 Windows, macOS, Linux 등이 있다.

리눅스란?

리눅스(Linux)는 가장 대표적인 오픈 소스 운영체제 중 하나로, Unix 운영체제를 모델로 하여 개발되었다. 1991년 리누스 토발즈(Linus Torvalds)가 최초로 커널을 개발하고 배포한 것이 시작이다.

리눅스는 오픈 소스 라이선스(GPL)로 배포되어 누구나 자유롭게 소스 코드를 사용, 수정, 배포할 수 있다. 이로 인해 다양한 용도와 환경에 맞는 수많은 배포판이 존재하며, 대표적인 배포판으로는 우분투(Ubuntu), 페도라(Fedora), CentOS, 데비안(Debian) 등이 있다.

리눅스는 안전성, 보안성, 유연성 덕분에 특히 서버, 임베디드 시스템, 클라우드 인프라 등에서 널리 사용되고 있다.

커널(Kernel)

‘Kernel’은 영어로 ‘알맹이’, ‘핵심’을 뜻한다. 운영체제의 핵심 부품으로서, CPU, 메모리, 파일 시스템, 네트워크, 입출력 장치 등 컴퓨터 자원을 직접 관리한다.

즉, 커널은 하드웨어와 소프트웨어 사이에서 자원 배분과 제어를 담당하는 가장 중요한 구성 요소이다.

셸(Shell)

커널은 시스템 자원 관리에 집중하기 때문에 사용자와 직접 상호작용하지 않는다. 이를 보완하기 위해 등장한 시스템 프로그램이 바로 ‘셸’이다.

셸은 사용자가 입력하는 명령어를 해석하고 실행하는 프로그램으로, 커널과 사용자 간의 다리 역할을 수행한다. 주로 터미널이나 명령 프롬프트를 통해 사용된다.

```
C:\Users\hey_m>ipconfig
```

```
Windows IP 구성
```

```
무선 LAN 어댑터 로컬 영역 연결* 1:
```

```
미디어 상태 . . . . . : 미디어 연결 끊김
연결별 DNS 접미사 . . . . . :
```

```
이더넷 어댑터 VMware Network Adapter VMnet1:
```

```
연결별 DNS 접미사 . . . . . :
링크-로컬 IPv6 주소 . . . . . : fe80::7e78:390d:481c:613c%3
IPv4 주소 . . . . . : 192.168.28.1
서브넷 마스크 . . . . . : 255.255.255.0
기본 게이트웨이 . . . . . :
```


온 프레미스(On-premise)

온 프레미스(On-premise) 방식이란, 과거에 서버를 관리하기 위해 데이터센터나 서버실에 직접 서버를 두고 운영하는 방식을 말한다. 이 방식에서는 서버에 운영체제를 설치하고 세팅하는 일이 쉽지 않아, 보통 하나의 서버에 하나의 운영체제와 하나의 프로그램만 운영하는 경우가 많았다. 이로 인해 물리 서버가 가진 성능을 100% 활용하지 못하는 비효율적인 상황이 발생하였고, 이를 위한 해결책이 필요로 해졌다.

가상화

기업은 서버를 효율적으로 운영하기 위해, 한 대의 물리 서버를 여러 대의 가상 서버처럼 나누어 사용하거나, 여러 대의 서버를 하나처럼 묶어 관리하는 방식을 원했다. 이러한 요구를 충족시키기 위해 등장한 기술이 바로 가상화이다.

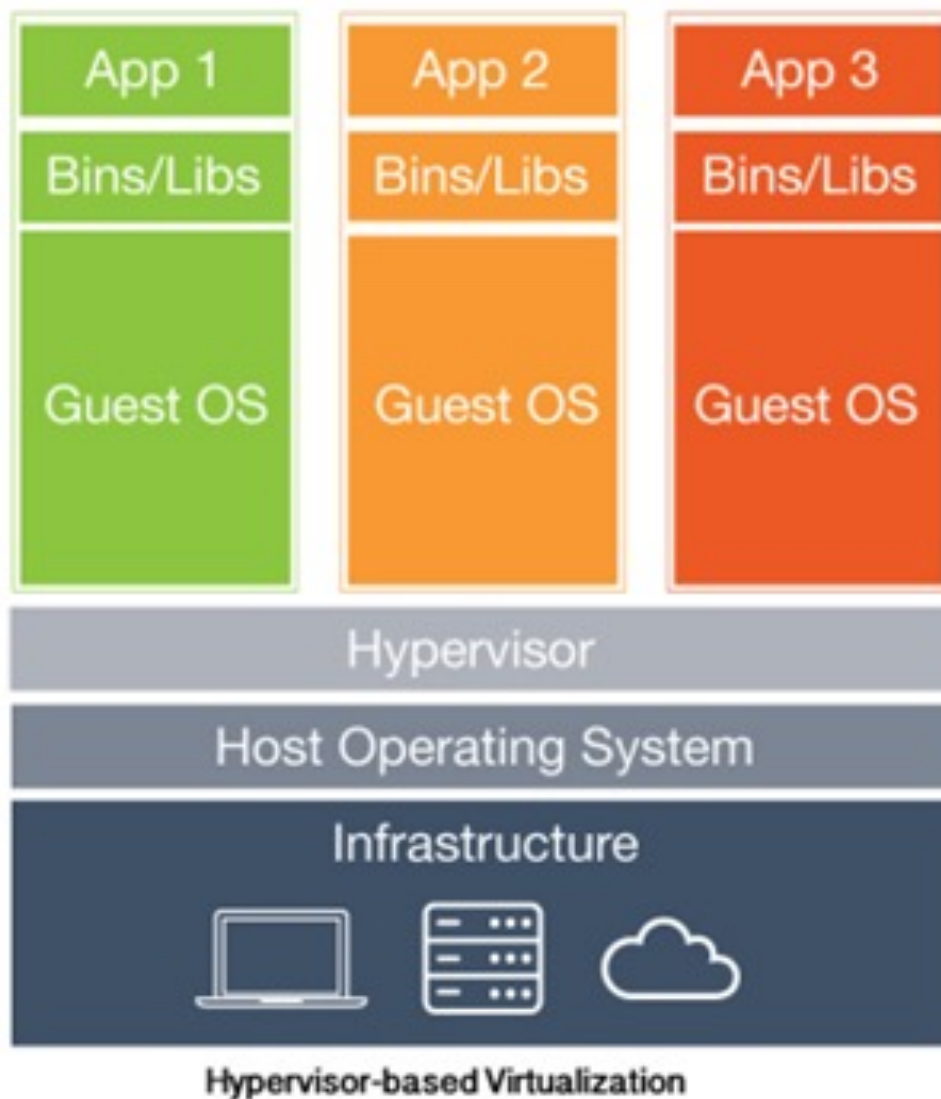
가상화란 가상화 관리 소프트웨어(주로 Hypervisor)를 사용하여 프로세서, 메모리, 스토리지 등 단일 컴퓨터의 하드웨어 자원을 여러 개의 가상 컴퓨터, 즉 가상 머신(Virtual Machine, VM)으로 분할하는 하드웨어 상의 추상화 계층을 구축하는 기술을 말한다. 각 가상 머신은 독립적인 운영체제와 애플리케이션 환경을 가지며, 마치 별도의 물리 서버처럼 동작한다.

이를 통해 자원 효율성이 증가하고, 유연한 운영과 관리가 가능해졌으며, 비용 절감 효과도 얻었다. 또한 각 가상 머신 간의 독립성이 보장되기에 보안도 강화되었다.

하이퍼바이저(Hypervisor)

하이퍼바이저는 하드웨어에서 운영체제와 리소스를 분리하여 가상 머신(VM)에 할당하고, 여러 VM을 생성 및 구동하는 소프트웨어이다. 주요 역할은 다음과 같다.

- 실제 CPU, RAM, 디스크, 네트워크 카드 등 하드웨어 자원을 가상화하여 VM에게 제공
- 각 VM은 자신의 독립적인 커널을 실행
- 여러 운영체제의 커널 명령을 하드웨어에 전달할 수 있도록 중재



서버 가상화의 단점

- 모든 명령이 하이퍼바이저를 거쳐야 하므로 일반 호스트에 비해 속도가 느리다
- 각 VM 이미지는 완전한 운영체제를 포함하므로 용량이 크다
- 각 가상 머신마다 독립된 운영체제를 가지고 있어서 추가적인 메모리 및 디스크 공간이 필요하다
- VM을 실행하지 않을 때도 할당된 디스크 공간은 계속 점유된다
- VM 간에 중복된 라이브러리와 애플리케이션이 각각 메모리에 로드된다.
- 다른 VM과 자원을 쉽게 공유할 수 없어서 중복이 발생한다
- 각 VM마다 관리가 필요하다

기존 가상화의 대표적인 예시로는 VMWare와 VirtualBox가 있다. 이러한 가상 머신을 사용하려면 처음에 게스트 운영체제를 별도로 설치해야 한다. 호스트 운영체제와 독립적으로 동작하는 게스트 운영체제 환경을 구축하고, 실제 PC처럼 별도의 환경설정도 필요하기 때문에 많은 시간이 소요되며, 메모리도 많이 사용한다.

컨테이너(Container)

컨테이너는 애플리케이션과 그 실행에 필요한 라이브러리, 설정 파일 등을 하나로 묶어 격리된 환경에서 실행할 수 있도록 해주는 가벼운 가상화 기술이다.

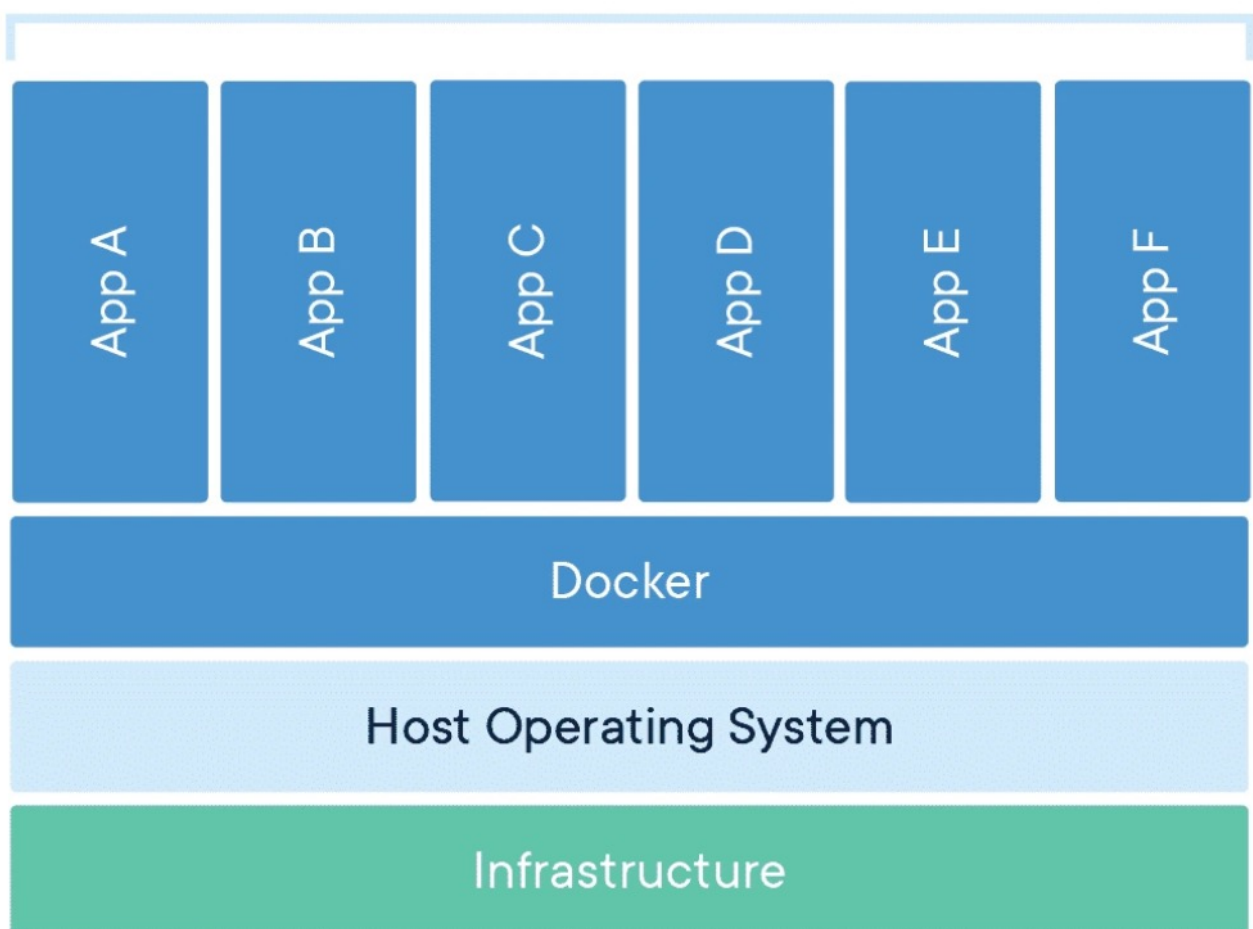
가상 머신(VM)과 달리, 운영체제 전체를 가상화 하지 않고 호스트 OS의 커널을 공유하기 때문에 리소스 사용에 효율적이고, 빠른 시작과 높은 이식성을 제공한다.

컨테이너는 개발환경과 운영환경 간의 차이로 인한 문제를 줄여주며, 일관된 실행 환경을 보장해 클라우드와 마이크로서비스 아키텍처에서 널리 사용된다.

Docker란?

컨테이너 가상화의 대표적인 기술이 바로 Docker이다. 도커 컨테이너는 리눅스의 chroot, 네임스페이스(namespace), cgroup과 같은 기능을 활용해 프로세스 단위의 격리된 환경을 생성하므로 성능 손실이 거의 없다. Docker를 사용하면 운영체제의 커널을 공유하는 경량 컨테이너를 쉽게 만들고 관리할 수 있어, 가상 머신보다 빠르고 자원 효율적으로 애플리케이션을 실행할 수 있다.

Containerized Applications



Docker의 특징

격리(Isolation)

Docker 컨테이너는 프로세스를 격리된 환경에서 관리하므로, 컨테이너 내 설정이나 작업이 호스트 운영체제에 영향을 주지 않는다. 이로 인해 각 컨테이너는 독립적인 개발 및 실행 환경을 보장받으며, 프로세스 관리와 확장이 용이해진다.

이식성(Protability)

Docker로 생성된 컨테이너는 Docker가 설치된 모든 환경에서 동일하게 실행된다. 컨테이너는 호스트 OS의 커널을 사용해 하드웨어와 통신하므로, 직접 하드웨어와 상호작용하지 않고 OS가 제공하는 표준화된 인터페이스를 통해 동작한다. 따라서 로컬 환경에서 만든 애플리케이션을 별도의 수정 없이 다른 서버 환경으로 쉽게 옮겨 실행할 수 있다.

경량성(Lightweight)

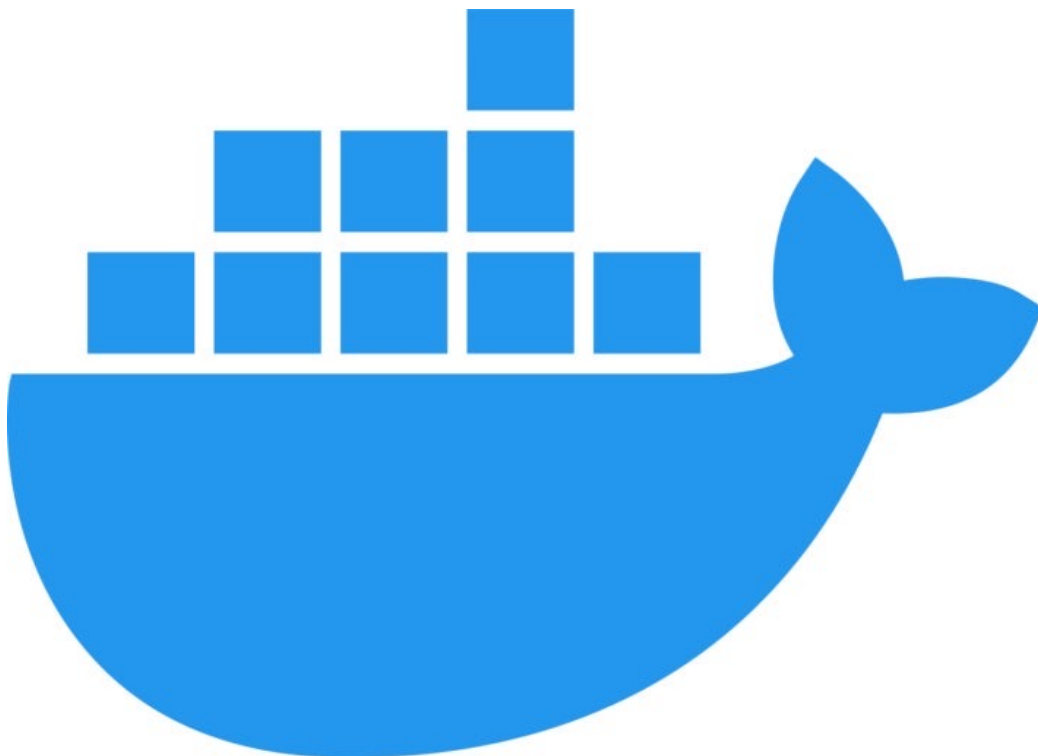
컨테이너는 가상 머신(VM)과 달리 전체 운영체제를 가상화하지 않아 메모리 사용량이 적고, 이미지 크기도 작다. 애플리케이션 실행에 필요한 파일과 라이브러리만 포함하여 효율적인 리소스 사용이 가능하다. 또한 컨테이너는 필요한 시점에만 자원을 사용하고, 사용하지 않을 때는 자원을 반환한다. 기존에 실행 중인 호스트 커널을 활용하기 때문에 별도의 커널 부팅 과정 없이 빠르게 작동하며, 한 명의 관리자가 여러 부서를 효율적으로 관리하는 것과 유사하다

버전 관리

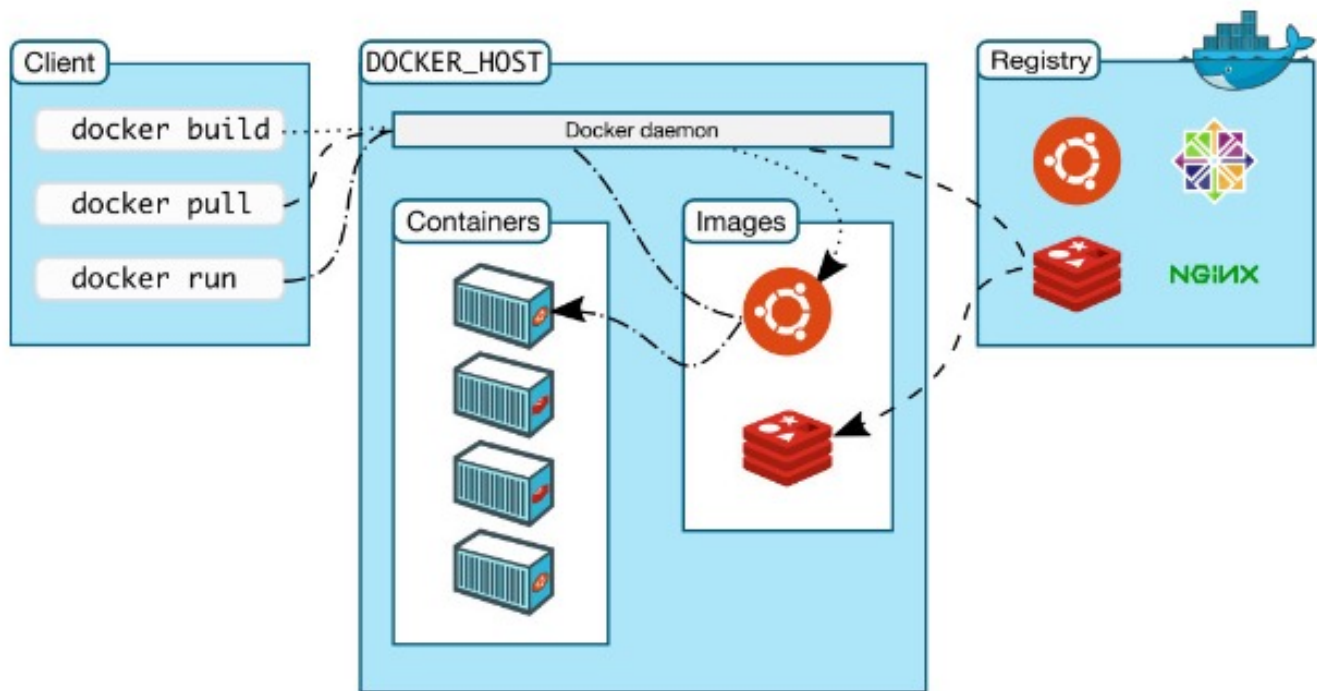
Docker는 이미지 빌드와 배포를 위한 버전 관리를 제공한다. 이미지는 여러 레이어로 구성되며, 변경 사항에 대해 새로운 레이어를 추가하여 효율적인 빌드와 저장 상황을 제공한다.

커뮤니티 및 이미지 공유

Docker Hub를 통해 다양한 Docker 이미지를 제공한다. 또한 사용자는 자신만의 Docker 이미지를 만들어 공유할 수 있다.



도커 아키텍처



Docker Client

도커를 설치하면 함께 제공되는 명령어 인터페이스로, 사용자가 `build`, `pull`, `run` 등의 명령을 실행하는 도구이다. 클라이언트는 이 명령을 Docker Daemon에 전달해 작업을 수행하도록 요청한다

Docker Host

도커가 설치되어있는 컨테이너와 이미지를 관리하는 서버 환경을 말한다. 컨테이너 실행, 이미지 저장, 네트워크 관리 등의 작업이 이 환경에서 이루어 진다.

Docker Daemon

도커 엔진의 핵심 프로세스로, 클라이언트의 명령을 받아 컨테이너 생성, 실행, 이미지 관리 등의 작업을 처리한다. 호스트 시스템에서 백그라운드로 실행된다.

Registry

도커 이미지를 저장하고 공유하는 원격 저장소(예 : Docker Hub)이다. 사용자는 Registry에서 이미지를 pull 하여 로컬 Docker Host에 가져올 수 있으며, 이 이미지를 기반으로 컨테이너를 실행한다

Docker Image

Docker 이미지란 애플리케이션을 실행하는데 필요한 소스 코드, 라이브러리, 실행 파일, 종속성, 설정 값, 도구 및 등 모든 요소를 하나로 묶은 패키지이다. 즉, 특정 애플리케이션이 정상적으로 동작하기 위한 환경을 포함하는 실행 가능한 템플릿이라고 할 수 있다.

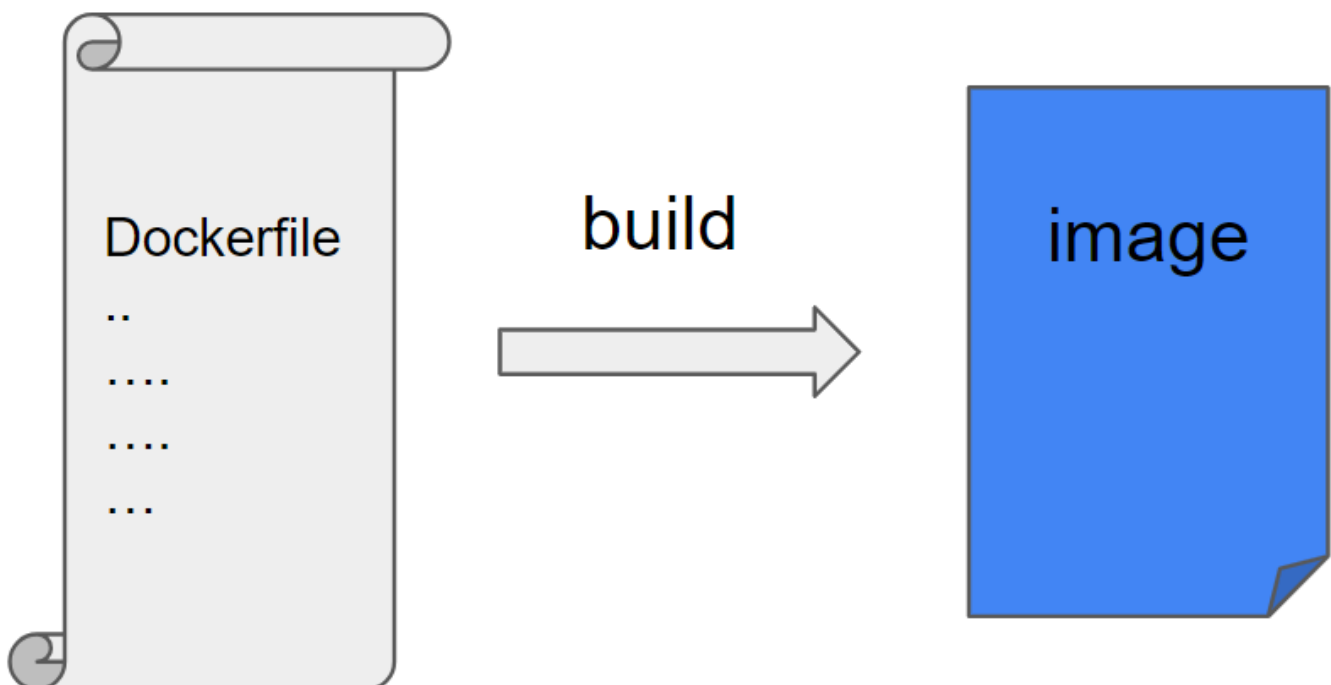
Docker 이미지는 불변(immutable) 특성을 가지며, 읽기 전용으로 유지된다. 이 때문에 스냅샷과 비슷하게 특정 시점의 애플리케이션과 환경 상태를 나타낸다. 또한 이미지는 버전 관리가 가능하며, GitHub와 유사한 개념으로 운영되는 Docker Hub와 같은 중앙 저장소에서 이미지 배포와 공유가 이루어진다. 하지만 이미지는 템플릿일 뿐, 직접 실행할 수는 없다. 실제로 실행되어 동작하는 것은 이 이미지를 기반으로 생성된 컨테이너(Container)이다.

Dockerfile

Dockerfile은 도커 이미지를 만들기 위해 사용하는 설정 스크립트 파일이다. 텍스트 형식으로 작성되며, 이미지 생성 과정에서 실행할 명령어들을 순서대로 나열한다. 이 파일을 통해 애플리케이션 환경 구축을 자동화하고 문서화할 수 있어, 이미지 빌드의 재현성과 일관성을 보장한다.

수동으로 이미지를 만들 필요 없이, Dockerfile 한 파일로 빌드 과정을 자동화 할 수 있으며, 텍스트 파일이기 때문에 Git과 같은 버전 관리 시스템으로도 관리할 수 있다.

또한, 이미지 빌드 과정이 명확히 기록되어 다른 사람도 동일한 환경을 쉽게 재현할 수 있다는 점이 주요 특징이다.

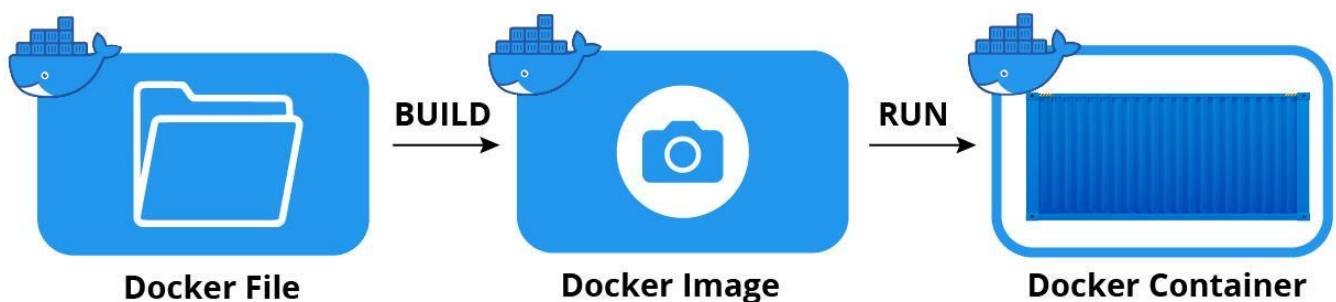


노커파일을 살펴봄으로써 이미지를 만드는 방법을 수중에 자세히 알아보자.

Docker Container

Docker container는 도커 이미지를 실행한 실행 중인 상태의 인스턴스이다. 즉 이미지를 기반으로 생성되어 독립적인 프로세스와 격리된 환경에서 실행되는 애플리케이션 단위이다. 컨테이너는 가볍고 빠르며, 호스트 운영체제의 커널을 공유하면서도 독립적인 사용자 공간을 제공한다.

도커 컨테이너는 도커 이미지로부터 생성되며, 이미지와 컨테이너는 1:N 관계를 가진다. 즉, 하나의 이미지로 여러 개의 컨테이너를 생성할 수 있다.



이미지와 컨테이너의 관계는 운영체제에서 프로그램과 프로세스, 객체지향 프로그래밍에서 클래스와 인스턴스의 관계와 비슷하다고 생각하면 이해하기 쉽다.

정리하면, 도커 파일을 이용해 정적인 도커 이미지를 빌드(build)하고, 이 이미지를 실행(run)함으로써 여러 개의 도커 컨테이너를 만들 수 있다. 대부분의 도커 컨테이너는 생성에 사용된 도커 이미지에 따라 알맞은 설정과 파일을 갖추고 있어, 이미지의 목적에 맞게 활용되는 것이 일반적이다.

4-2. WSL2 설치하기

WSL2란?

WSL2(Windows Subsystem for Linux 2)는 Windows에서 리눅스 배포판을 네이티브 수준으로 실행할 수 있도록 마이크로소프트가 개발한 리눅스 호환 계층이다. 기존의 WSL1은 리눅스 시스템 콜을 윈도우 커널이 번역해 흉내내는 방식이었기 때문에 호환성에 한계가 많았고, 이를 개선한 WSL2는 실제 리눅스 커널을 경량 가상 머신(VM)에 포함해 실행하는 방식을 채택했다.

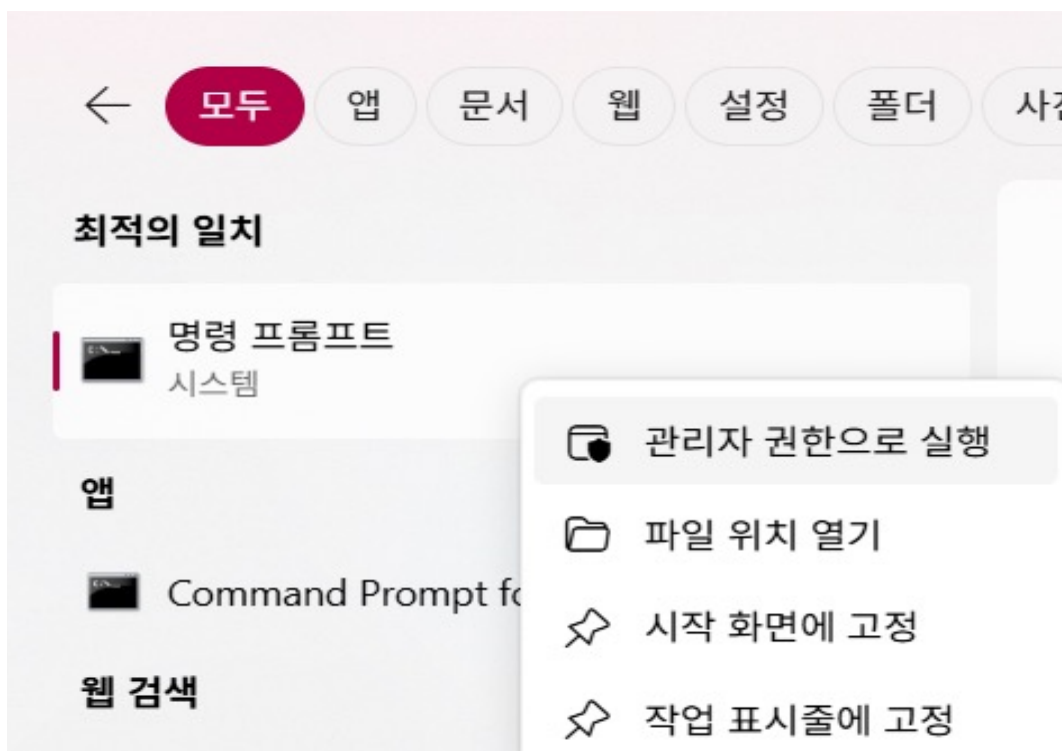
Docker Desktop은 WSL2를 백엔드로 사용하며, WSL2 환경에 Docker Engine을 직접 설치할 수도 있다. 또한 Visual Studio Code에서 WSL 터미널을 활용해 빠르고 편리하게 명령어를 실행할 수 있기 때문에 설치를 권장한다. 다만 Mac 사용자는 이 부분을 건너뛰어도 되며, 윈도우 사용자라도 WSL2를 사용할 계획이 없다면 역시 생략해도 무방하다.



WSL2 설치하기

공식 홈페이지(<https://learn.microsoft.com/ko-kr/windows/wsl/install>)에서 더 자세한 내용을 확인할 수 있으니, 설치가 되지 않는다면 참고 하길 바란다.

제일 먼저 화면 아래에 cmd 또는 powershell를 입력하고 명령 프롬프트를 우클릭하여 관리자 권한으로 실행을 클릭한다.



클릭하면 새로운 터미널 창이 나오는데 여기에 아래와 같은 명령어를 입력한다.

WSL 설치 명령어

```
wsl --install
```

실행이 다 완료되면 컴퓨터를 재부팅한다.

```
Microsoft Windows [Version 10.0.26100.4351]
(c) Microsoft Corporation. All rights reserved.

C:\Windows\System32>wsl --install
다운로드 중: Ubuntu
[ 0.9% ]
```

재부팅 후 아래 명령어로, 현재 내 컴퓨터에 설치된 목록을 확인할 수 있다.

설치된 wsl 목록 명령어

```
wsl --l -v
```

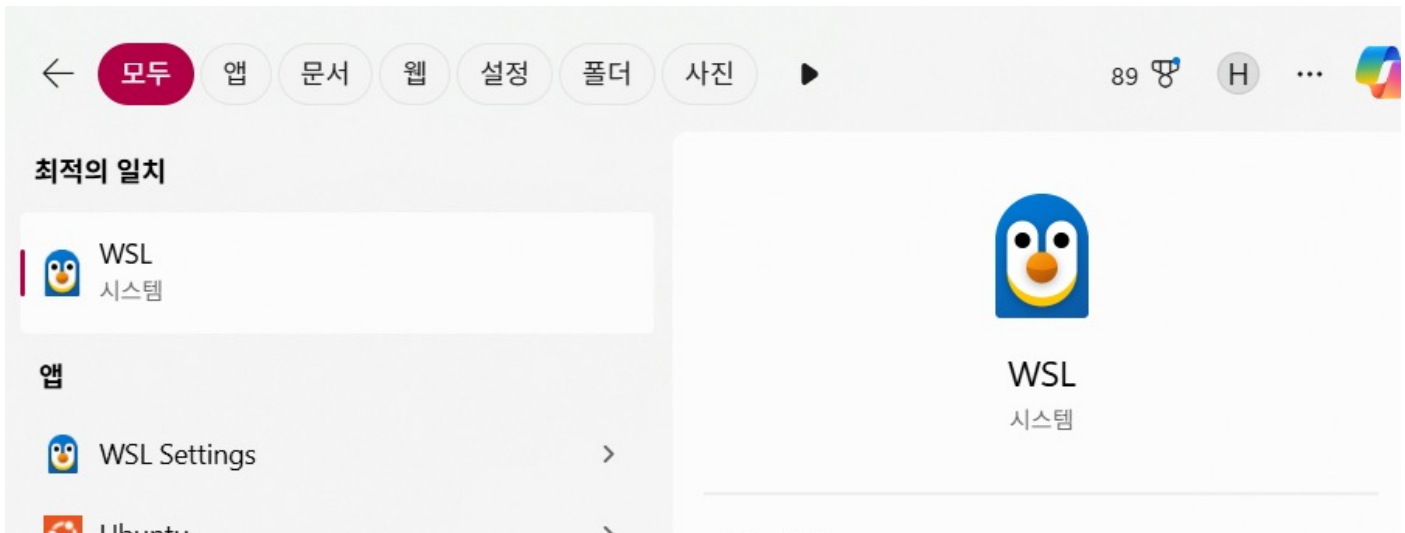
```
C:\Windows\System32>wsl -l -v
  NAME                STATE      VERSION
*  Ubuntu              Stopped    2
   Ubuntu-20.04        Stopped    2

C:\Windows\System32>
```

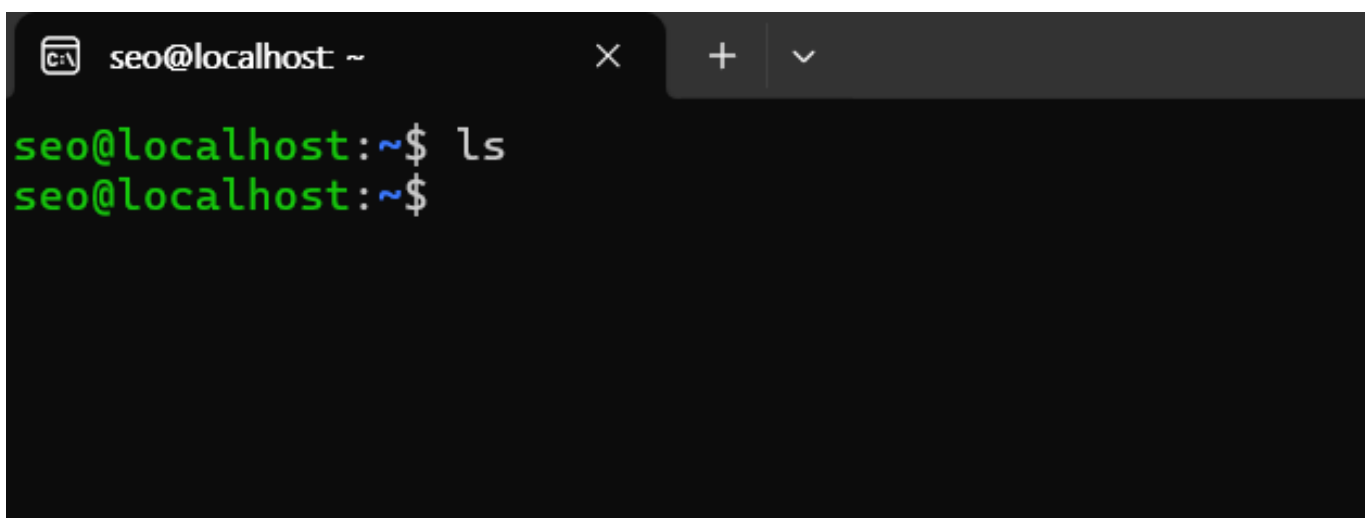
꼭 VERSION 부분이 2인지 확인하자.

Ubuntu Version 2가 적혀있으면 설치가 완료되었다.

제대로 설치되었는지 확인하기 위해서 시작 검색창에다가 'WSL'을 입력하고 실행을 눌러보자.



사용자 이름과 비밀번호를 설정하면 윈도우 환경에서 리눅스를 이용할 수 있는 WSL2가 정상적으로 동작하고, 사용할 수 있다는 것을 확인할 수 있다. Visual Studio Code에서도 터미널을 Ubuntu(WSL)으로 설정하면 해당 터미널 창을 똑같이 이용할 수 있다.

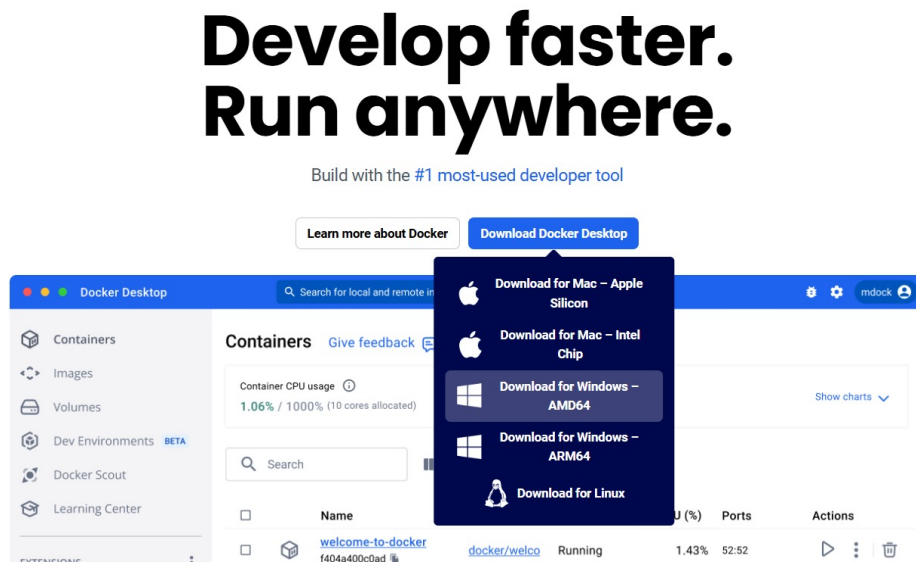


이제 WSL2 가 설치되었으니, 본격적으로 Docker를 설치해보자.

4-3. Docker설치하기

윈도우에서 Docker 설치하기

Docker의 공식 홈페이지(<https://www.docker.com/>)에 접속하면 메인 화면에서 Download Docker Desktop 버튼을 볼 수 있다. 이 버튼을 클릭하면 사용중인 운영체제에 맞는 Docker 설치 파일을 다운로드 할 수 있다.



윈도우 버전에서 Docker를 설치하려면, Download for Windows – AMD64 버튼을 클릭해 설치파일을 내려받으면 된다.



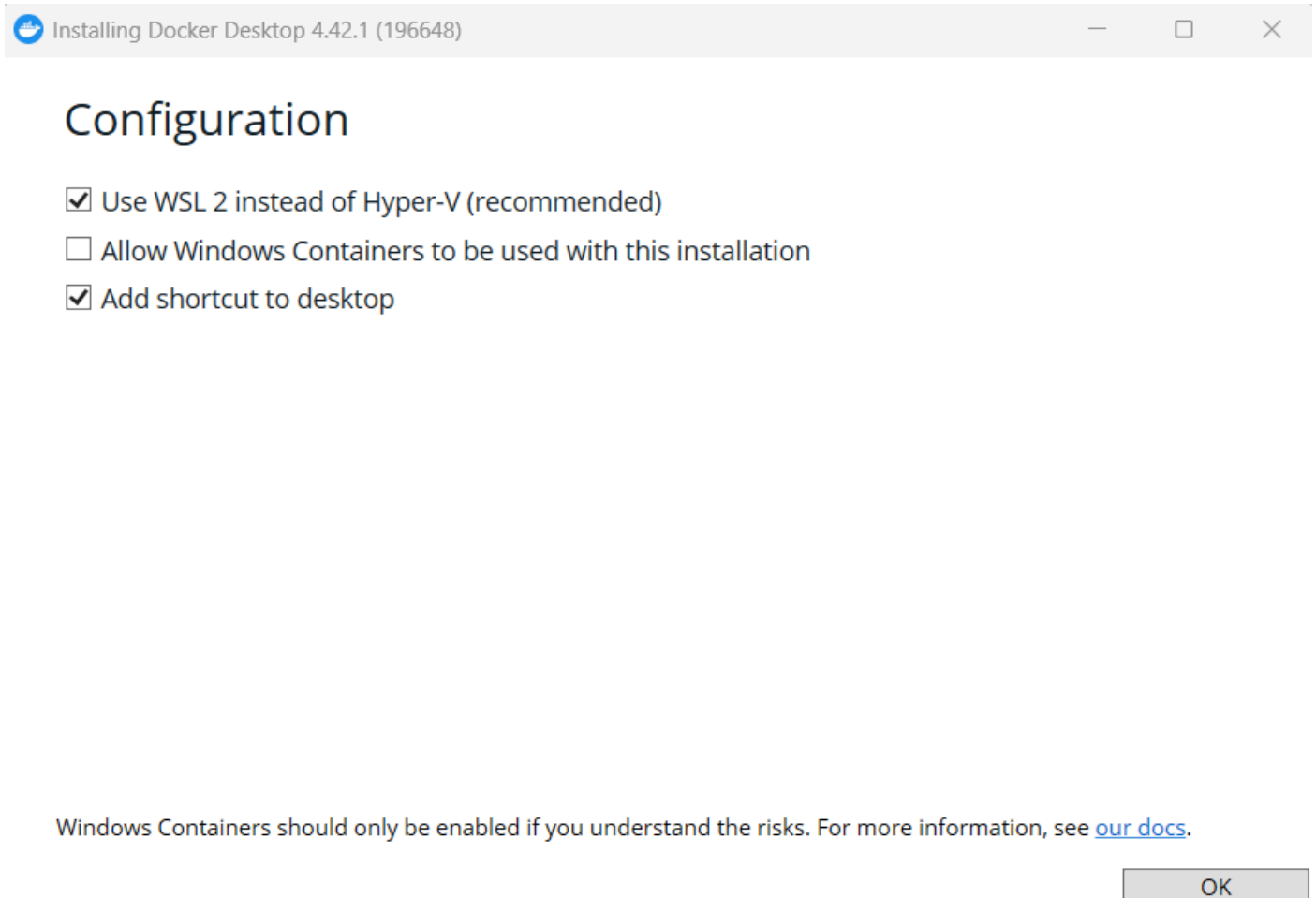
Docker를 설치하고 사용하기 전에 먼저 로그인을 해야 한다. 이미 계정이 있다면 미리 로그인을 해 두고, 계정이 없다면 회원가입을 진행해야 한다.

로그인은 메인 화면 오른쪽 위에 있는 'Sign in' 버튼을 클릭하여 할 수 있고, 회원가입은 로그인 창 하단의 'Sign Up' 버튼을 통해 진행할 수 있다.

The image shows the Docker website's account management interface. On the left is the 'Sign in' section, which includes a text input for 'Username or email address*', a blue 'Continue' button, and three social login options: 'Continue with Google', 'Continue with GitHub', and 'Continue with SSO'. Below these is a link for 'Don't have an account? Sign Up'. On the right is the 'Create your account' section, with tabs for 'Work' and 'Personal' (selected). It features a red-outlined 'Email' input field with a red error message 'Email is required.', a 'Username' input field with a note 'Use 4 to 30 letters & digits only.', and a 'Password' input field with an eye icon. There is an unchecked checkbox for 'Send me occasional product updates and announcements.' and a greyed-out 'Sign up' button. At the bottom, there are social login buttons for Google and GitHub, and a link for 'Already have an account? Sign in'.

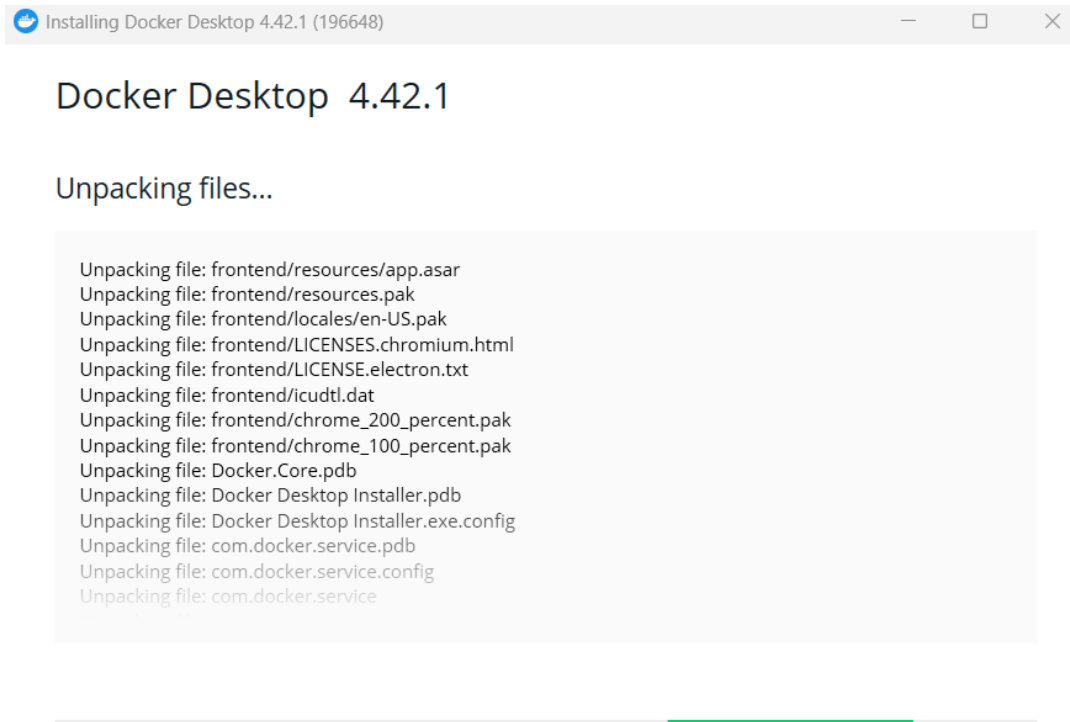
Docker를 사용하려면 반드시 계정이 필요하므로, 미리 로그인해 두는 것이 좋다. 또한, GitHub나 Google 계정을 통해서도 회원가입과 로그인이 가능하다.

Docker 설치 파일을 진행하면 설치를 진행한다.

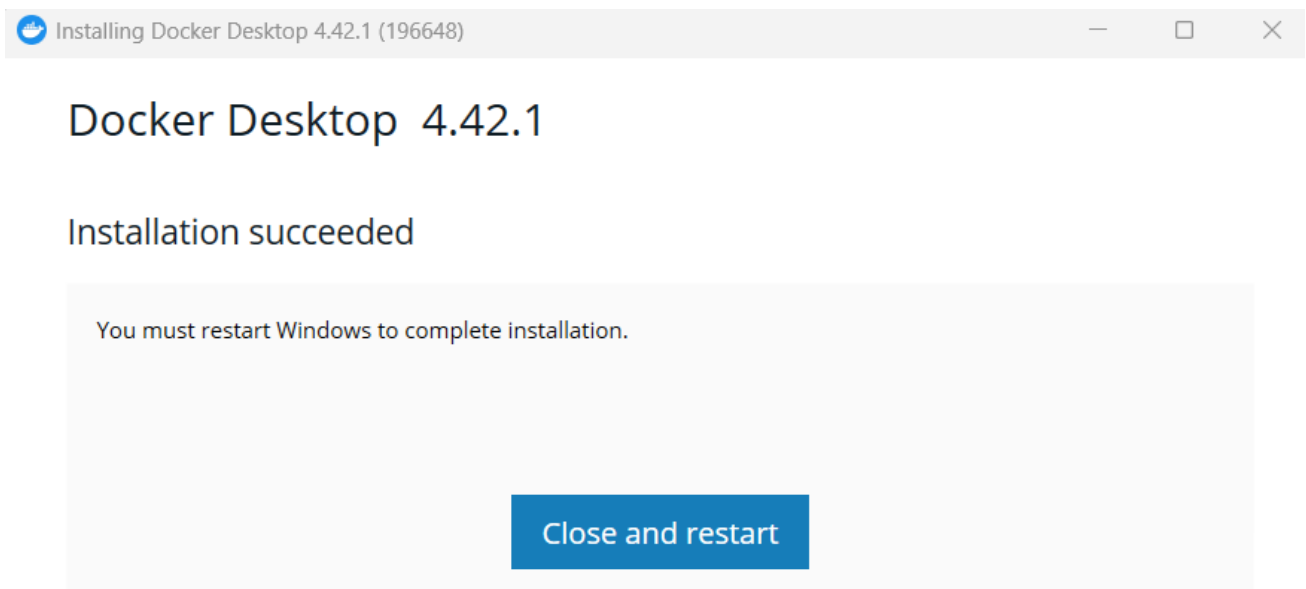


설치 과정 중 나타나는 Configuration 항목에서는 맨 위의 ‘Use WSL 2 instead of Hyper-V (recommended)’와 ‘Add shortcut to desktop’ 옵션을 선택한 상태로 진행하는 것을 권장한다. 첫 번째 옵션은 ‘Hyper-V 대신 WSL를 사용하길 권장한다’는 의미이며, WSL 2는 Windows에서 리눅스 커널을 직접 실행하는 경량화 된 가상화 환경으로, 기존 Hyper-V에 비해 빠른 성능과 더 나은 호환성을 제공한다. 마지막 옵션은 바탕화면에 바로가기 추가를 의미한다. 따라서, 설치 시 위 두 옵션을 선택하여 진행하는 것이 Docker를 보다 원활하게 사용할 수 있는 방법이다.

Unpacking files 구간에서 시간이 조금 걸릴 수 있으니 조금만 기다리면 설치가 완료된다. 그러니 인내심을 갖고 기다리도록 하자.



설치가 완료되면 컴퓨터 재시작을 요청하는 버튼이 나타난다. 해당 버튼을 클릭하여 재부팅하면 Docker를 정상적으로 사용할 수 있다.

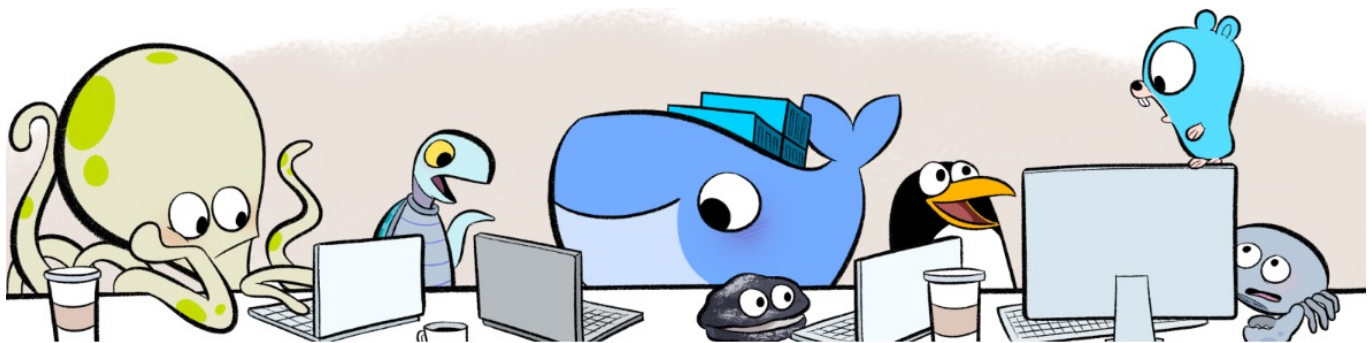


재부팅이 완료되면 Docker Desktop이 자동으로 실행되고, Docker 구독 서비스 계약에 동의하는지를 묻는 창이 나타난다.

‘Accept’을 선택하면 구독 서비스 계약, Docker 데이터 처리 계약, 데이터 개인정보 보호 정책에 모두 동의하는 것으로 간주된다.

상업적으로 Docker Desktop을 이용하는 규모가 큰 회사(직원 250명 이상 또는 연매출 1천만 달러 이상)라면 유료 구독이 필요하다는 안내도 함께 표시된다.

하지만 개인 사용자인 우리는 이 부분은 신경 쓰지 않아도 되며, ‘Accept’ 또는 ‘Close’ 버튼을 눌러 원하는 대로 진행하면 된다.



Docker Subscription Service Agreement

By selecting **accept**, you agree to the [Subscription Service Agreement](#), the [Docker Data Processing Agreement](#), and the [Data Privacy Policy](#).

Commercial use of Docker Desktop at a company of more than 250 employees OR more than \$10 million in annual revenue requires a paid subscription (Pro, Team, or Business). [See subscription details](#)

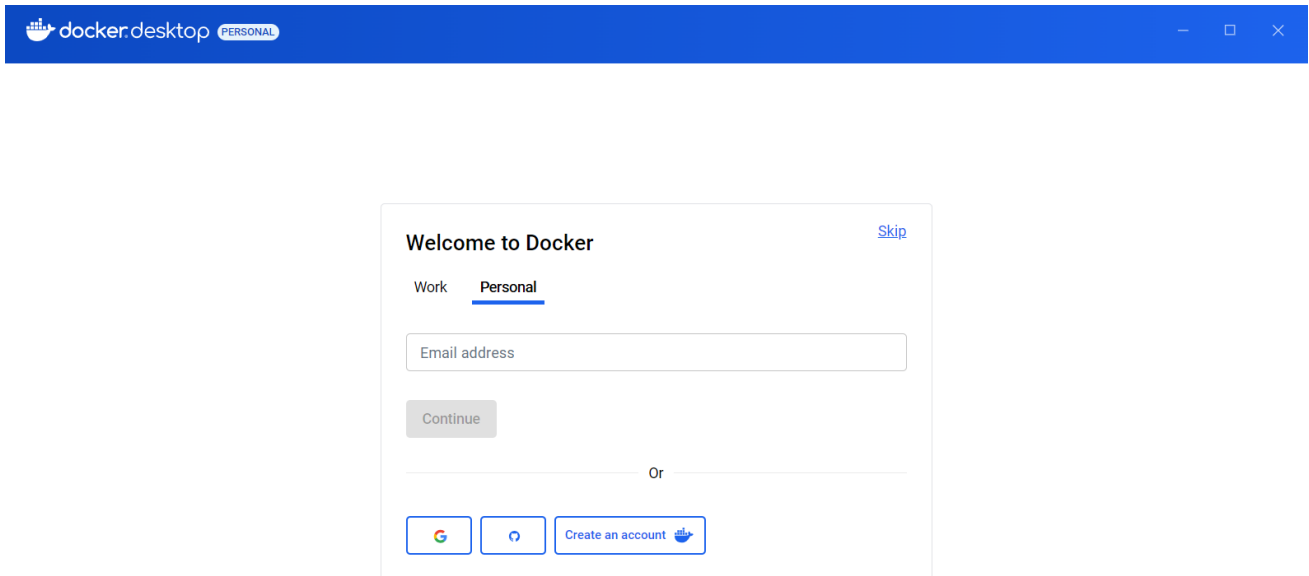
[View Full Terms](#)

Accept

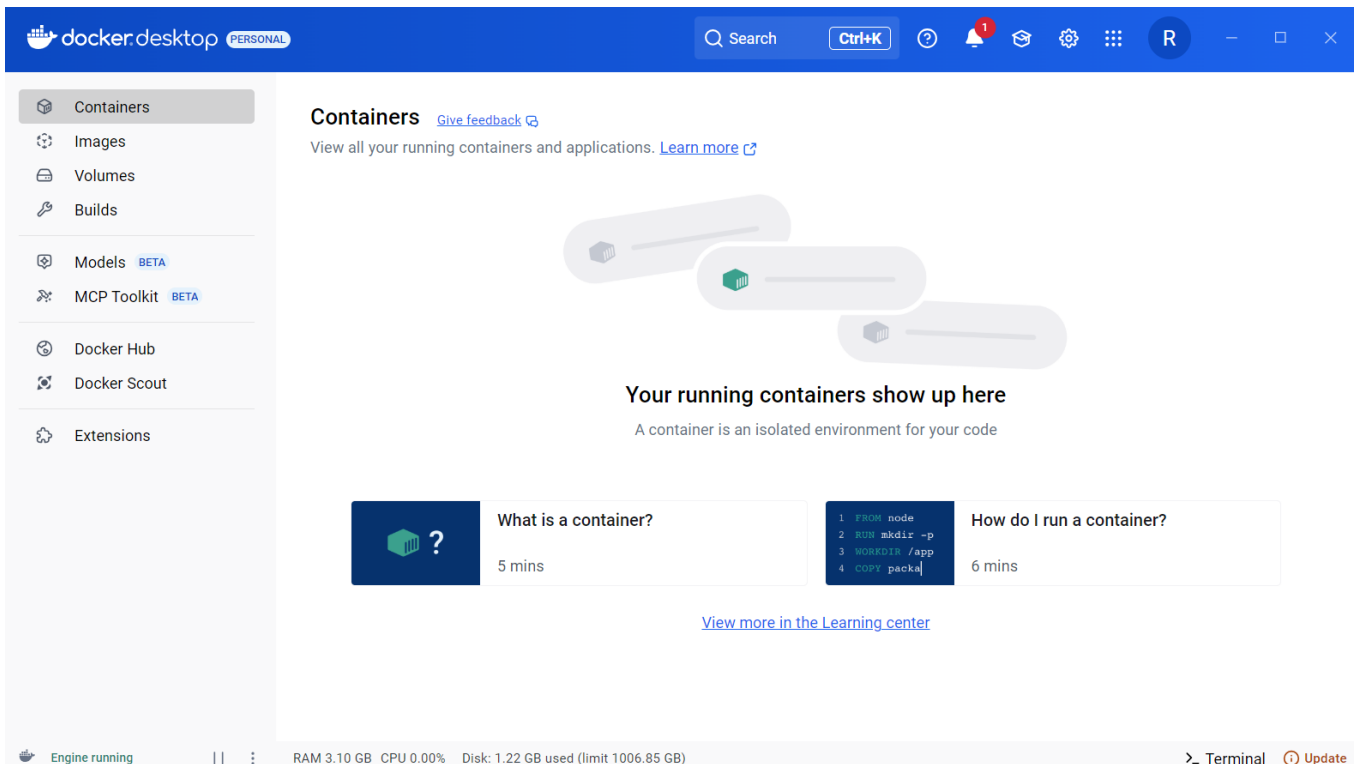
Close

우리는 개인 사용자이고, 앞으로 Docker의 업데이트나 기능을 모두 원활하게 사용하기 위해서 ‘Accept’ 버튼을 클릭한다.

그리고 Personal부분에 들어가서 로그인을 진행한다.

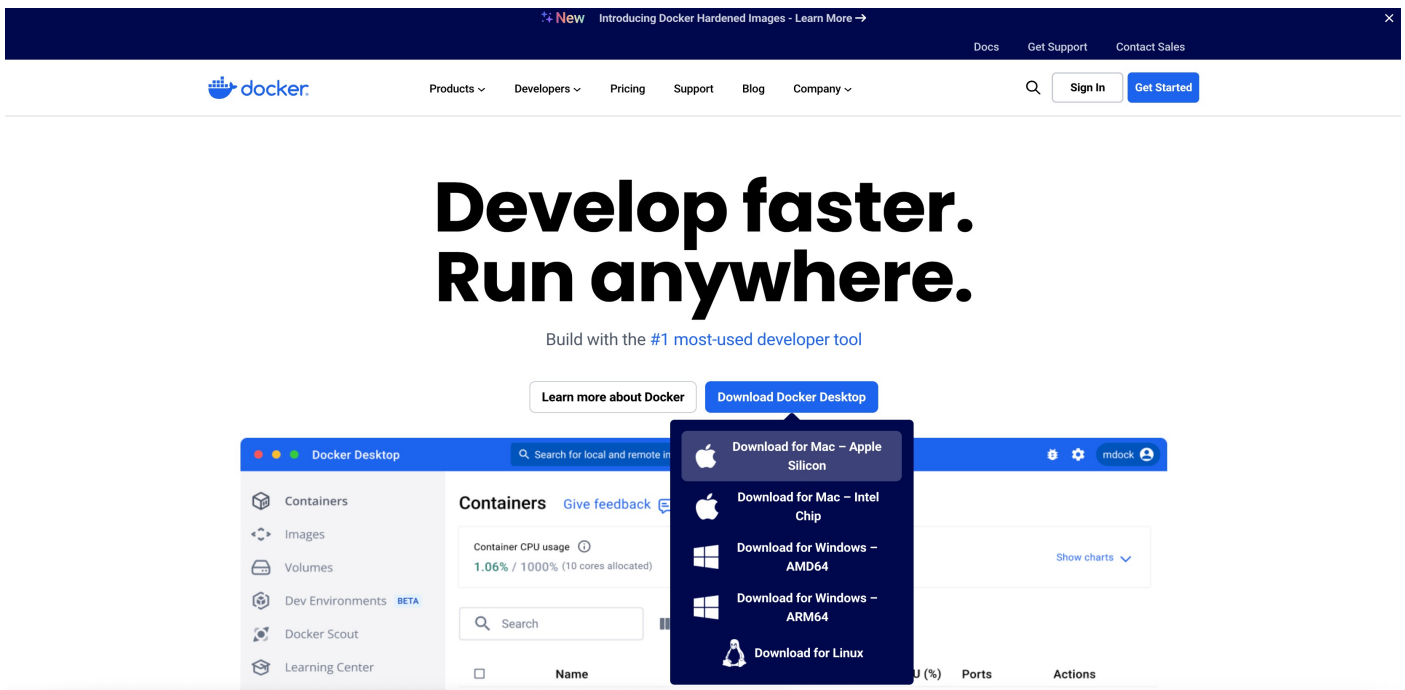


로그인을 모두 마지면 Docker를 정상적으로 이용할 수 있는 환경이 구성되었다.

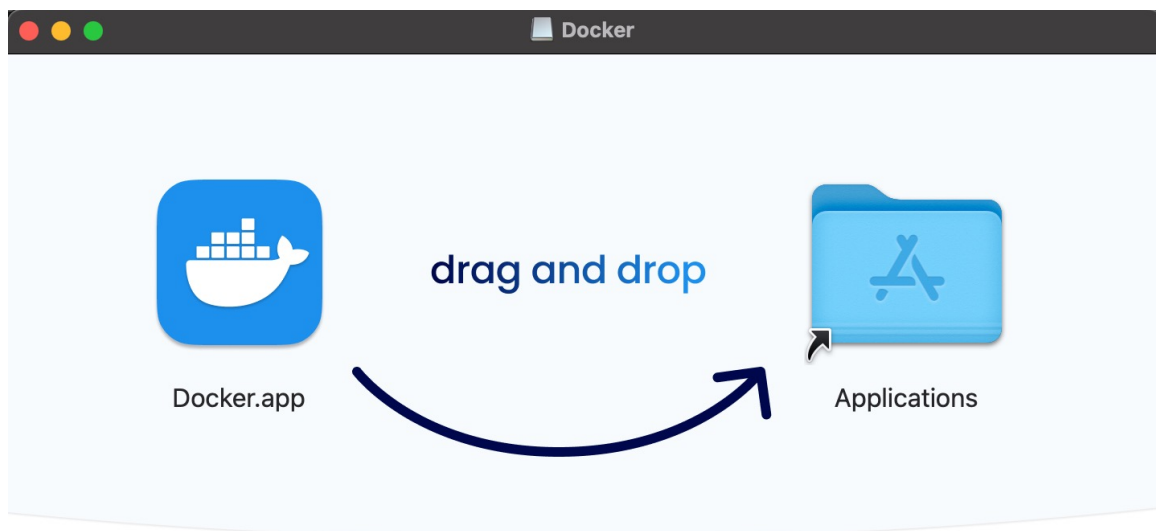


Mac OS에서 Docker 설치하기(GUI)

Docker의 공식 홈페이지(<https://www.docker.com/>)에 접속하면 메인 화면에서 Download Docker Desktop 버튼을 볼 수 있다. 이 버튼을 클릭하면 사용중인 운영체제에 맞는 Docker 설치 파일을 다운로드 할 수 있다.



다운로드한 .dmg 파일을 실행하여 Docker.app을 응용 프로그램으로 복사해준다. 그리고 응용프로그램에서 Docker.app을 실행한다.

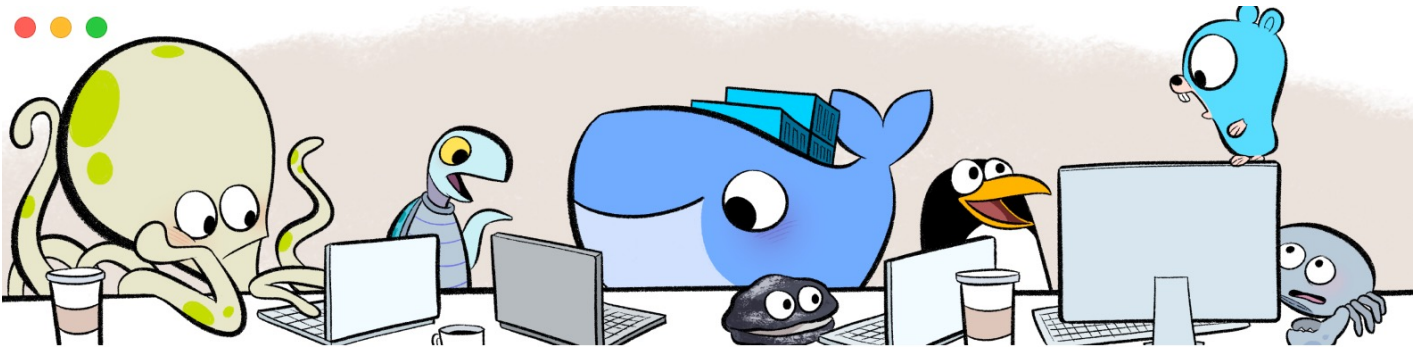


재부팅이 완료되면 Docker Desktop이 자동으로 실행되고, Docker 구독 서비스 계약에 동의하는지를 묻는 창이 나타난다.

‘Accept’을 선택하면 구독 서비스 계약, Docker 데이터 처리 계약, 데이터 개인정보 보호 정책에 모두 동의하는 것으로 간주된다.

상업적으로 Docker Desktop을 이용하는 규모가 큰 회사(직원 250명 이상 또는 연매출 1천만 달러 이상)라면 유료 구독이 필요하다는 안내도 함께 표시된다.

하지만 개인 사용자인 우리는 이 부분은 신경 쓰지 않아도 되며, ‘Accept’ 또는 ‘Close’ 버튼을 눌러 원하는 대로 진행하면 된다.



Docker Subscription Service Agreement

By selecting **accept**, you agree to the [Subscription Service Agreement](#), the [Docker Data Processing Agreement](#), and the [Data Privacy Policy](#).

Commercial use of Docker Desktop at a company of more than 250 employees OR more than \$10 million in annual revenue requires a paid subscription (Pro, Team, or Business). [See subscription details](#)

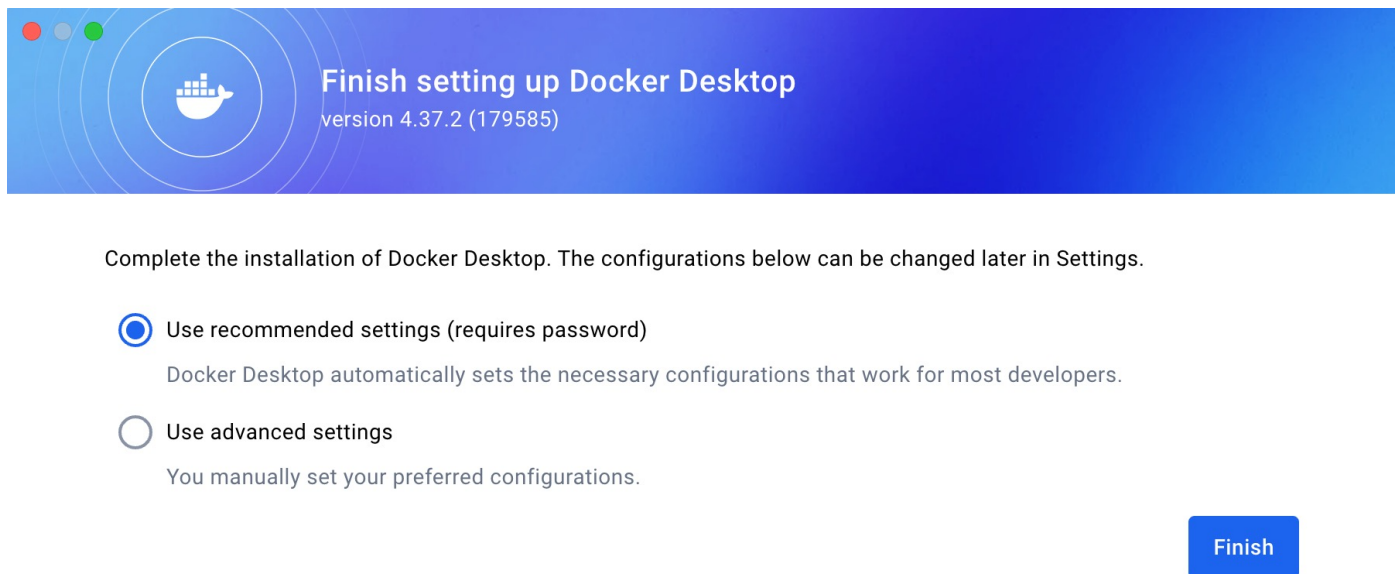
[View Full Terms](#)

Accept

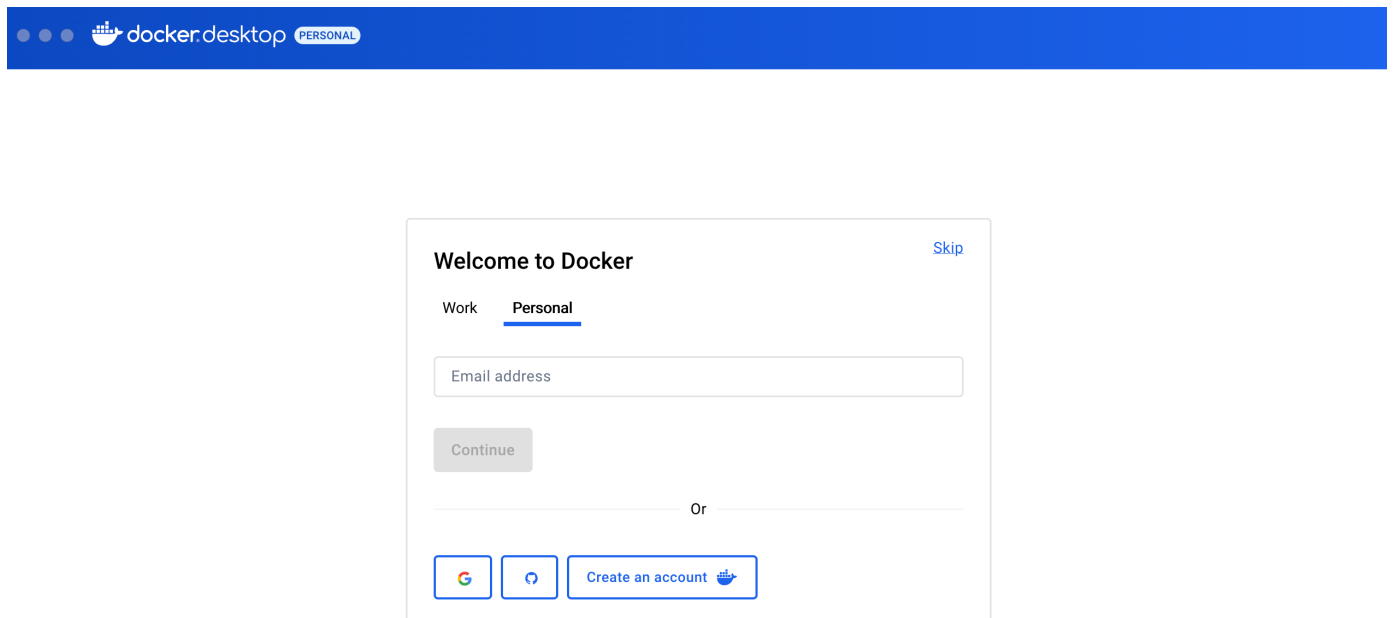
Close

우리는 개인 사용자이고, 앞으로 Docker의 업데이트나 기능을 모두 원활하게 사용하기 위해서 ‘Accept’ 버튼을 클릭한다.

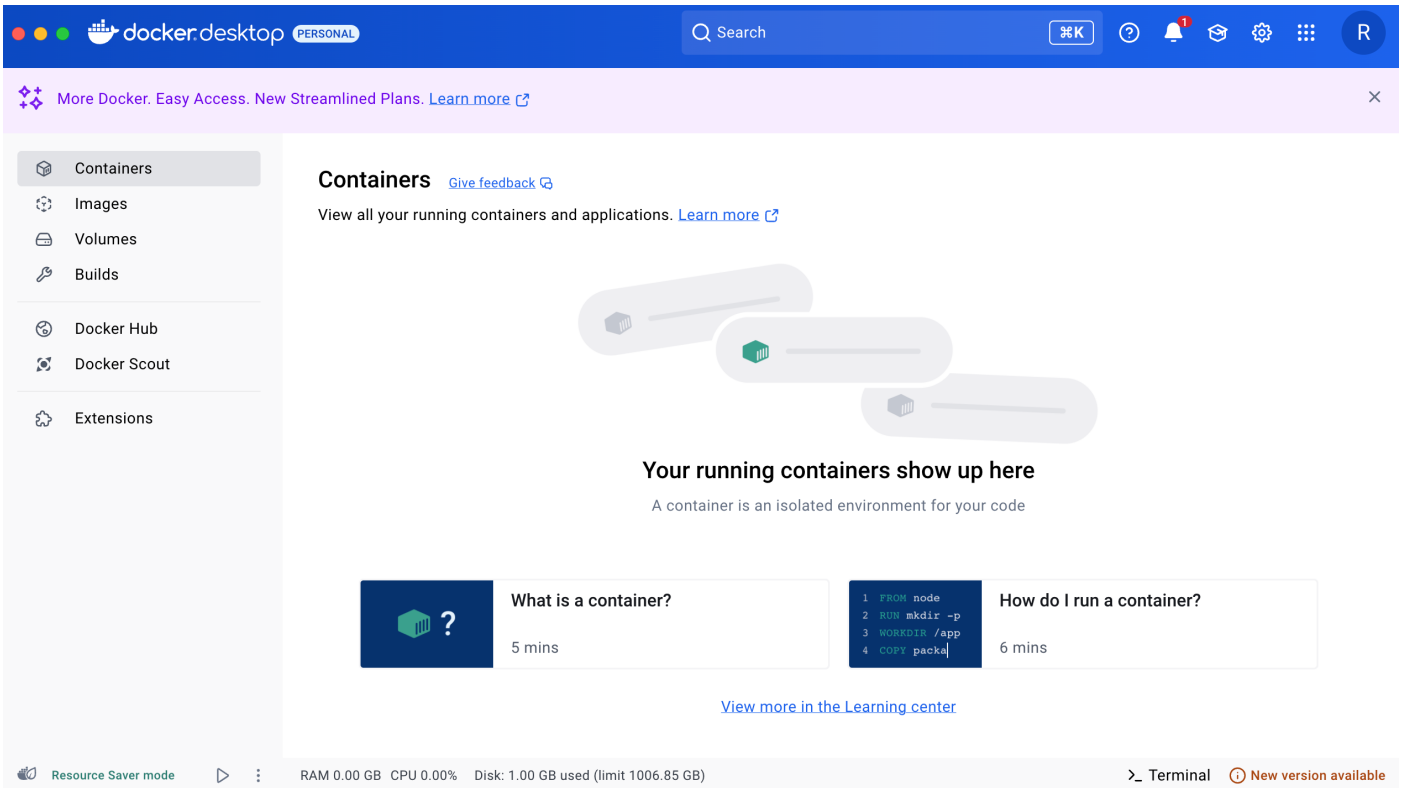
Docker 세팅을 마무리 하기 전에 또 하나의 선택창이 나온다. Use recommended settings (requires password)를 클릭하여 세팅을 마저 마무리 하자.



그리고 Personal부분에 들어가서 로그인을 진행한다.



로그인을 모두 마지면 Docker를 정상적으로 이용할 수 있는 환경이 구성되었다.



docker --version 명령어로 정상 설치 여부를 확인할 수 있다

```
seo ~ docker
Usage: docker [OPTIONS] COMMAND

A self-sufficient runtime for containers

Common Commands:
run      Create and run a new container from an image
exec     Execute a command in a running container
ps       List containers
build    Build an image from a Dockerfile
pull     Download an image from a registry
push     Upload an image to a registry
images   List images
login    Authenticate to a registry
logout   Log out from a registry
search   Search Docker Hub for images
version  Show the Docker version information
info     Display system-wide information

Management Commands:
ai*      Ask Gordon - Docker Agent
builder  Manage builds
buildx*  Docker Buildx
compose* Docker Compose
container Manage containers
context  Manage contexts
debug*   Get a shell into any image or container
```

4-4. Docker 실행 흐름

프로로그

이제부터는 도커 명령어를 입력했을 때, 도커가 내부에서 어떻게 동작하고 처리하는지 간단히 살펴보자. 명령어를 배우기 전에 도커의 전체 동작 흐름을 이해하는 것이 중요하다. 아래 그림을 참고하며 천천히 과정을 확인해보자.

```
x seo ~ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
c9c5fd25a1bd: Pull complete
Digest: sha256:940c619fbd418f9b2b1b63e25d8861f9cc1b46e3fc8b018ccfe8b78f19b8cc4f
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm64v8)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
```

seo ~ █

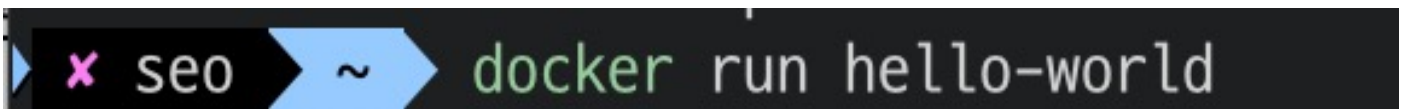
1. 명령어 입력(Docker Client)

사용자는 터미널 또는 커맨드라인 인터페이스(CLI)에 아래와 같은 도커 명령어를 입력한다.

도커 명령어 입력

```
docker run hello-world
```

이 명령어는 도커 클라이언트(Docker Client)를 통해 도커 엔진에 전달된다. 도커 클라이언트는 사용자와 상호작용하는 인터페이스 역할을 한다.



위 명령어는 'hello-world' 라는 이미지를 사용해 새로운 컨테이너를 생성하고 실행하라는 뜻이다.

2. 명령 전달 및 처리(Docker Daemon)

도커 클라이언트는 입력된 명령을 도커 데몬(Docker Daemon)에 전송한다. 도커 데몬은 백그라운드에서 실행중인 서버 프로세스로, 컨테이너 생성, 이미지 관리, 네트워크 설정 등 모든 실제 작업을 수행한다

3. 이미지 캐시 확인

도커는 먼저 로컬 시스템(사용자 컴퓨터)에 해당 이미지가 이미 존재하는지 확인한다. 로컬에 저장된 이미지들은 이미지 캐시에 보관되어 있으며, 도커 클라이언트가 이미지를 캐시한다. 만약 이전에 해당 이미지를 사용한 적이 없다면, 이미지 캐시에서 찾지 못하고 아래와 같은 메시지가 출력된다.

```
seo ~ docker run hello-world
Unable to find image 'hello-world:latest' locally
```

4. 도커 허브에서 이미지 가져오기

이미지 캐시에 해당 이미지가 없으면, 도커 허브에서 이미지를 다운로드 한다. 도커 허브는 도커 이미지의 공식 원격 저장소로, 필요한 이미지를 다운로드(pull) 할 수 있다. 도커 허브에서 이미지를 가져오면 로컬에 저장되며, 도커 데스크톱의 이미지 목록에서 확인할 수 있다. 만약 이미지 캐시에 존재한다면 이 과정은 생략된다.

아래 사진은 도커 허브에서 hello-world 이미지를 가져오는 과정이다.

```
latest: Pulling from library/hello-world
c9c5fd25a1bd: Pull complete
Digest: sha256:940c619fbd418f9b2b1b63e25d8861f9cc1b46e3fc8b018ccfe8b78f19b8cc4f
Status: Downloaded newer image for hello-world:latest
```

hello-world 뒤에 별도의 버전 태그가 없으므로, 도커는 최신 버전(latest)을 가져온다. 다음으로 c9c5fd25a1bd라는 고유한 이미지 레이어를 다운로드 한다. 마지막으로 Digest(이미지 해시값)를 확인하여 이미지 무결성을 검증하고, 이미지가 정확한지 확인한다.

과정이 정상적으로 수행되면 도커 데스크톱의 이미지 목록에 아래와 같이 hello-world가 추가된 것을 확인할 수 있다.

Images [Give feedback](#)

View and manage your local and Docker Hub images. [Learn more](#)

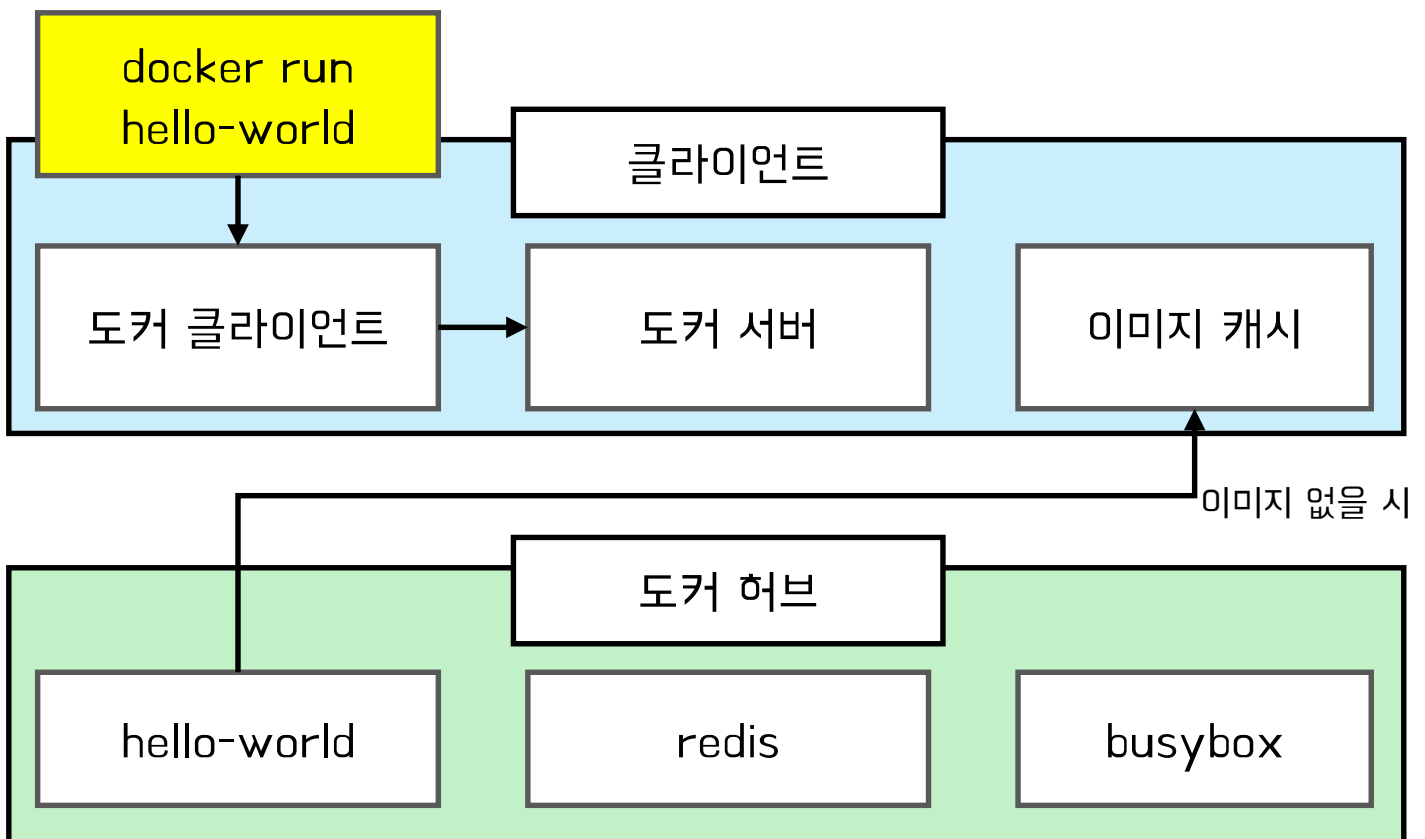
Local Hub repositories

5.2 KB / 5.2 KB in use 1 images

Last refresh: 2 hours ago ↻

Q Search						
☐	Name	Tag	Image ID	Created	Size	Actions
☐	hello-world	latest	f1f77a0f96b7	6 months ago	5.2 KB	▶ ⋮ 🗑

지금까지의 흐름을 정리하면 다음과 같다.



5. 컨테이너 생성 및 실행

컨테이너가 생성되어 실행된다. 방금 입력한 명령어로 happy_germain이라는 이름의 컨테이너가 생성되었다.

The screenshot shows the Docker Desktop interface. On the left is a sidebar with navigation options: Containers, Images, Volumes, Builds, Docker Hub, Docker Scout, and Extensions. The main area is titled 'Containers' and includes a 'Give feedback' link. Below the title, it says 'View all your running containers and applications. Learn more'. There are two status indicators: 'Container CPU usage' and 'Container memory usage', both showing 'No containers are running.' and a 'Show charts' link. A search bar and a toggle switch 'Only show running containers' are present. Below this is a table of running containers.

☐	Name	Container ID	Image	Port(s)	CPU (%)	Last state	Actions
☐	happy_germain	e3473fc70fdb	hello-world		N/A	42 minutes ago	▶ ⋮ 🗑️

hello-world라는 이미지는 아래와 같이 간단한 인사 메시지를 출력하는 이미지이므로, 정상적으로 실행되었는지 확인할 수 있다.

```

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
   (arm64v8)
3. The Docker daemon created a new container from that image which runs the
   executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
   to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/

For more examples and ideas, visit:
https://docs.docker.com/get-started/
  
```

4-5. Docker 생명 주기와 명령어

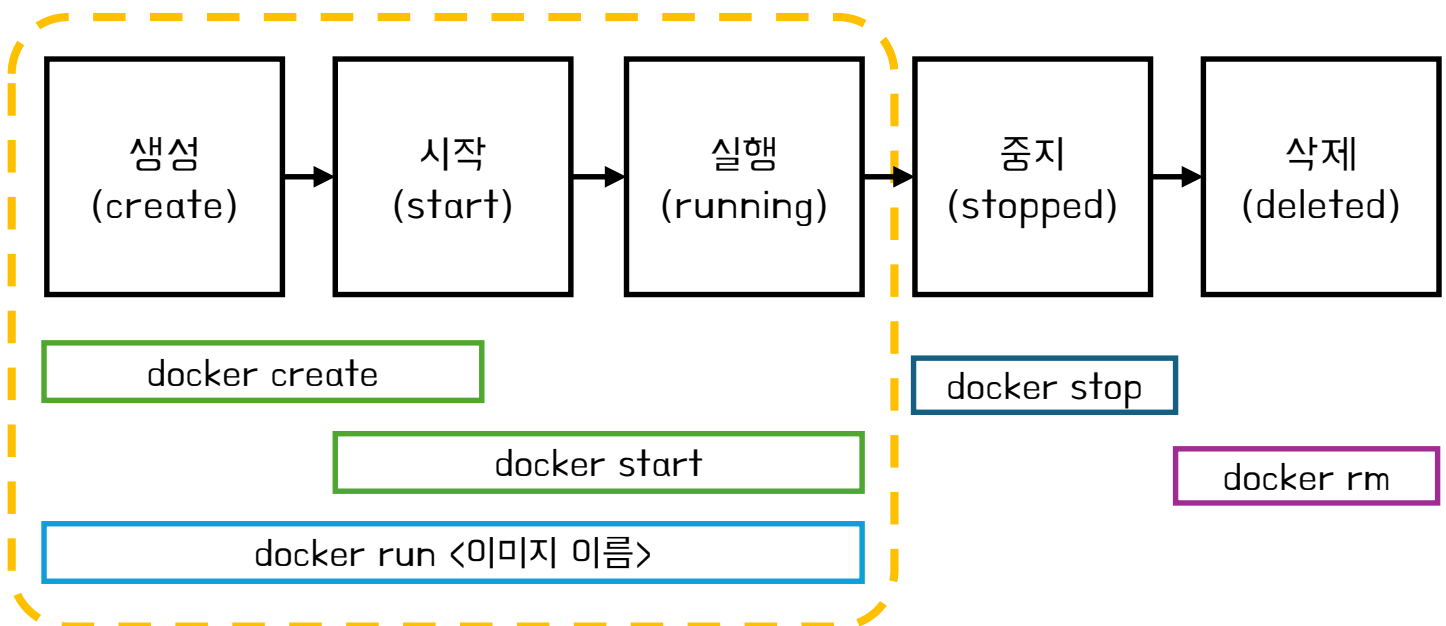
Docker 컨테이너 생명주기(Lifecycle)

Docker 컨테이너 생명주기란 컨테이너가 생성되어 실행되고, 중지되며, 삭제되기까지의 전체 과정을 의미한다. 컨테이너는 생성(Create), 시작(Start), 실행(Running), 중지(Stop), 삭제(Remove) 와 같은 여러 상태를 거치며 운영된다.

이 생명주기 과정을 통해 컨테이너의 상태를 관리하고, 필요한 시점에 컨테이너를 제어할 수 있다. 각 단계는 도커 명령어로 손쉽게 조작 가능하며, 효율적인 리소스 활용과 안정적인 서비스 운영에 필수적이다.

아래 그림은 도커 생명주기에 대한 전반적인 과정이다.

생명 주기



docker create

docker create 명령어는 도커 이미지를 기반으로 새로운 컨테이너를 생성만 하는 명령어이다. 이 명령어는 컨테이너를 생성하지만, 생성된 컨테이너는 즉시 실행하지 않고 중지 상태로 존재한다.

docker create 명령어

docker create (옵션) [이미지 이름]

```
seo ~ docker create hello-world
18716a95514f9f697907a0a88511fb913fbe29ea97e4dfe0a68db459fc3afa59
```

해당 명령어를 수행하면, 이미지로부터 컨테이너 하나가 만들어지고, 컨테이너 ID 값을 출력해준다.

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)

Container CPU usage ⓘ No containers are running.		Container memory usage ⓘ No containers are running.		Show charts		
Search		☒ Only show running containers				
Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
romantic_lampo	18716a95514f	hello-world		N/A		▶ ⋮ 🗑

이름은 기본적으로 랜덤하게 설정되기 때문에, 컨테이너 이름을 지정하려면 --name 옵션을 사용한다.

docker create 명령어

docker create --name [컨테이너 이름] [이미지 이름]

--name 옵션을 부여하여 seominjae 라는 이름의 컨테이너를 생성하였다,

```
seo ~ docker create --name seominjae hello-world
3054dc859c553f9e413e718748e3f120a479e9bac0f62c75c087bdc55acf6d77
```

도커 데스크톱을 확인해보면 seominjae라는 이름의 컨테이너가 생성된 것을 확인할 수 있다. 출력된 컨테이너 ID 값으로 방금 생성된 컨테이너임을 알 수 있다.

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)

Container CPU usage ⓘ

No containers are running.

Container memory usage ⓘ

No containers are running.

[Show charts](#)

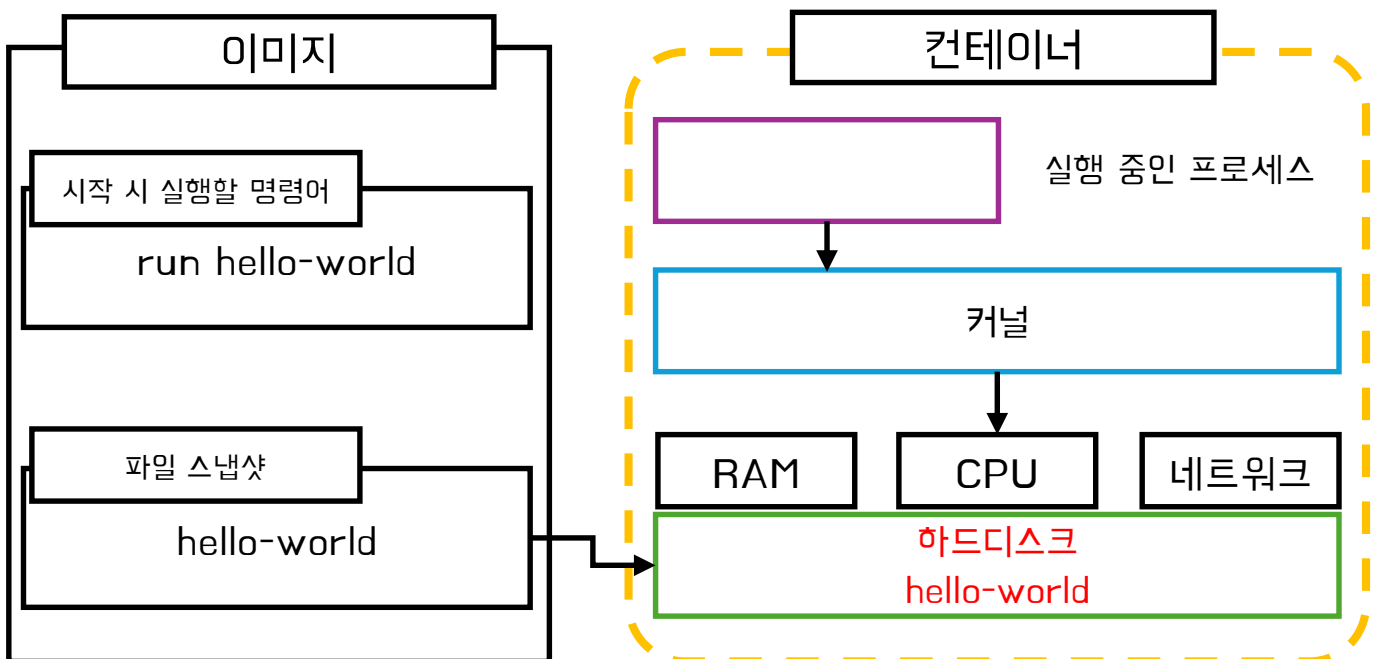
Q Search



Only show running containers

Name	Container ID	Image	Port(s)	CPU (%)	Last started	Actions
romantic_lampo	18716a95514f	hello-world		N/A		▶ ⋮ 🗑
seominjae	3054dc859c55	hello-world		N/A		▶ ⋮ 🗑

그림으로 과정을 표현하면 다음과 같다. 파일 스냅샷을 하드디스크에 저장하는 과정까지가 docker create의 과정이다.



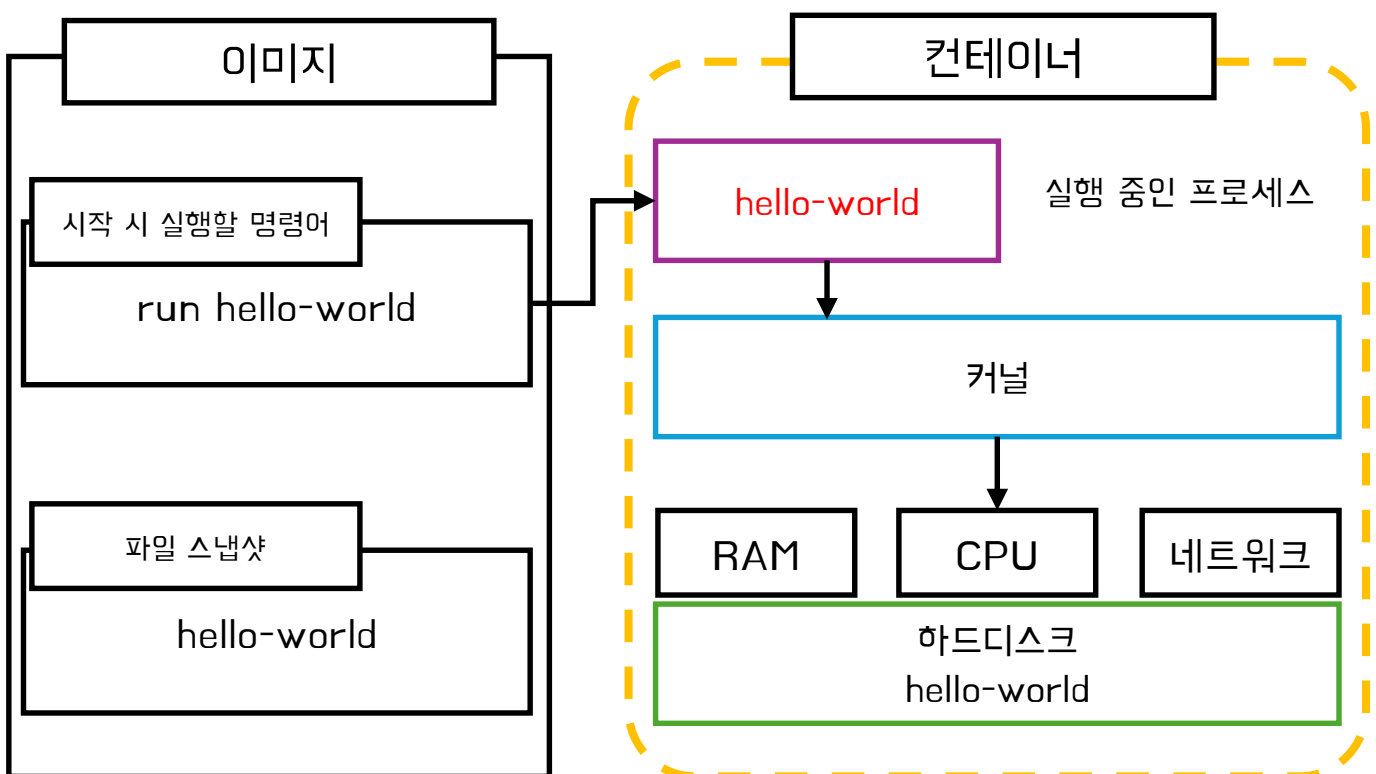
docker start

docker start 명령어는 이미 생성되어 있지만 중지된 컨테이너를 다시 실행하는 명령어이다. 이 명령어는 컨테이너를 생성하는 것이 아니라, 기존의 컨테이너를 시작하여 내부 애플리케이션과 프로세스를 동작 상태로 만든다. 따라서 컨테이너가 종료되거나 중지된 상태일 때만 사용 가능하며, 컨테이너가 실행 중일 경우 명령어가 무시되거나 오류가 발생할 수 있다.

docker start 명령어

docker start (옵션) [컨테이너 아이디/컨테이너 이름]

그림으로 과정을 표현하면, 시작 명령어가 실행 중인 프로세스에 전달되어 컨테이너가 실행되는 상태가 된다.



```
seo ~ docker start 3054dc859c553f9e413e718748e3f120a479e9bac0f62c75c087bdc55acf6d77
3054dc859c553f9e413e718748e3f120a479e9bac0f62c75c087bdc55acf6d77
seo ~ docker start seominjae
seominjae
```

위 사진은 전체 실행 결과이다. 앞서 본 Docker 실행 흐름의 출력 결과와는 다르게, 현재 실행된 결과에서는 별도의 출력물이 나타나지 않았다. 컨테이너에서 실행 중인 프로세스의 출력 내용을 보기 위해서는 `-a` 옵션을 추가해야 한다. `-a` 옵션은 `attach`의 약자로, 해당 컨테이너에 계속 붙어 있어 출력 결과물까지 실시간으로 보여주는 옵션이다.

또한, 컨테이너 ID가 너무 길어 외우기 어려우므로 ID의 앞부분만 입력해도 실행이 가능하다. 단, 입력한 부분까지는 다른 컨테이너와 명확히 구분되어야 한다.

```
seo ~ docker start -a 305
```

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:

1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub. (arm64v8)
3. The Docker daemon created a new container from that image which runs the executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
\$ `docker run -it ubuntu bash`

Share images, automate workflows, and more with a free Docker ID:
<https://hub.docker.com/>

For more examples and ideas, visit:
<https://docs.docker.com/get-started/>

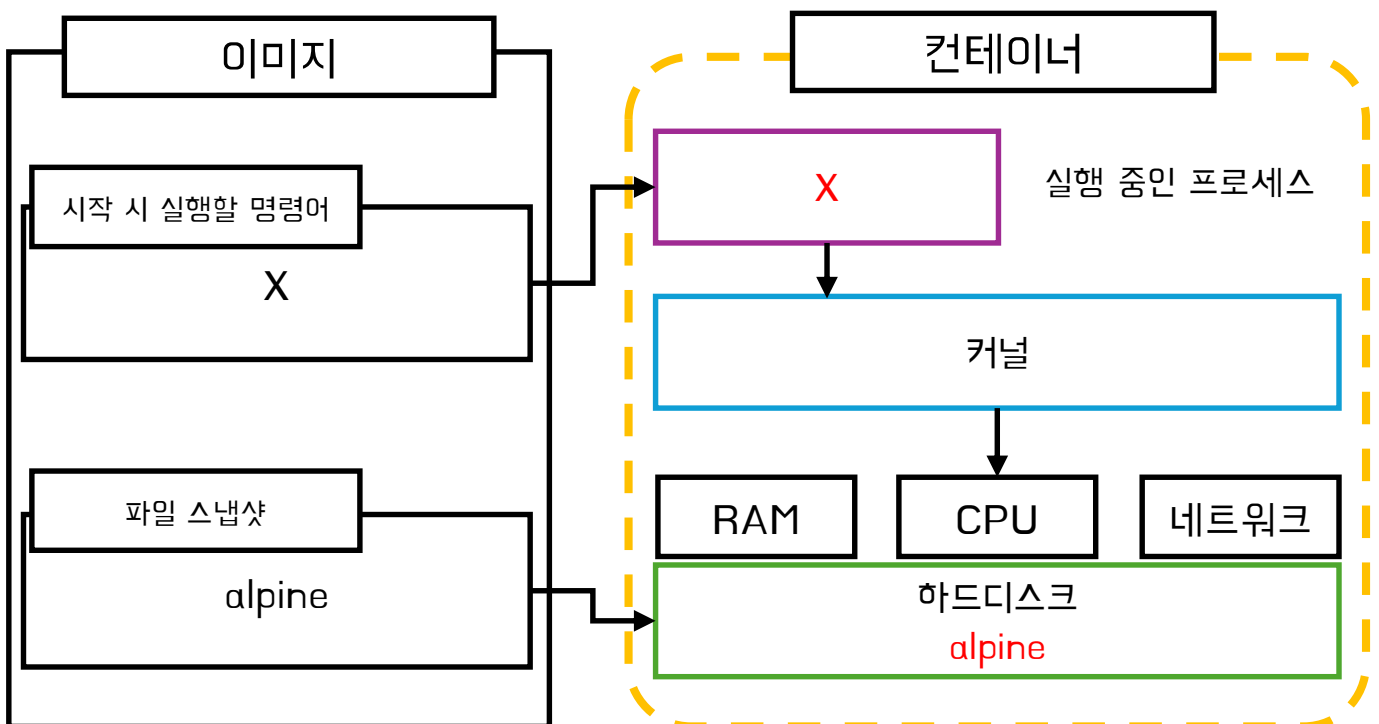
docker run

docker run 명령어는 도커 이미지로부터 새로운 컨테이너를 생성하고, 즉시 실행까지 하는 명령어이다. 이 명령어는 컨테이너 생성(create)과 시작(start)을 동시에 수행하며, 가장 많이 사용되는 도커 명령어 중 하나이다. 지정한 이미지를 기반으로 새로운 컨테이너를 생성하고 즉시 실행하여 내부 애플리케이션을 동작시킨다.

docker run 명령어

docker run (옵션) [이미지 이름] (수행할 명령어)

도커 이미지에는 컨테이너가 처음 시작되었을 때 실행할 명령어가 들어가 있어서 수행할 명령어를 적지 않게 된다면 아래 그림과 같이 자동적으로 해당 이미지의 시작 명령어가 수행이 된다.



이번에는 alpine 이미지를 토대로 실험해보자.

```
seo ~ docker run alpine
Unable to find image 'alpine:latest' locally
latest: Pulling from library/alpine
d69d4d41cfe2: Pull complete
Digest: sha256:8a1f59ffb675680d47db6337b49d22281a139e9d709335b492be023728e11715
Status: Downloaded newer image for alpine:latest
```

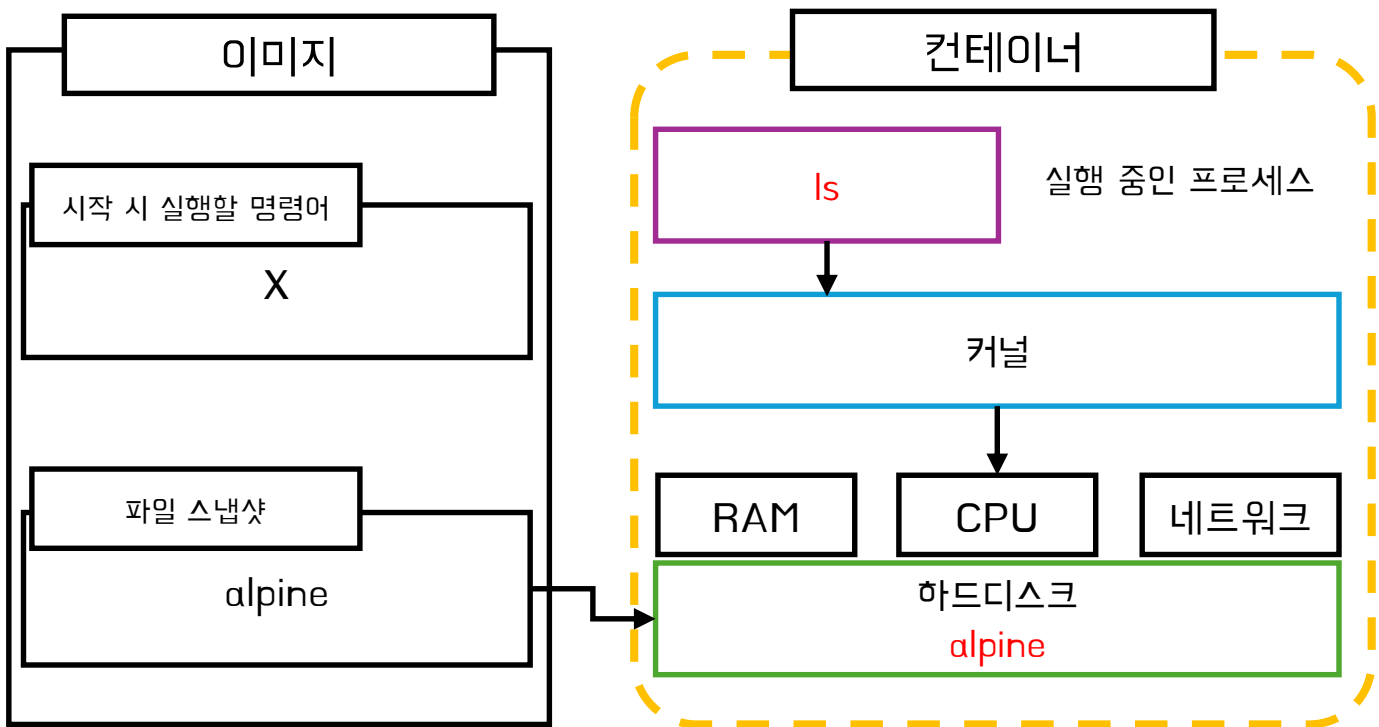
실행은 되었지만 출력 결과물이 나타나지 않는 것을 확인하였다. 이번에는 아래와 같이 실행할 명령어를 붙여 다시 작성해보자.

alpine 이미지 ls

```
docker run alpine ls
```

```
seo ~ docker run alpine ls
bin
dev
etc
home
lib
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
```

위와 같이 수행할 명령어가 들어오면, 해당 시작 명령어는 무시되고 우리가 작성한 명령어가 실행된다.



그림으로 표현하면 위와 같이 표현할 수 있다.

Containers

Images

Volumes

Builds

Docker Hub

Docker Scout

Extensions

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)

Container CPU usage

No containers are running.

Container memory usage

No containers are running.

[Show charts](#)

Q Search

Only show running containers

	Name	Container ID	Image	Port(s)	CPU (%)	Last star	Actions
☐	☐ determined_rhoi	4a9b125ce7e8	alpine		N/A	4 second	▶ ⋮ 🗑
☐	☐ interesting_nortl	27ce317170c3	alpine		N/A	0 second	▶ ⋮ 🗑

Showing 2 items

도커 데스크탑을 다시 가보면, ls를 입력했을 때와 입력하지 않았을 때의 각각 생성된 컨테이너를 확인할 수 있다. 아까와 마찬가지로 `-a` 옵션으로 컨테이너의 이름을 지정할 수 있다.

docker ps

docker ps 명령어는 현재 실행 중인 도커 컨테이너들의 목록을 보여준다. 이 명령어를 통해 컨테이너 ID, 이미지 이름, 실행 상태, 포트 매핑, 생성 시간 등 다양한 정보를 확인할 수 있다. 기본적으로는 실행 중인 컨테이너만 출력하지만, 옵션을 사용하면 중지된 컨테이너까지 모두 표시할 수 있어 컨테이너 상태를 빠르게 파악하고 관리하는 데 유용하다.

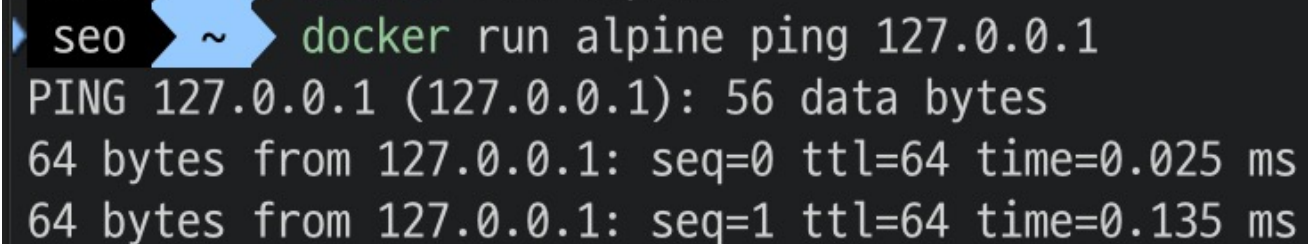
docker ps 명령어

docker ps (옵션)

해당 명령어 실행하기 위해 터미널 창을 띄우고 아래와 같이 입력한다.

첫 번째 터미널 입력 명령어

docker run alpine ping 127.0.0.1



```
seo ~ docker run alpine ping 127.0.0.1
PING 127.0.0.1 (127.0.0.1): 56 data bytes
64 bytes from 127.0.0.1: seq=0 ttl=64 time=0.025 ms
64 bytes from 127.0.0.1: seq=1 ttl=64 time=0.135 ms
```

이후, 새로운 터미널 창을 하나 더 열고 아래 명령어를 입력한다.

두 번째 터미널 입력 명령어

docker ps

```
seo ~ docker ps
```

CONTAINER ID	IMAGE	COMMAND	CREATED	STATUS	PORTS	NAMES
d65577a9d27f	alpine	"ping 127.0.0.1"	5 minutes ago	Up 5 minutes		elastic_leavitt

위 사진과 같이 현재 실행중인 컨테이너 목록을 확인할 수 있다. 목록에는 Container ID, Image, Command, Created, Status, Ports, Names 가 포함되어 있으며, 각 항목에 대한 자세한 설명은 아래와 같다.

CONTAINER ID

컨테이너의 고유한 아이디 해시 값이다. 실제 값은 훨씬 길지만, 목록에는 일부분만 표시된다.

IMAGE

컨테이너 생성 시 사용된 도커 이미지 이름이다.

COMMAND

컨테이너 시작 시 실행되는 명령어를 나타낸다. 대부분의 경우 이미 이미지에 내장된 명령어가 실행되므로 별도의 설정이 필요하지 않다. 단, 실행할 때 추가 명령어를 입력한 경우 이 부분에 해당 내용이 표시된다.

CREATED

컨테이너가 생성된 시점을 나타낸다. 일반적으로 ‘몇 분 전’, ‘몇 시간 전’과 같이 상대적인 시간으로 표시된다.

STATUS

컨테이너의 상태이다. 실행 중일 경우 Up, 종료는 Exited, 일시정지는 Paused로 표시된다.

PORTS

호스트와 컨테이너 간에 매핑된 포트 정보를 보여준다. 예를 들어, 0.0.0.0:8000 -> 80/tcp는 호스트의 8080번 포트가 컨테이너의 80번 포트와 연결되어 있음을 의미한다. 특별한 설정을 하지 않은 경우에는 출력되지 않는다.

NAMES

컨테이너에 부여된 이름이다. 기본적으로 도커가 랜덤한 이름을 자동으로 생성하지만, 컨테이너 생성 시 --name 옵션으로 직접 지정할 수도 있다. id와 마찬가지로 중복이 되지 않고, docker rename 명령어로 이름을 변경할 수 있다.

만약 docker ps로 출력 결과 중에 NAMES와 IMAGES 등 원하는 항목만 보고 싶다면 --format 옵션을 사용할 수 있다.

docker ps 원하는 항목만 보기

```
docker ps --format 'table {{.Names}}/t{{.Image}}'
```

```
seo ~ docker ps --format 'table {{.Names}} \t {{.Image}}'
NAMES      IMAGE
elastic_leavitt  alpine
```


추가적으로, 실행 중이지 않은 모든 컨테이너를 확인하고 싶다면 `-a` 옵션을 사용하여 전체 컨테이너 목록을 볼 수 있다. `-a` 옵션은 'all'의 약자로, 모든 컨테이너를 보여준다는 의미이다.

docker ps -a 로 전체 컨테이너 보기

```
docker ps -a
```

기본적으로 `docker ps` 명령어는 실행 중인 컨테이너만 표시하지만, 아래 명령어는 종료된 컨테이너까지 모두 출력한다.

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)

Container CPU usage

0.02% / 800% (8 CPUs available)

Container memory usage

556KB / 7.47GB

Show charts

Q

Search

Only show running containers

☐		Name	Container ID	Image	Port(s)	CPU (%)	Last star	Actions
☐	☐	epic_cerf	79d4e7014bc0	alpine		0%	1 minute	
☐	☐	gifted_ellis	183dee6d5d00	alpine		0%	1 minute	
☐	☒	elegant_pasteur	4d54dfe6372d	alpine		0.02%	53 second	

위 사진에서는 현재 3개의 컨테이너 중 1개만 실행 중이며, 나머지 2개의 컨테이너는 종료 상태이다. 이때 `-a` 옵션을 사용하면 아래와 같이 모든 컨테이너가 출력한다.

```
seo ~ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS              PORTS          NAMES
4d54dfe6372d   alpine    "ping 127.0.0.1"        10 seconds ago Up 10 seconds          elegant_pasteur
183dee6d5d00   alpine    "/bin/sh"               19 seconds ago Exited (0) 19 seconds ago gifted_ellis
79d4e7014bc0   alpine    "/bin/sh"               22 seconds ago Exited (0) 22 seconds ago epic_cerf
```

docker stop

docker stop 명령어는 도커 컨테이너를 중지시키는 명령어이다. 마치 윈도우 시작메뉴에서 시스템 종료를 선택하는 것과 비슷하다.

이 명령어는 컨테이너 내에서 실행 중인 작업들이 모두 완료되고, 출력할 내용이 전부 출력될 때까지 기다린 후 컨테이너를 종료한다. 이를 자비로운 종료(Graceful shutdown)라고 하며, 이 과정에서 종료 준비 시간인 Grace Period를 제공한다.

docker stop 명령어

docker stop [컨테이너 아이디 / 컨테이너 이름]

```
seo ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
426f3b8ad23a   alpine    "ping 127.0.0.1"        6 seconds ago Up 6 seconds          romantic_easley
4d54dfe6372d   alpine    "ping 127.0.0.1"        2 days ago    Up 13 seconds          elegant_pasteur
seo ~ docker stop 4d54
4d54
seo ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
426f3b8ad23a   alpine    "ping 127.0.0.1"        37 seconds ago Up 37 seconds          romantic_easley
```

처음에 docker ps 명령어로 2개의 컨테이너가 실행 중임을 확인했다. 그 중 컨테이너 ID가 4d54로 시작하는 컨테이너를 docker stop 명령어로 하면, 터미널에 해당 컨테이너 ID가 출력된다.

이후 다시 docker ps 명령어로 실행 중인 컨테이너 목록을 확인하면, 방금 종료한 4d54 컨테이너가 목록에 나타나지 않는다. 이를 통하여 해당 컨테이너가 정상적으로 종료 되었음을 알 수 있다.

docker kill

docker kill 명령어는 도커 컨테이너를 즉시 강제 종료시키는 명령어이다. docker stop이 프로세스에 정상 종료 요청(SIGTERM)을 보내고 종료 준비 시간(Grace Period)을 주어 안전하게 종료하는 것과 달리, docker kill은 종료 준비 시간 없이 즉시 강제 종료(SIGKILL)를 수행하기 때문에, 프로세스가 현재 수행 중인 작업을 마치지 못할 수 있다.

따라서 docker kill은 컨테이너가 정상적으로 종료되지 않거나 멈춘 경우 빠른 종료가 필요할 때 사용한다.

docker kill 명령어

docker kill [컨테이너 아이디 / 컨테이너 이름]

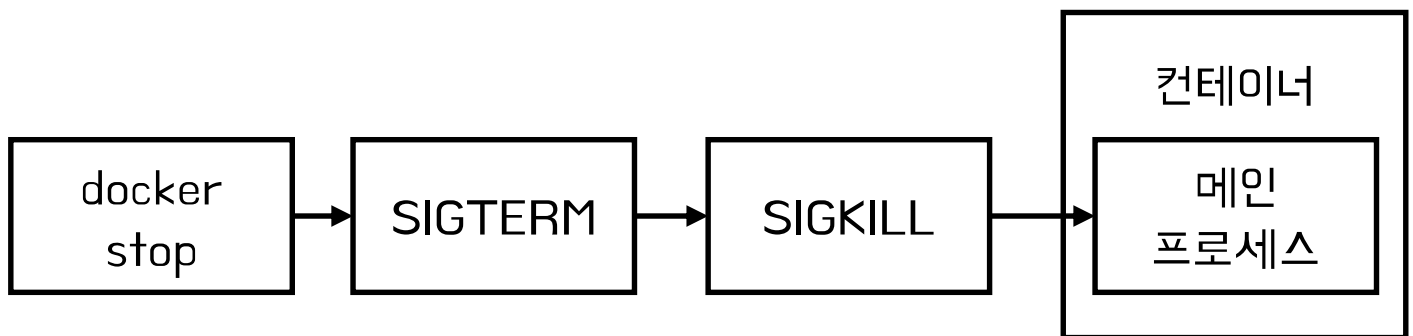
```
seo ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
426f3b8ad23a   alpine    "ping 127.0.0.1"        37 seconds ago Up 37 seconds        romantic_easley
seo ~ docker kill 426f
426f
seo ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS        NAMES
seo ~
```

이번에는 docker ps 명령어로 컨테이너 목록을 확인한 결과, 컨테이너 ID가 426f로 시작하는 컨테이너가 현재 실행 중임을 확인하였다. 이 컨테이너를 docker kill 명령어로 강제 종료한 후, 다시 실행 중인 컨테이너 목록을 불러오니 아무 컨테이너도 표시되지 않았다. 이를 통해 426f 컨테이너가 정상적으로 강제 종료되었음을 확인할 수 있다.

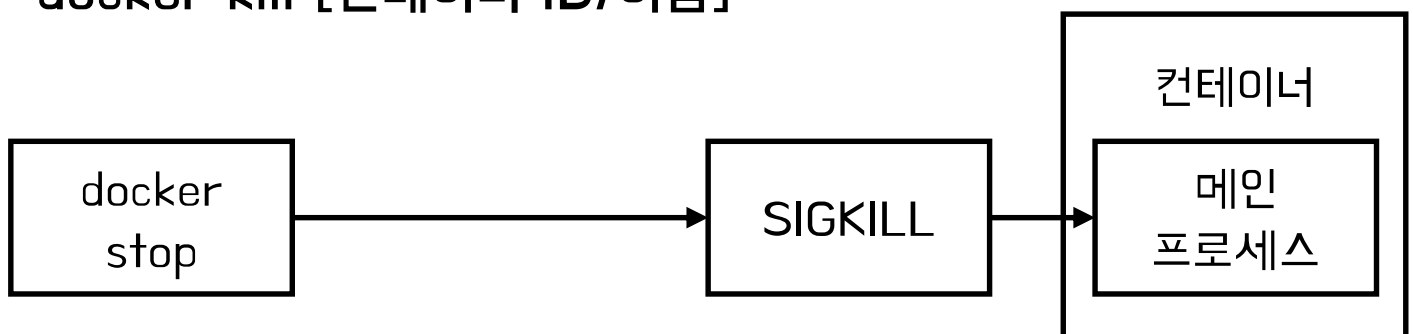
docker stop 과 docker kill의 차이

docker stop과 docker kill은 둘 다 컨테이너를 종료하는 명령어이다. 이 두 명령어의 큰 차이점은 컨테이너를 종료하는 방식에 있다. docker stop은 컨테이너에 정상 종료 요청(SIGTERM)을 보내고 Grace Period를 주어 안전하게 종료하도록 기다리는 반면, docker kill은 종료 준비 시간 없이 강제로 종료(SIGKILL)을 한다.

docker stop [컨테이너 ID/이름]



docker kill [컨테이너 ID/이름]



따라서 docker stop은 프로세스가 작업을 마칠 시간을 제공하지만, docker kill은 즉시 프로세스를 중단 시켜 데이터 손상 위험이 있을 수 있다.

docker rm

docker rm은 도커 컨테이너를 삭제하는 명령어이다. 중지된 컨테이너를 시스템에서 완전히 제거하여 디스크 공간을 해제하고, 더 이상 해당 컨테이너를 사용할 수 없게 만든다. 따라서 지컨테이너를 삭제하기 전에 반드시 컨테이너가 종료되어 있어야 한다.

docker rm 명령어

docker rm [컨테이너 아이디 / 컨테이너 이름]

```
seo ~ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS              PORTS          NAMES
426f3b8ad23a   alpine    "ping 127.0.0.1"        6 hours ago    Exited (137) 5 minutes ago
4d54dfe6372d   alpine    "ping 127.0.0.1"        2 days ago     Exited (137) 5 minutes ago
183dee6d5d00   alpine    "/bin/sh"               2 days ago     Exited (0) 2 days ago
79d4e7014bc0   alpine    "/bin/sh"               2 days ago     Exited (0) 2 days ago
epic_cerf

seo ~ docker rm 79d
79d
seo ~ docker ps -a
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS              PORTS          NAMES
426f3b8ad23a   alpine    "ping 127.0.0.1"        6 hours ago    Exited (137) 6 hours ago
4d54dfe6372d   alpine    "ping 127.0.0.1"        2 days ago     Exited (137) 6 hours ago
183dee6d5d00   alpine    "/bin/sh"               2 days ago     Exited (0) 2 days ago
gifted_ellis

seo ~
```

docker ps 명령어는 기본적으로 실행 중인 컨테이너만 표시하기 때문에, 종료된 컨테이너 목록을 보려면 -a 옵션을 추가하여 docker ps -a로 조회해야 한다. 현재 종료된 컨테이너는 4개임을 확인할 수 있다. 이 중 가장 마지막으로 출력된 컨테이너 ID가 79d4로 시작하는 컨테이너를 docker rm 명령어로 삭제하였다. 이후 다시 docker ps -a 명령어로 전체 컨테이너 목록을 확인하니, 79d4로 시작하는 컨테이너가 목록에서 사라진 것을 확인할 수 있다.

만약 삭제해야 할 컨테이너가 많을 경우, 다음과 같은 명령어로 모든 컨테이너를 한 번에 삭제할 수도 있다.

모든 컨테이너 지우기

```
docker rm `docker ps -a -q`
```

```

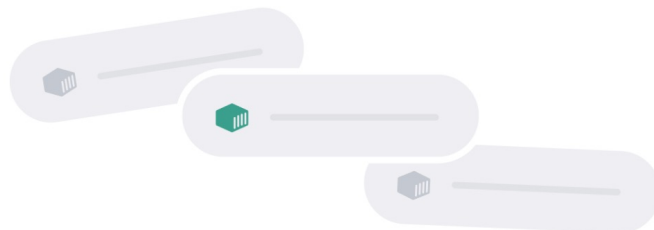
x seo ~ docker rm `docker ps -a -q`
426f3b8ad23a
4d54dfe6372d
183dee6d5d00
seo ~ docker ps -a
CONTAINER ID    IMAGE    COMMAND    CREATED    STATUS    PORTS    NAMES
seo ~

```

앞서 `docker ps -a` 명령어로 출력되었던 모든 컨테이너가 삭제되어, 컨테이너 목록을 조회해도 아무것도 나타나지 않는 것을 확인할 수 있다.

Containers [Give feedback](#)

View all your running containers and applications. [Learn more](#)



Your running containers show up here

A container is an isolated environment for your code



What is a container?

5 mins

```

1 FROM node
2 RUN mkdir -p
3 WORKDIR /app
4 COPY packa

```

How do I run a container?

6 mins

[View more in the Learning center](#)

또한, Docker Desktop에서도 모든 컨테이너가 삭제된 것을 확인할 수 있다.

docker images

docker images 명령어는 로컬 시스템에 저장된 도커 이미지 목록을 보여주는 명령어이다. 이 명령어를 통해 이미지의 레포지토리 이름, 태그, 이미지 ID, 생성 시점, 크기 등 다양한 정보를 확인할 수 있다

docker images 명령어

docker images (옵션)

```
seo ~ docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
alpine	latest	2abc5e834071	6 weeks ago	8.52MB
hello-world	latest	f1f77a0f96b7	5 months ago	5.2kB

docker rmi

docker rmi 명령어는 도커 이미지를 삭제하는 명령어이다. 더 이상 사용하지 않는 도커 이미지를 시스템에서 제거하여 디스크 공간을 확보할 때 사용한다. 사용하지 않는 이미지나 오래된 이미지를 삭제할 수 있으나, 이미지 삭제 시, 해당 이미지를 사용하는 컨테이너가 존재하면 삭제가 되지 않는다.

docker rmi 명령어

docker rmi (옵션) [이미지 ID / 이미지 이름]

-f (force) 옵션을 사용하면 강제로 이미지를 삭제할 수 있긴 하지만, 이 경우 이미지와 연결된 컨테이너도 함께 삭제할 수 있으므로 주의해야 한다.

docker ps와 docker images를 사용해 현재 alpine 이미지를 사용하는 컨테이너가 실행 중임을 확인할 수 있다.

```
seo ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS       NAMES
fcce5488e2ae   alpine    "ping 127.0.0.1"        11 seconds ago Up 10 seconds          busy_hawking
seo ~ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
alpine        latest    2abc5e834071   6 weeks ago    8.52MB
hello-world    latest    f1f77a0f96b7   5 months ago    5.2kB
```

docker rmi hello-world 명령어를 입력하면

```
seo ~ docker rmi hello-world
Untagged: hello-world:latest
Untagged: hello-world@sha256:940c619fbd418f9b2b1b63e25d8861f9cc1b46e3fc8b018ccfe8b78f19b8cc4f
Deleted: sha256:f1f77a0f96b7251d7ef5472705624e2d76db64855b5b121e1cbefe9dc52d0f86
Deleted: sha256:98a92b28c9b8d3cd4353801cba6ad181e86aaa616221e14739210acbda35b7fd
seo ~ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
alpine        latest    2abc5e834071   6 weeks ago    8.52MB
seo ~
```

위 사진과 같이 정상적으로 이미지가 삭제된다. 그러나, alpine을 지우려고 하면,

```
seo ~ docker rmi alpine
Error response from daemon: conflict: unable to remove repository reference "alpine" (must force) - container fcce5488e2ae is using its referenced image 2abc5e834071
seo ~
```

해당 이미지를 사용하는 컨테이너가 있어서 삭제할 수 없다는 메시지가 출력된다. 이 경우 -f 옵션을 사용해 강제로 삭제할 수 있다.

```
seo ~ docker rmi -f alpine
Untagged: alpine:latest
Untagged: alpine@sha256:8a1f59ffb675680d47db6337b49d22281a139e9d709335b492be023728e11715
seo ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS       NAMES
fcce5488e2ae   2abc5e834071   "ping 127.0.0.1"        21 minutes ago Up 21 minutes          busy_hawking
seo ~ docker images
REPOSITORY    TAG       IMAGE ID       CREATED        SIZE
<none>        <none>    2abc5e834071   6 weeks ago    8.52MB
seo ~
```

이미지가 삭제되어 비어 있는 것을 확인할 수 있다. 다행히 컨테이너는 삭제되지 않았지만, -f 옵션은 주의해서 사용해야 한다.

docker system prune

docker system prune 명령어는 도커 시스템에서 사용하지 않는 데이터들을 한꺼번에 정리(cleanup) 하는 명령어이다. 이 명령어를 실행하면 중지된 컨테이너, 사용하지 않는 네트워크, dangling 이미지(태그가 없는 이미지), 빌드 캐시 등이 삭제 되어 디스크 공간을 확보할 수 있다.

docker system prune 명령어

docker system prune (옵션)

```

x seo ~ docker system prune
WARNING! This will remove:
- all stopped containers
- all networks not used by at least one container
- all dangling images
- unused build cache

Are you sure you want to continue? [y/N] y
Deleted Containers:
fcce5488e2ae1a43c40125a70efb61a03eb3a74ddeb148ff434f9f42fa847c7f

Deleted Images:
deleted: sha256:2abc5e83407155714f171c293f197e1310176959e106f8ad63ffa2e7e9635d4a
deleted: sha256:1231a673589ac9e2f8a98ed916d1fa8301b36a44dd14fc40eba5a05905da69f1

Total reclaimed space: 8.515MB
seo ~

```

해당 명령어를 실행하면 위 사진과 같이 시스템에 불필요하게 남아있는 리소스를 한 번에 정리한다. `-a` 옵션을 추가하면 모든 사용하지 않는 이미지까지 삭제하며, `-f` 옵션을 사용하면 확인 메시지 없이 강제로 삭제를 진행한다.

docker exec

docker exec 명령어는 이미 실행 중인 도커 컨테이너 내부에서 새로운 명령어나 프로세스를 실행할 때 사용하는 명령어이다. 이 명령어를 사용하면 컨테이너가 실행 중인 상태에서 추가적인 작업을 수행하거나, 쉘에 접속하여 직접 명령을 입력할 수 있다. 컨테이너가 실행 중일 경우에만 사용할 수 있다, 컨테이너 내부 환경에 따라 사용할 수 있는 쉘이나 명령어가 다를 수 있다.

docker exec 명령어

docker exec (옵션) [컨테이너 ID / 컨테이너 이름] 명령어

```
seo ~ docker ps
CONTAINER ID   IMAGE     COMMAND                  CREATED        STATUS        PORTS          NAMES
9260eb32def4   alpine    "ping 127.0.0.1"        34 seconds ago Up 34 seconds          interesting_solomon
seo ~ docker exec 9260 ls
bin
dev
etc
home
lib
media
mnt
opt
proc
root
run
sbin
srv
sys
tmp
usr
var
seo ~
```

docker ps로 현재 실행 중인 9260 컨테이너를 확인한 후, docker exec 9260 ls 명령어를 사용해 컨테이너의 현재 디렉터리 목록을 출력하였다. 여러 명령어를 한 번에 입력하려면 ;을 구분자로 명령어를 연속으로 입력하면 사용할 수 있다.

하지만 새로운 명령어를 실행할 때마다 `docker exec` 컨테이너 ID를 반복해서 입력 하는 것은 번거로울 수 있다. 이런 경우, `-it` 옵션을 사용해 컨테이너 내부에 직접 접속하는 방법이 있다.

`-i` 옵션은 표준 입력(stdin)을 활성화하여 터미널로부터 입력을 받을 수 있게 해주고, `-t` 옵션은 가상 터미널(tty)을 할당하여 실제 터미널과 같은 환경을 제공한다. 두 옵션을 함께 사용하면 컨테이너 내부에서 대화형 터미널을 사용할 수 있어, 마치 직접 접속한 것처럼 명령어를 연속해서 실행할 수 있다.

docker exec -it로 터미널 셸 사용하기

`docker exec -it 컨테이너 이름 /bin/bash`

```

x seo ~ docker exec -it 9260 /bin/sh
/ # ls
bin dev etc home lib media mnt opt proc root run sbin srv sys tmp usr var
/ # ls -al
total 64
drwxr-xr-x 1 root root 4096 Jul 16 05:17 .
drwxr-xr-x 1 root root 4096 Jul 16 05:17 ..
-rwxr-xr-x 1 root root 0 Jul 16 05:17 .dockerenv
drwxr-xr-x 2 root root 4096 Jul 15 10:42 bin
drwxr-xr-x 5 root root 340 Jul 16 05:17 dev
drwxr-xr-x 1 root root 4096 Jul 16 05:17 etc
drwxr-xr-x 2 root root 4096 Jul 15 10:42 home
drwxr-xr-x 6 root root 4096 Jul 15 10:42 lib
drwxr-xr-x 5 root root 4096 Jul 15 10:42 media
drwxr-xr-x 2 root root 4096 Jul 15 10:42 mnt
drwxr-xr-x 2 root root 4096 Jul 15 10:42 opt
dr-xr-xr-x 215 root root 0 Jul 16 05:17 proc
drwx----- 1 root root 4096 Jul 16 05:27 root
drwxr-xr-x 3 root root 4096 Jul 15 10:42 run
drwxr-xr-x 2 root root 4096 Jul 15 10:42 sbin
drwxr-xr-x 2 root root 4096 Jul 15 10:42 srv
dr-xr-xr-x 11 root root 0 Jul 16 05:17 sys
drwxrwxrwt 2 root root 4096 Jul 15 10:42 tmp
drwxr-xr-x 7 root root 4096 Jul 15 10:42 usr
drwxr-xr-x 11 root root 4096 Jul 15 10:42 var
/ #

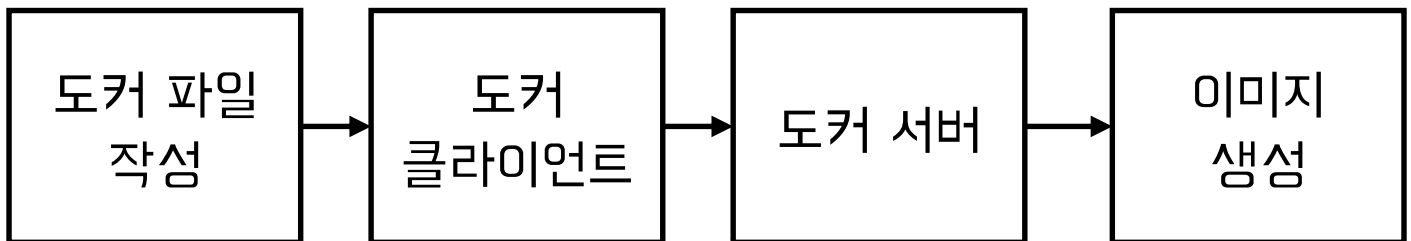
```

컨테이너 내부에서 작동하는 서비스에 접근하거나 문제를 확인하기 위해서는 위 명령어를 사용해 직접 컨테이너 내부로 들어가 확인해야 한다.

4-6. Docker 이미지 파일 만들기

도커 이미지 생성하기

도커 이미지는 컨테이너를 만들기 위한 설명서와 같다. 이 설명서에는 애플리케이션을 실행하는 데 필요한 모든 환경과 설정이 담겨 있다. 도커 이미지를 생성하는 과정은 이러한 설명서를 작성하고, 도커가 이를 바탕으로 완성된 환경을 만드는 단계로 이루어진다.



도커 이미지가 생성되는 과정은 위 흐름도와 같다.

첫 번째 단계는 도커 이미지의 설계도라 할 수 있는 Dockerfile을 작성하는 것이다. 작성된 Dockerfile을 기반으로, 두 번째 단계에서 도커 클라이언트(Docker Client)가 `docker build` 명령어를 실행한다. 이 명령은 도커 데몬(Docker Daemon)에게 빌드 작업을 요청하는 과정으로, 클라이언트가 사용자의 명령을 받아 데몬에 전달하고 빌드 진행 상황을 사용자에게 보여준다. 마지막으로 도커 데몬은 Dockerfile에 적힌 각 명령어를 차례로 처리하며, 그 결과를 레이어(layer) 형태로 쌓아 최종 도커 이미지가 완성된다. 이 이미지는 로컬 저장소에 보관되며, 필요에 따라 컨테이너를 생성하거나 원격 저장소로 업로드 할 수 있다.

이처럼 도커 이미지는 설계서 역할을 하는 Dockerfile 작성부터 시작해, 클라이언트와 데몬의 협력으로 효율적이고 재현 가능한 실행 환경을 만들어내는 과정을 거친다.

Dockerfile이란?

Dockerfile은 도커 이미지를 만들고 빌드하기 위한 텍스트 파일이다. 컨테이너 이미지를 구성하는데 필요한 모든 단계와 명령어를 정의하며, 이를 통해 반복적이고 일관된 이미지 빌드 프로세스를 자동화 할 수 있다.

Dockerfile 특징

1. 텍스트 기반

일반 텍스트 파일로 작성되어, 코드 버전 관리 시스템을 통해 추적하고 관리할 수 있다.

2. 실행 명령어

Docker 이미지를 구성하는 데 필요한 명령어를 포함한다. 이 명령어들은 순차적으로 실행되며, 각 명령어가 새로운 레이어를 Docker 이미지에 추가한다.

3. 고정된 환경

Dockerfile을 통해 생성된 이미지는 이를 실행하는 환경에 상관없이 항상 동일한 결과를 생성한다.

4. 재사용 가능

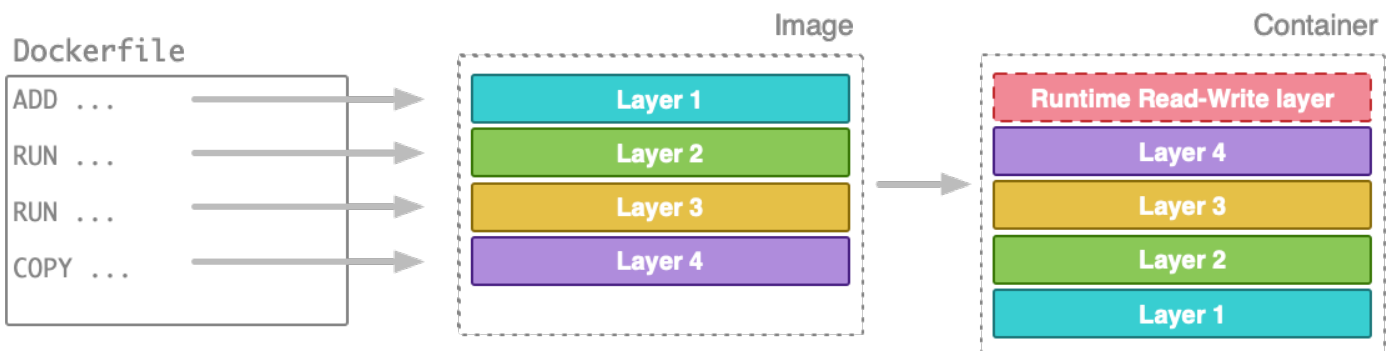
Dockerfile은 재사용이 가능하며, 다른 Dockerfile에 기반을 제공한다.

5. 빌드 자동화

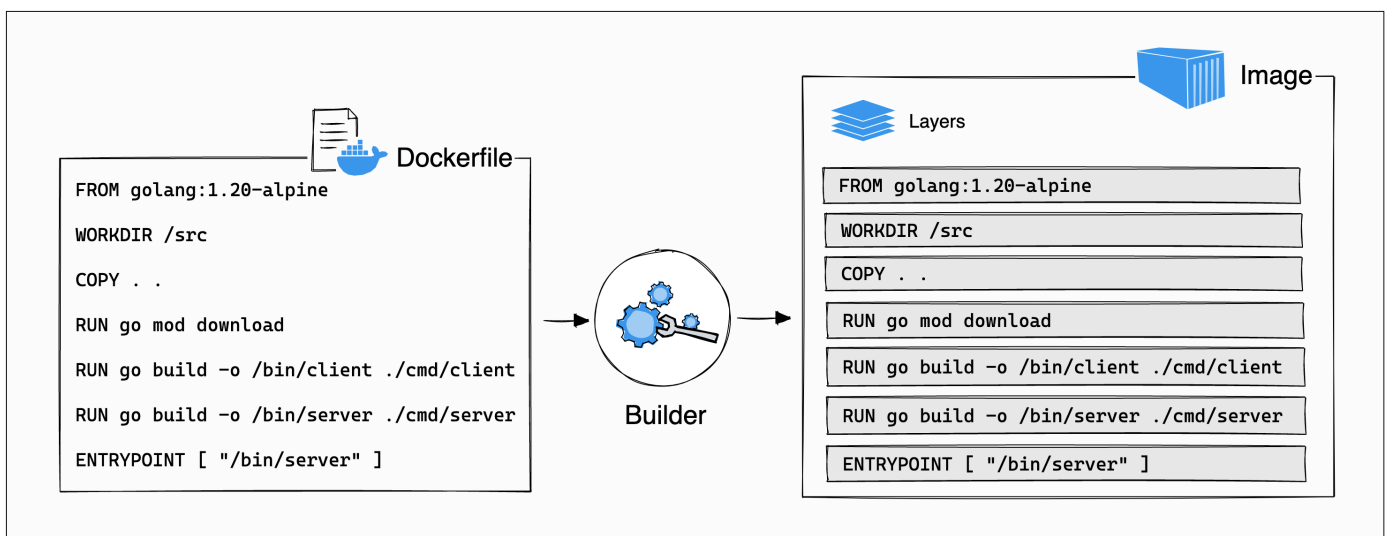
Docker build 명령을 사용하여 Dockerfile을 바탕으로 Docker 이미지를 생성하여 이미지 빌드 과정을 자동화 할 수 있다.

Dockerfile Layer

Dockerfile을 빌드하여 이미지를 생성할 때 각 명령어를 한 줄씩 실행하며 이미지의 새로운 레이어를 생성한다. 생성된 레이어는 변경 사항이 있을 때만 다시 빌드되는데, 이를 통해 변경된 부분만 재빌드하여 빌드 속도를 향상시키고 불필요한 중복 작업을 방지한다. 이 과정을 레이어 캐싱(Layer Caching)이라고 부르며, 도커 빌드의 효율성과 속도를 크게 개선하는 핵심 기술이다.



또한, 각 레이어는 불변(immutable) 특성을 가지며, 여러 이미지 간에 동일한 레이어를 공유할 수 있어 저장 공간과 네트워크 전송량을 절약하는 효과도 있다. 이런 계층화된 구조 덕분에 도커 이미지는 빠르게 배포되고 관리될 수 있다.



Dockerfile Cached Layer

도커 빌드 과정에서 이전에 실행한 명령어의 결과로 생성된 레이어를 캐시(Layer Cache)라고 한다. 도커는 동일한 명령어와 환경에서 이미 빌드된 레이어가 있으면, 이를 재사용해서 빌드 시간을 단축한다.

각 명령어 실행 시 도커는 이전 빌드에서 생성된 레이어가 존재하는지 확인한다. 변경 사항이 없으면 기존 레이어를 그대로 사용하고, 변경이 발생하면 새 레이어를 생성한다. 이를 통해 변경된 부분만 새로 빌드하고, 나머지는 캐시된 레이어를 활용해 빌드 효율을 높인다.

캐시를 활용하면 변경되지 않은 부분을 다시 빌드하지 않아 빠르게 이미지를 생성할 수 있어 빌드 시간이 단축된다. 또한 중복 작업이 줄어들어 CPU와 디스크 사용량이 절감되어 자원을 효율적으로 사용할 수 있다. 이전 빌드 결과를 재사용함으로써 안정적인 이미지 빌드가 가능해 일관성 또한 유지된다.

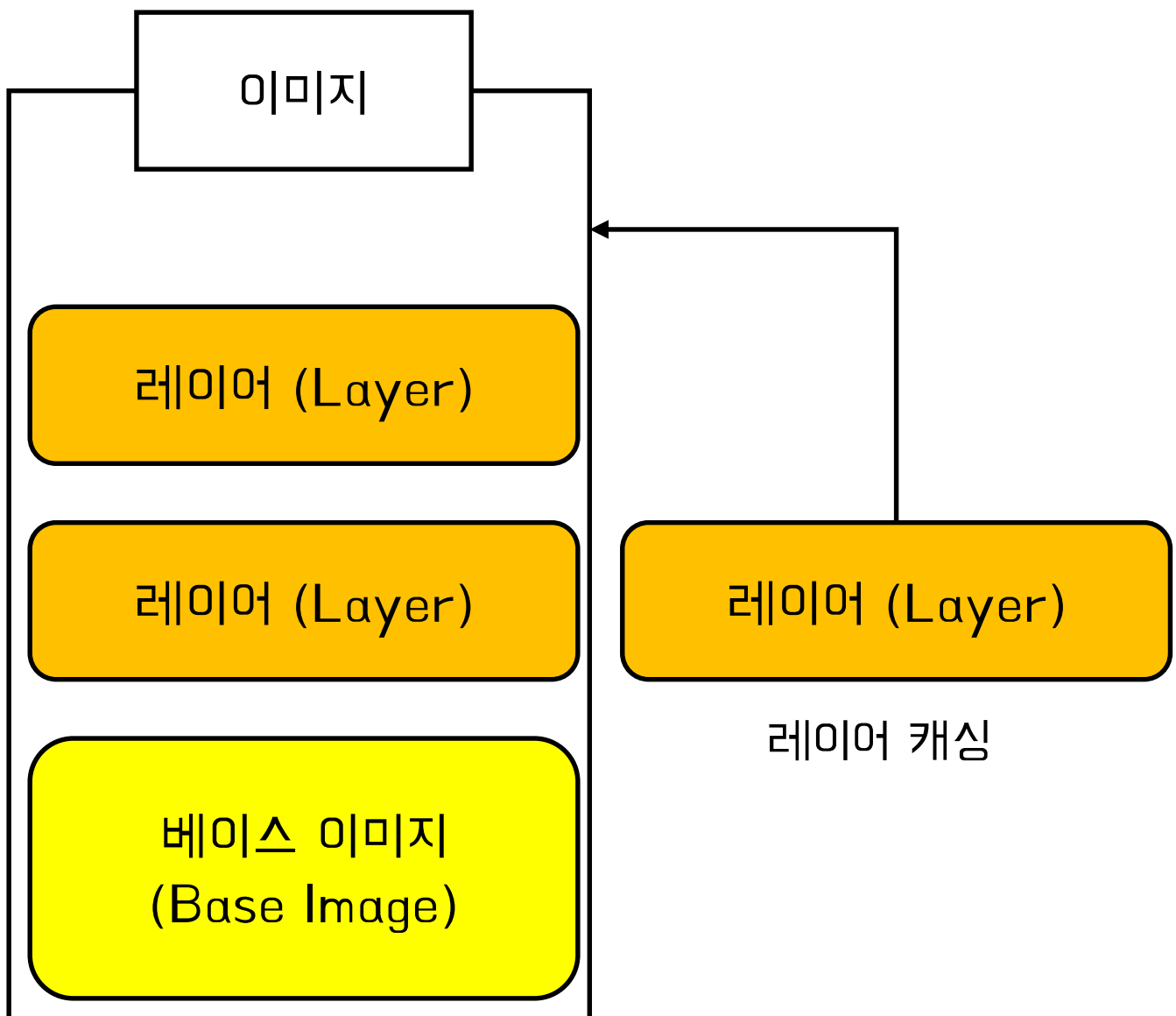
하지만, 캐시가 너무 오래되거나 변경 사항이 제대로 반영되지 않는 경우에는 수동으로 캐시를 무효화하거나 빌드 옵션을 조정하여 새롭게 빌드할 필요가 있다.

Layers	Cache?
FROM golang:1.20-alpine	✓
WORKDIR /src	✓
COPY . .	✗
RUN go mod download	✗
RUN go build -o /bin/client ./cmd/client	✗
RUN go build -o /bin/server ./cmd/server	✗
ENTRYPOINT ["/bin/server"]	✗

베이스 이미지(Base Image)

베이스 이미지는 Dockerfile에서 다른 모든 명령어를 실행하기 전에 기반이 되는 도커 이미지이다. 즉, 새로 만들 도커 이미지의 출발점이 되는 이미지로, 보통 운영체제(OS)나 미리 설정된 환경을 포함한다.

도커 이미지는 여러 개의 레이어들로 구성되어있다. 그 중에서 베이스 이미지는 이미지의 기반이 되는 부분이다. 레이어는 중간 단계의 이미지 정도라고 생각하면 된다.



Dockerfile 작성하기

간단하게 “Hello World”를 출력하는 도커 이미지를 만들어 보자. 기본적으로 도커 이미지의 구조는 다음과 같은 주요 명령어들로 이루어지며, 그 중 가장 근본적인 명령어는 FROM, RUN 그리고 CMD이다.

Dockerfile 구조

```
# 베이스 이미지 명시
FROM baseImage

# 추가적으로 필요한 파일들 명시
RUN command

# 컨테이너 시작시 실행될 명령어
CMD [ "executable" ]
```

기본 문법은 다음과 같다. FROM은 이미지 생성 시 기반이 되는 베이스 이미지를 지정한다. RUN은 이미지 빌드 과정에서 실행할 쉘 명령어를 입력하며, 이를 통해 필요한 패키지 설치나 설정 작업을 수행한다. CMD는 컨테이너가 시작될 때 실행할 명령어나 스크립트를 지정하며, Dockerfile 내에서 한 번만 사용할 수 있다.

이 세 가지 명령어를 조합해 원하는 환경과 동작을 정의하면, 도커 이미지를 통해 컨테이너를 실행할 수 있다. 위의 구조대로 “Hello World”를 출력하는 문구를 작성하면 다음과 같다.

hello world를 출력하는 Dockerfile 만들기

```

1  FROM alpine
2  # 베이스 이미지 명시
3
4  # 추가적으로 필요한 파일들 명시
5  # RUN command
6
7  # 컨테이너 시작시 실행될 명령어
8  CMD [ "echo", "Hello", "World"]

```

이 때 FROM 부분에서 우분투, CentOS 등 다양한 베이스 이미지를 사용할 수 있다. 하지만 이번에는 출력만 하는 간단한 기능을 구현할 예정이므로, 용량이 작고 경량화 된 alpine 이미지를 선택하였다.

docker build

다음, 이미지를 빌드 하기 위해 아래와 같이 입력한다.

도커파일 빌드하기

```
docker build (옵션) [Dockerfile 파일이 위치한 디렉터리]
```

현재 있는 디렉터리에 Dockerfile이 있다면, Dockerfile 파일이 위치한 디렉터리 부분에 현재 위치를 알려주는 . 이나 ./을 입력한다.

```
seo ~ docker build ./
[+] Building 2.8s (5/5) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 257B
=> [internal] load metadata for docker.io/library/alpine:latest
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [1/1] FROM docker.io/library/alpine:latest
=> exporting to image
=> => exporting layers
=> => writing image sha256:fc0b12fd36075051e095b797578e2010cd00306820f309e89940f35cae7ccdfe
```

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/r1z5nu5mfjs27mql5ypaxzcfi

What's next:
View a summary of image vulnerabilities and recommendations → [docker scout quickview](#)

도커 빌드가 완료되면, 위 화면의 파란색 글씨 맨 아래에 이미지ID가 표시된다. 이 이미지를 `docker run` 명령어로 실행하면, 방금 Dockerfile로 생성한 도커 이미지를 기반으로 컨테이너를 실행할 수 있다.

```
What's next:
View a summary of image vulnerabilities and recommendations → docker scout quickview
seo ~ docker run fc0b
Hello World
seo ~
```

컨테이너 ID처럼 이미지 ID로 앞자리 일부만 입력해도 실행이 가능하다. 위 사진과 같이 정상적으로 실행된 것을 확인할 수 있다.

```
seo ~ docker build ./
[+] Building 0.0s (5/5) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 257B
=> [internal] load metadata for docker.io/library/alpine:latest
=> [internal] load .dockerignore
=> => transferring context: 2B
=> CACHED [1/1] FROM docker.io/library/alpine:latest
=> exporting to image
=> => exporting layers
=> => writing image sha256:fc0b12fd36075051e095b797578e2010cd00306820f309e89940f35cae7ccdfe
```

View build details: docker-desktop://dashboard/build/desktop-linux/desktop-linux/eaiup8qdfwlv5y6x8vfruo8g

What's next:
View a summary of image vulnerabilities and recommendations → [docker scout quickview](#)

동일한 Dockerfile로 `docker build ./` 명령을 다시 실행하면, 파란색 글씨의 CACHED [1/1] 부분에서 확인할 수 있듯이, 도커는 캐시된 레이어(CACHED Layer)를 재사용하여 이전에 빌드한 레이어를 그대로 활용한다는 것을 알 수 있다.

제 5 장

AWS

- 1) AWS 개요 & 비용(온디맨드/예약/스팟)
- 2) IAM 기본
- 3) 보충: IP와 Port
- 4) 백엔드 API 서버 배포하기
- 5) 보충: 서버가 잘 돌아가는지 확인하기

5-1. AWS 개요 & 비용(온디맨드/예약/스팟)

AWS 개요

아마존 웹 서비스(Amazon Web Services, AWS)는 전 세계에서 가장 널리 사용되는 클라우드 컴퓨팅 플랫폼이다.

AWS는 데이터센터, 네트워크, 서버, 스토리지, 데이터베이스, 머신러닝, 보안 서비스까지 모두 제공하여, 사용자는 물리적 하드웨어를 직접 구매하거나 관리하지 않고도 원하는 IT 자원을 즉시 확보할 수 있다.

특징:

- 온디맨드(On-Demand): 필요한 순간에 즉시 자원을 생성하고, 사용량만큼만 비용을 지불한다.
- 탄력성: 트래픽이 늘어나면 자원을 확장하고, 줄어들면 축소할 수 있다.
- 글로벌 인프라: 리전(Region)과 가용 영역(Availability Zone)을 기반으로 전 세계 어디서든 안정적인 서비스 제공이 가능하다.
- 다양한 서비스: EC2(서버), S3(스토리지), RDS(데이터베이스), Lambda(서버리스) 등 수백 개의 서비스를 제공한다.



비용 모델

AWS의 가장 큰 장점 중 하나는 유연한 과금 체계이다. 사용자는 자원 운영 방식에 따라 다른 비용 모델을 선택할 수 있다.

1) 온디맨드(On-Demand)

- 특징: 사용한 만큼만 시간 단위(또는 초 단위)로 비용을 지불한다.
- 장점: 초기 투자 없이 빠르게 시작할 수 있다.
- 적합한 경우: 예측 불가능한 워크로드, 단기 프로젝트, 테스트 환경.

2) 예약 인스턴스(Reserved Instance)

- 특징: 1년 또는 3년 약정을 맺고 특정 인스턴스를 예약한다.
- 장점: 온디맨드 대비 최대 70%까지 비용 절감 가능.
- 적합한 경우: 장기간 안정적으로 운영되는 서비스, 예측 가능한 트래픽.

3) 스팟 인스턴스(Spot Instance)

- 특징: AWS의 유휴 자원을 경매식으로 매우 저렴하게 사용한다.
- 장점: 온디맨드 대비 최대 90%까지 저렴하다.
- 단점: AWS가 자원을 회수할 수 있어 언제든지 인스턴스가 종료될 수 있다.
- 적합한 경우: 긴급하지 않고 중단에 강한 워크로드 (예: 빅데이터 분석, 배치 작업).

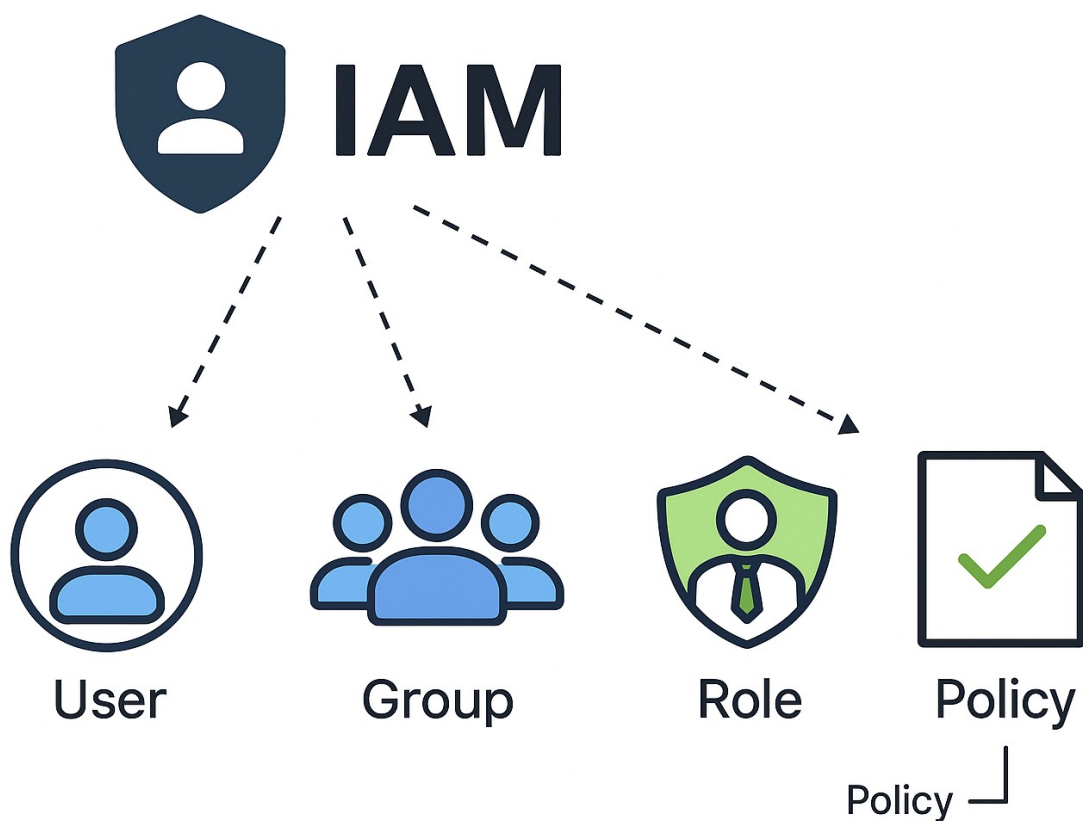
5-2. IAM 기본

IAM이란?

AWS Identity and Access Management(IAM)은 AWS 자원에 대한 접근 권한을 안전하게 제어하는 서비스이다.

IAM을 통해 사용자는 누가(Who) 어떤 자원(Resource)에 어떤 작업(Action)을 수행할 수 있는지를 정의할 수 있다.

즉, AWS 보안의 출발점이자 가장 중요한 기초 서비스이다.



IAM의 핵심 개념

1. User (사용자)

- AWS 콘솔이나 CLI로 로그인하는 개별 계정을 의미한다.
- 각 User는 인증 수단(비밀번호, Access Key)을 가진다.

2. Group (그룹)

- 여러 사용자를 묶어 동일한 권한을 부여할 수 있다.
- 예: Developers 그룹에 S3 읽기/쓰기 권한을 부여.

3. Role (역할)

- 특정 AWS 리소스(예: EC2 인스턴스)에 임시로 권한을 부여하는 방법이다.
- 예: EC2가 S3 버킷에 접근할 수 있도록 IAM Role을 부여.

4. Policy (정책)

- JSON 형식으로 작성된 권한 정의 문서이다.
- “어떤 자원(Resource)에 어떤 작업을 허용/거부할 것인가”를 기술한다.
- 예: S3의 특정 버킷에서 s3:GetObject만 허용하는 정책.

IAM 권한 제어 흐름

IAM은 다음 세 가지 요소를 기반으로 권한을 제어한다.

- 주체(Principal): 요청을 하는 사용자(User), 역할(Role), 또는 서비스.
- 정책(Policy): 허용 또는 거부 규칙.
- 자원(Resource): 접근 대상 (EC2, S3, RDS 등).

💡 권한은 기본적으로 거부이며, 정책에서 명시적으로 허용(Allow)해야 한다.
👉 여러 정책이 적용될 경우, 명시적 거부(Deny)가 최우선으로 적용된다.

5-3. 보충: IP와 Port

IP의 개념

💡 한 줄 요약: 네트워크 상에서의 특정 컴퓨터를 가리키는 주소
: 13.250.15.132

도메인의 IP 주소 알아내는 법

- nslookup example.com

```
> nslookup naver.com
Server:          192.168.1.1
Address:         192.168.1.1#53

Non-authoritative answer:
Name:   naver.com
Address: 223.130.200.236
Name:   naver.com
Address: 223.130.200.219
Name:   naver.com
Address: 223.130.192.247
Name:   naver.com
Address: 223.130.192.248
```

Port의 개념

💡 한 줄 요약: 한 컴퓨터 내에서 실행되고 있는 특정 프로그램의 주소이다.
 : 13.259.15.132:**3000**
 -> :**3000** 부분이 포트 번호

한 컴퓨터 내에서 여러 프로그램이 실행되고 있을 것이다. 내가 지금 사용하고 있는 노트북만 봐도 크롬, 카카오톡 등 여러가지 프로그램이 동시에 실행되고 있다.

→ 실제 서버를 운영하는 컴퓨터도 동일하다.

하나의 컴퓨터에서 여러가지의 프로그램이 실행된다.

! 그럼 외부에서 특정 컴퓨터 내부에 있는 Spring Boot라는 서버에 통신을 하고 싶다고 가정하자.

- 하지만 외부에서 IP 주소만 알아선 실행되고 있는 여러 프로그램 중 어떤 프로그램이랑 통신을 해야 할 지 알 수가 없다.
- 그렇기 때문에 특정 서버와 통신을 할 때는 IP 주소와 서버가 실행되고 있는 **포트 번호**까지 알고 있어야 한다.

브라우저 창에 포트 번호를 입력하지 않는 이유는?

도메인 주소를 통해서 알 수 있는 건 IP 주소 뿐이다. 그럼 포트 번호를 입력해주지도 않았는데 어떻게 정상적으로 통신을 할 수 있을까?

 naver.com

주소창에 도메인 주소를 입력해서 엔터를 누르면, 브라우저(크롬, 익스플로러 등)는 기본적으로 80번 포트로 통신을 보내게 설정되어 있다.

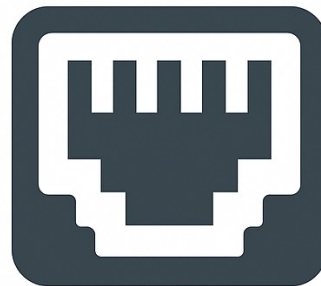
→ 포트 번호를 입력해주지 않아도 통신이 잘 됐던 것

만약 80번 포트가 아닌 3000번 포트로 통신하고 싶다면 아래처럼 입력하면 된다.

 naver.com:3000



**IP
Address**



**Port
Number
80**

잘 알려진 포트(well-known port)란?

포트 번호는 0 ~ 65,535번까지 사용할 수 있다. 그 중에서 0 ~ 1023번까지의 포트 번호는 주요 통신을 위한 규약에 따라 이미 정해져 있다. 이렇게 규약을 통해 역할이 정해져있는 포트 번호를 보고 잘 알려진 포트(well-known port)라고 부른다.

규약으로 정해져 있는 포트 번호 중 자주 사용되는 포트 번호에 대해서만 알아보자.

- 22번 (SSH, Secure Shell Protocol): 원격 접속을 위한 포트 번호
 - EC2 인스턴스에 연결할 때 22번 포트를 사용한다.
- 80번(HTTP): HTTP로 통신을 할 때 사용
- 443번 (HTTPS): HTTPS로 통신을 할 때 사용

여기서 알아야 할 점은, 위에서 정해놓은 규약을 꼭 지키지 않아도 된다.

→ 규약으로 정해져 있는 포트 번호 다르게 사용해도 된다.

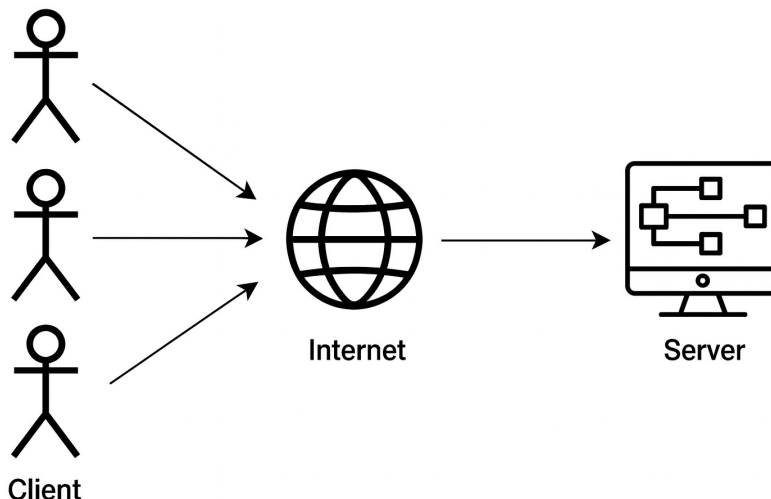
예를 들어 특정 서버와 HTTP 통신을 할 때 80번 포트를 쓰지 않고 3000포트나 8080번 포트를 사용해도 상관 없다.

5-4. 백엔드 API 서버 배포하기(EC2)

5.4.1. 배포(Deployment)란?

개발자들이 말하는 배포란 무엇일까

→ 다른 사용자들이 인터넷을 통해서 사용할 수 있게 만드는 것을 의미한다.



내 컴퓨터에서 개발을 할 때 localhost:8080 이런 주소로 테스트도 하고 개발도 한다. 하지만 이건 다른 컴퓨터에서 접속이 불가능하다.

배포를 하거나 프론트와 테스트를 할 때 IP(ex. 192.168.xx)나 도메인과 같이 고유의 주소를 부여받게 되고, 다른 컴퓨터에서 그 주소로 접속할 수 있게 된다.

이것이 배포!

따라서 어떤 서비스를 완성했다면, 그 다음 단계로 해야 하는 게 배포(Deployment)이다.

코딩으로 배울 때도 기본적인 백엔드 서버를 만들 수 있는 역량이 갖춰졌다면, 그 다음에 배워야 하는 게 배포(Deployment)이다.

5.4.2. EC2(Elastic Compute Cloud)

💡 한 줄 요약: 컴퓨터를 빌려서 원격으로 접속해 사용하는 서비스이다.

EC2를 쉽게 얘기하면 하나의 컴퓨터를 의미한다.

사용자는 직접 하드웨어를 구매하거나 24시간 전원을 켜둘 필요 없이, 필요할 때 즉시 서버를 생성하고 사용할 수 있다.

EC2(Elastic Compute Cloud)를 왜 배울까

서버를 배포하려면 반드시 컴퓨터가 필요하다.

물리적인 개인용 PC를 서버로 사용할 수도 있지만 다음과 같은 문제점이 있다.

- 서버를 24시간 켜둬야 하므로 전력 및 유지보수 부담이 크다.
- 개인 PC를 인터넷에 직접 노출하면 보안 위험이 증가한다.

이러한 불편을 해결하기 위해 AWS EC2를 사용한다.

EC2는 안정적인 네트워크 연결, 보안 관리 기능, 자동 확장(오로 스케일링), 부하 분산(로드 밸런싱) 등 다양한 부가 기능을 제공한다.



현업에서의 활용

실무 환경에서도 EC2는 가장 널리 사용되는 서비스 중 하나이다.

특히 백엔드 서버를 배포할 때 EC2 인스턴스에 애플리케이션을 배포하여 운영한다.

예시:

- 스타트업 → API 서버를 EC2에 배포
- 대기업 → 여러 개의 EC2 인스턴스를 Auto Scaling 그룹으로 묶어 대규모 트래픽 대응

프론트엔드 배포와 비교

프론트엔드 웹 페이지 배포에는 EC2를 사용하기도 하지만, 보통은 정적 호스팅 서비스가 더 효율적이다.

- **Vercel, Netlify**: React, Next.js 기반 프로젝트에 특화
- **AWS S3**: 정적 웹 사이트 호스팅 가능

👉 따라서 “EC2는 주로 백엔드 서버를 배포할 때 사용한다”는 정도만 이해해도 충분하다.

5.4.3. 리전(Region) 선택하기

AWS는 전 세계에 걸쳐 여러 개의 리전(Region)을 운영한다.

각 리전은 서로 독립된 데이터센터 집합으로, 고유한 이름을 가진다.

예시: `us-east-1` (미국 동부 버지니아 북부), `ap-northeast-2` (아시아 태평양 서울)

리전 선택 기준

사용자는 네트워크를 통해 애플리케이션에 접근한다.

이때, 사용자와 애플리케이션 서버의 물리적 거리가 멀수록

네트워크 지연(latency)이 발생한다.

따라서, 주요 사용자가 위치한 지역과 가까운 리전을

선택하는 것이 성능상 유리하다.

지역	리전 코드
미국	
버지니아 북부	us-east-1
오하이오	us-east-2
캘리포니아	us-west-1
오레곤	us-west-2
아시아 태평양	
몰바이	ap-south-1
오사카	ap-northeast-3
서울	ap-northeast-2
싱가포르	ap-southeast-1
시드니	ap-southeast-2
도쿄	ap-northeast-1

예시: 한국 사용자가 주로 이용하는 서비스라면, 아시아 태평양(서울, `ap-northeast-2`) 리전을 선택하는 것이 적절하다.

많이 발생하는 실수

AWS 리전은 서로 독립적으로 관리된다.

따라서 서울 리전에서 EC2 인스턴스를 생성해 놓고, 미국 동부(버지니아 북부) 리전 콘솔에 접속하면 생성한 인스턴스가 보이지 않는다.

👉 인스턴스가 사라진 것이 아니라, 단순히 다른 리전을 보고 있는 것이므로 항상 현재 선택된 리전을 확인해야 한다.

5.4.4. EC2 셋팅하기 – 기본 설정

1. 이름 및 태그

이름 및 태그 정보

이름

예: 내 웹 서버

추가 태그 추가

EC2의 이름을 설정하는 곳이다. 이름을 지을 때는 이 컴퓨터가 어떤 역할을 하는 지 알아볼 수 있게 작성해야 한다.

ex) `instagram-server`

2. Application and OS Images (Amazon Machine Image)

Ubuntu 22.04 LTS 선택

▼ 애플리케이션 및 OS 이미지(Amazon Machine Image) 정보

AMI는 인스턴스를 시작하는 데 필요한 소프트웨어 구성(운영 체제, 애플리케이션 서버 및 애플리케이션)이 포함된 템플릿입니다. 아래에서 찾고 있는 항목이 보이지 않으면 AMI를 검색하거나 찾아보세요.

Q 수천 개의 애플리케이션 및 OS 이미지를 포함하는 전체 카탈로그 검색

최근 사용

Quick Start

Amazon Linux

macOS

Ubuntu

Windows

Red Hat

SUSE Linux

Debian

더 많은 AMI 찾아보기
AWS, Marketplace 및 커뮤니티의 AMI 포함

Amazon Machine Image(AMI)

Ubuntu Server 24.04 LTS (HVM), SSD Volume Type

ami-0d5bb3742db8fc264 (64비트(x86)) / ami-064b0e4b6529b20cf (64비트(Arm))

가상화: hvm ENA 활성화됨: true 루트 디바이스 유형: ebs

프리 티어 사용 가능

설명

Ubuntu Server 24.04 LTS (HVM),EBS General Purpose (SSD) Volume Type. Support available from Canonical (<http://www.ubuntu.com/cloud/services>).

Canonical, Ubuntu, 24.04, amd64 noble image

아키텍처

AMI ID

게시 날짜

사용자 이름 ⓘ

확인된 공급 업체

64비트(x86)

ami-0d5bb3742db8fc264

2025-03-05

ubuntu

- OS 선택 단계

- Windows나 Mac OS는 생각보다 용량도 많이 차지하고 성능도 많이 잡아먹는다.

⇒ 서버 배포용 컴퓨터의 OS는 훨씬 가벼운 **Ubuntu**를 많이 사용한다.

277

3. 인스턴스 유형

인스턴스(Instance)는 AWS EC2에서 빌려 사용하는 가상 컴퓨터 한 대를 의미한다.

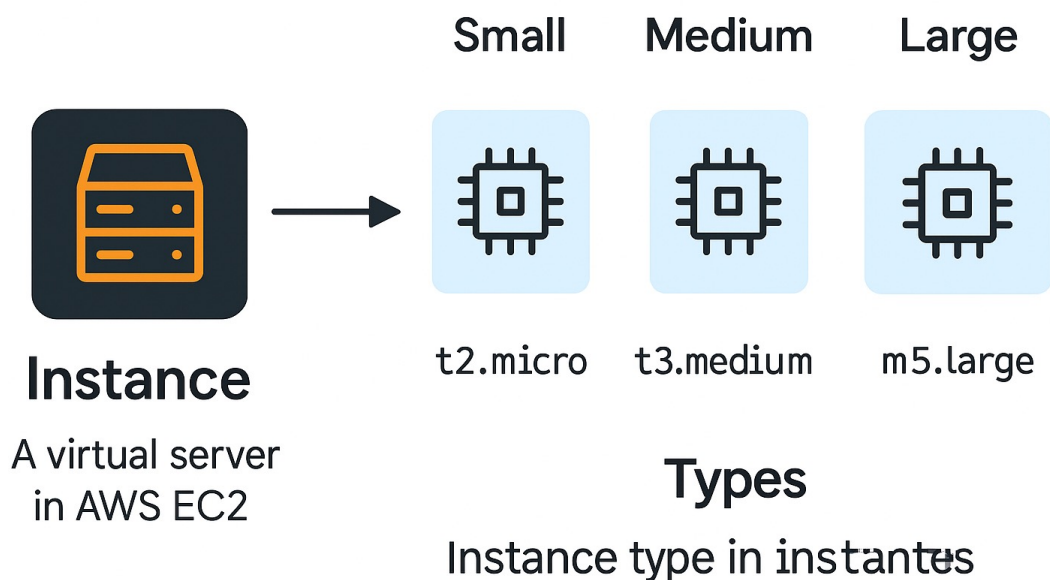
사용자는 EC2 인스턴스를 생성하여 서버를 배포하거나, 애플리케이션을 실행할 수 있다.

인스턴스 유형의 의미

AWS는 다양한 성능과 가격대의 인스턴스 유형을 제공한다.

- 인스턴스 유형은 곧 컴퓨터 사양을 의미한다.
- CPU, 메모리, 네트워크 성능이 높을수록 더 많은 요청을 처리하거나, 무거운 서버 프로그램을 실행할 수 있다.
- 예: t2.micro, t3.medium, m5.large 등

인스턴스 선택은 서비스의 성격, 예상 사용자 수, 비용을 종합적으로 고려해야 한다.



3. 인스턴스 유형

프리 티어와 t2.micro

AWS는 신규 사용자에게 **프리 티어(Free Tier)**를 제공한다.

프리 티어에서는 **t2.micro**라는 인스턴스를 무료로 일정 기간 사용할 수 있다.

- t2.micro 사양: 1 vCPU, 1GB RAM
- 학습, 테스트 환경에 적합하다.

▼ 인스턴스 유형

정보 | 조건 보기

인스턴스 유형

t2.micro

패밀리: t2 1 vCPU 1 GiB 메모리 현재 세대: true 온디맨드 Ubuntu Pro 기본 요금: 0.0162 USD 시간당 프리 티어 사용 가능

온디맨드 RHEL 기본 요금: 0.0288 USD 시간당 온디맨드 Linux 기본 요금: 0.0144 USD 시간당

온디맨드 SUSE 기본 요금: 0.0144 USD 시간당 온디맨드 Windows 기본 요금: 0.019 USD 시간당

모든 세대

인스턴스 유형 비교

소프트웨어가 사전 설치된 AMI에는 추가 비용이 적용됩니다.

많이 하는 오해

t2.micro를 단순히 “아주 낮은 사양이라 실제 서비스에는 못 쓴다”라고 생각하는 경우가 많다.

그러나 실제로는 소규모 트래픽을 처리하기에는 충분하다.

- 예시: 하루 방문자 수 약 2,000명 수준의 서비스가 t2.micro로 무리 없이 운영된 사례가 있다.
- 따라서 성능에 문제가 발생하기 전까지는 굳이 불필요하게 높은 사양으로 시작할 필요가 없다.

핵심은, 서비스 규모에 따라 적절한 인스턴스를 선택하면 된다는 것이다.

4. 키 페어 생성

키 페어란?

키 페어(Key Pair)는 EC2 인스턴스에 로그인할 때 사용하는 비밀번호와 같은 열쇠이다. AWS는 사용자 계정과 별도로, 인스턴스에 직접 접속하기 위한 인증 수단으로 키 페어를 제공한다.

키 페어의 특징

1. 인스턴스 접근 권한 부여

- 키 페어는 해당 EC2 인스턴스에 접속할 수 있는 권한을 가진다.
- 즉, 올바른 키 파일을 소지한 사람만이 서버에 로그인할 수 있다.

2. 키 페어 이름

- 어떤 EC2 인스턴스와 연결된 키 페어인지 쉽게 구분할 수 있도록, 직관적인 이름을 지정하는 것이 좋다.
- 예: myapp-server-key

3. 키 파일 형식

- 키 페어 생성 시 .pem 파일이 다운로드된다.
- 형식에 대한 세부 내용은 깊이 알 필요는 없지만, 이 파일은 절대 잃어버리면 안 된다.
- 분실하면 같은 키로는 다시 접속할 수 없고, 새 키를 만들어 인스턴스에 다시 등록해야 한다.

키 페어 생성 [X]

키 페어 이름
키 페어를 사용하면 인스턴스에 안전하게 연결할 수 있습니다.

 이름에는 최대 255개의 ASCII 문자가 포함됩니다. 앞 또는 뒤에 공백을 포함할 수 없습니다.

키 페어 유형

☒ **RSA**
RSA 암호화된 프라이빗 및 퍼블릭 키 페어

☐ **ED25519**
ED25519 암호화된 프라이빗 및 퍼블릭 키 페어

프라이빗 키 파일 형식

☒ **.pem**
OpenSSH와 함께 사용

☐ **.ppk**
PuTTY와 함께 사용

메시지가 표시되면 프라이빗 키를 사용자 컴퓨터의 안전하고 액세스 가능한 위치에 저장합니다. 나중에 인스턴스에 연결할 때 필요합니다. 자세히 알아보기

취소 키 페어 생성

5.4.5. EC2 셋팅하기 - 보안그룹 설정

1. 네트워크 설정

▼ 네트워크 설정 정보

네트워크 | 정보
vpc-0766069eb1042d7a2

서브넷 | 정보
기본 설정 없음(가용 영역의 기본 서브넷)

퍼블릭 IP 자동 할당 | 정보
활성화
프리 티어 허용 범위를 벗어나는 경우 추가 요금이 적용됩니다.

방화벽(보안 그룹) | 정보
보안 그룹은 인스턴스에 대한 트래픽을 제어하는 방화벽 규칙 세트입니다. 특정 트래픽이 인스턴스에 도달하도록 허용하는 규칙을 추가합니다.

☒ 보안 그룹 생성 ☐ 기존 보안 그룹 선택

다음 규칙을 사용하여 'launch-wizard-2'(이)라는 새 보안 그룹을 생성합니다.

☒ 다음에서 SSH 트래픽 허용
인스턴스 연결에 도움 위치 무관 0.0.0.0/0

☐ 인터넷에서 HTTPS 트래픽 허용
예를 들어 웹 서버를 생성할 때 엔드포인트를 설정하려면

☐ 인터넷에서 HTTP 트래픽 허용
예를 들어 웹 서버를 생성할 때 엔드포인트를 설정하려면

⚠ 소스가 0.0.0.0/0인 규칙은 모든 IP 주소에서 인스턴스에 액세스하도록 허용합니다. 알려진 IP 주소의 액세스만 허용하도록 보안 그룹을 설정하는 것이 좋습니다. ✕

EC2 인스턴스를 생성할 때 네트워크 설정(Network Settings) 단계가 나타난다.

편집 버튼을 누르면 다음과 같은 항목들을 확인할 수 있다.

- VPC (Virtual Private Cloud)
- 서브넷(Subnet)
- 퍼블릭 IP 자동 할당 여부
- 방화벽(Security Group)

이 중에서 보안 그룹(Security Group)은 서버에 접근할 수 있는 방화벽 규칙을 의미하며, 반드시 설정해야 하는 부분이다.

5.4.5. EC2 셋팅하기 - 보안그룹 설정

1. 네트워크 설정

The screenshot shows the '보안 그룹 생성' (Create Security Group) page in the AWS Management Console. The page is for a VPC named 'vpc-0766069eb1042d7a2' with CIDR '172.31.0.0/16'. The '서브넷' (Subnet) is set to '기본 설정 없음' (No default). The '퍼블릭 IP 자동 할당' (Automatic public IP) is set to '활성화' (Enabled). The '방화벽(보안 그룹)' (Firewall (Security Group)) section shows '보안 그룹 생성' (Create security group) selected. The '보안 그룹 이름' (Security group name) is 'launch-wizard-2'. The '설명' (Description) is 'launch-wizard-2 created 2025-04-16T07:56:39.720Z'. The '인바운드 보안 그룹 규칙' (Inbound security group rules) section shows a single rule for 'ssh' (port 22) using 'TCP' protocol. The '소스 유형' (Source type) is '위치 무관' (Anywhere). The '원본' (Source) is 'CIDR, 접두사 목록 또는 보안 그룹 추가' (Add CIDR, prefix list, or security group). The '설명 - 선택 사항' (Description - optional) is '예: 관리자 데스크톱용 SSH' (Example: SSH for administrator desktop).

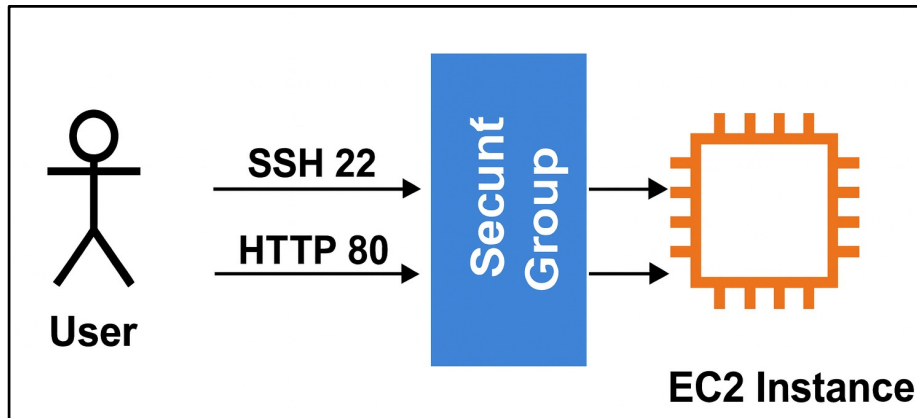
네트워크 설정 화면에 VPC라는 항목도 보이지만, AWS 입문 단계에서는 크게 신경 쓰지 않아도 된다.

- VPC는 AWS에서 네트워크를 가상으로 분리하는 기능이다.
 - 하지만 EC2 인스턴스를 처음 생성하는 과정에서는 기본 VPC가 자동으로 선택되므로, 특별히 설정하지 않아도 서버를 배포하는 데 문제가 없다.
 - VPC에 대한 심화 학습은 AWS 서비스 구조에 익숙해진 뒤에 진행하면 된다.
- 따라서 **보안 그룹(Security Group) 설정만 집중해도 충분하다.**

5.4.5. EC2 셋팅하기 - 보안그룹 설정

2. 보안 그룹(Security Group)

보안 그룹(Security Group)이란 AWS 클라우드에서의 네트워크 보안을 의미한다.



EC2 인스턴스를 집이라고 생각한다면, 보안 그룹은 집 바깥 쪽에 쳐져있는 울타리와 대문이라고 생각하면 된다.

집에 접근할 때 울타리의 대문에서 접근해도 되는 요청인지 보안 요원이 검사를 하는 것과 비슷하다.

인터넷에서 일부 사용자가 EC2 인스턴스에 접근(액세스)하려고 한다고 가정해보자.

- 위 그림과 같이 EC2 인스턴스 주위에 방화벽 역할을 할 보안 그룹(Security Group)을 만들고 보안 그룹에 규칙을 지정한다.
- 이 보안 규칙에는 인바운드 트래픽(즉, 외부에서 EC2 인스턴스로 보내는 트래픽)에서 어떤 트래픽만 허용할 지 설정할 수 있고,
- 아웃바운드 트래픽(즉, EC2 인스턴스에서 외부로 나가는 트래픽)에서 어떤 트래픽만 허용할 지 설정할 수 있다.

5.4.5. EC2 셋팅하기 - 보안그룹 설정

2. 보안 그룹 설정

보안 그룹을 설정할 때는 허용할 IP 범위와 포트(port)를 설정할 수 있다.

EC2 인스턴스를 생성할 때 어떻게 보안 그룹(Security Group)을 설정해야 하는지 알아보자.

인바운드 보안 그룹 규칙

▼ 보안 그룹 규칙 1 (TCP, 22, 0.0.0.0/0) 제거

유형	정보	프로토콜	정보	포트 범위	정보
ssh		TCP		22	

소스 유형 정보 원본 정보 설명 - 선택 사항 정보

위치 무관 원본 예: 관리자 데스크톱용 SSH

0.0.0.0/0 ×

▼ 보안 그룹 규칙 2 (TCP, 80) 제거

유형	정보	프로토콜	정보	포트 범위	정보
HTTP		TCP		80	

소스 유형 정보 원본 정보 설명 - 선택 사항 정보

사용자 지정 원본 예: 관리자 데스크톱용 SSH

0.0.0.0/0 ×

⚠ 소스가 0.0.0.0/0인 규칙은 모든 IP 주소에서 인스턴스에 액세스하도록 허용합니다. 알려진 IP 주소의 액세스만 허용하도록 보안 그룹을 설정하는 것이 좋습니다. ×

외부에서 EC2로 접근할 포트는 **22번 포트**와 **80번 포트**라고 생각해서 이 2가지에 대해 인바운드 보안 그룹 규칙을 추가했다.

- 22번 포트는 우리가 EC2에 원격 접속할 때 사용하는 포트이고, 80번 포트에는 백엔드 서버를 띄울 예정이기 때문이다.
- 어떤 IP에서든 전부 접근할 수 있게 만들기 위해 소스 유형은 위치 무관으로 설정했다.

5.4.6. EC2 셋팅하기 - 스토리지 구성

스토리지가 필요한 이유

우리가 사용하는 노트북이나 데스크톱 컴퓨터에는 모두 하드디스크(HDD/SSD)가 있다.

하드디스크는 운영체제, 프로그램, 데이터 파일을 저장하는 필수 공간이다.

EC2 인스턴스도 하나의 컴퓨터이므로 파일을 저장할 저장 장치(Storage)가 필요하다.

AWS에서 이를 담당하는 서비스가 EBS(Elastic Block Store)이다.

▼ 스토리지(볼륨) 정보 단순

EBS 볼륨 세부 정보 숨기기

▶ 볼륨 1 (AMI 루트)(사용자 지정) : 8 GiB, EBS, 범용 SSD(gp3), 3000 IOPS

① 프리 티어를 사용할 수 있는 고객은 최대 30GB의 EBS 범용(SSD) 또는 마그네틱 스토리지를 사용할 수 있습니다. ×

새 볼륨 추가

선택한 AMI에 인스턴스가 허용하는 것보다 많은 인스턴스 스토어 볼륨이 포함되어 있습니다. AMI에서 처음 0개의 인스턴스 스토어 볼륨에만 액세스할 수 있습니다.

⌚ 백업 정보를 보려면 새로 고침 클릭 ↻
 할당된 태그에 따라 Data Lifecycle Manager 정책으로 인스턴스를 백업할지 여부가 결정됩니다.

파일 시스템 세부 정보 숨기기

☒ EFS ☐ FSx

현재 이 인스턴스에 파일 시스템이 없습니다. EFS 파일 시스템을 추가하려면 먼저 서브넷을 선택해야 합니다.

EBS(Elastic Block Store)

- EBS는 EC2 인스턴스에 연결되는 가상 하드디스크이다.
- 운영체제, 애플리케이션, 데이터 파일을 저장한다.
- 하나의 인스턴스에 여러 개의 EBS 볼륨을 연결할 수 있다.
- 이러한 저장 공간을 포괄적으로 스토리지(Storage), 또는 볼륨(Volume)이라고 부른다.

EBS는 EC2 인스턴스가 종료되더라도 데이터를 유지할 수 있다는 장점이 있다.

5.4.6. EC2 셋팅하기 - 스토리지 구성

볼륨 유형

볼륨 1 (AMI 루트)(사용자 지정)

스토리지 유형 | 정보

EBS

디바이스 이름 - 필수 | 정보

/dev/sda1

스냅샷 | 정보

snap-02d61981ad5b98ac4

크기(GiB) | 정보

30

볼륨 유형 | 정보

gp3

IOPS | 정보

3000

종료 시 삭제 | 정보

예

암호화됨 | 정보

암호화되지 않음

KMS 키 | 정보

선택

이 볼륨에서 암호화가 설정된 경우에만 KMS 키를 적용할 수 있습니다.

AWS는 다양한 볼륨 유형을 제공한다. 그중 gp3(General Purpose SSD)는 가성비가 가장 좋은 범용 볼륨이다.

- gp3: 범용 SSD, 대부분의 서비스에 적합
- io2: 높은 IOPS(초당 입출력 작업)가 필요한 고성능 DB용
- st1: 대규모 순차적 데이터 처리용 (예: 로그 저장)
- sc1: 저비용 아카이빙 용도

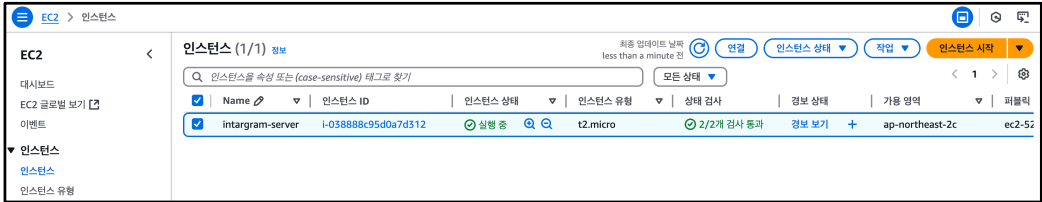
학습이나 서비스 초기 단계에서는 gp3를 선택하는 것이 적절하다.

스토리지 크기

- 프리 티어에서는 최대 30GB까지 무료로 사용할 수 있다.
- 따라서 실습에서는 30GB로 설정하는 것이 일반적이다.
- 스토리지 크기는 추후에 확장 가능하므로, 처음부터 지나치게 크게 잡을 필요는 없다.

5.4.7. EC2 접속하기

1. 생성된 인스턴스 정보 확인하기 – 세부 정보



인스턴스의 세부 정보를 확인하기 위해 인스턴스 ID를 누른다.



이러한 정보들이 있는데, 다 이해하고 사용하려면 시간이 많이 소요된다.

여기서 알아야 할 것은 퍼블릭 IPv4 주소와 인스턴스 상태이다.

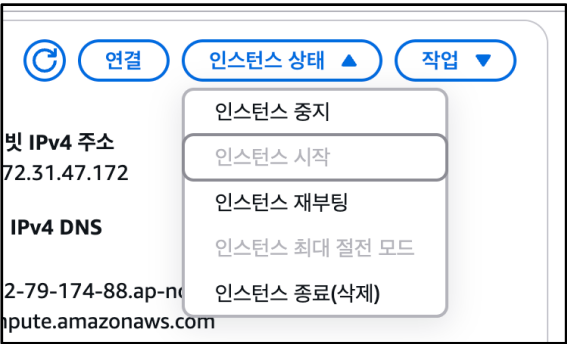
- 퍼블릭 IPv4 주소는 EC2 인스턴스가 생성되면서 부여받은 IP 주소이다. EC2 인스턴스에 접근하려면 이 IP 주소로 접근하면 된다.
- 인스턴스 상태는 말 그대로 EC2 인스턴스가 실행 중이라는 뜻은 컴퓨터가 켜져 있다는 뜻이다.

5.4.7. EC2 접속하기

EC2 인스턴스를 중지, 재부팅, 종료도 할 수 있다.

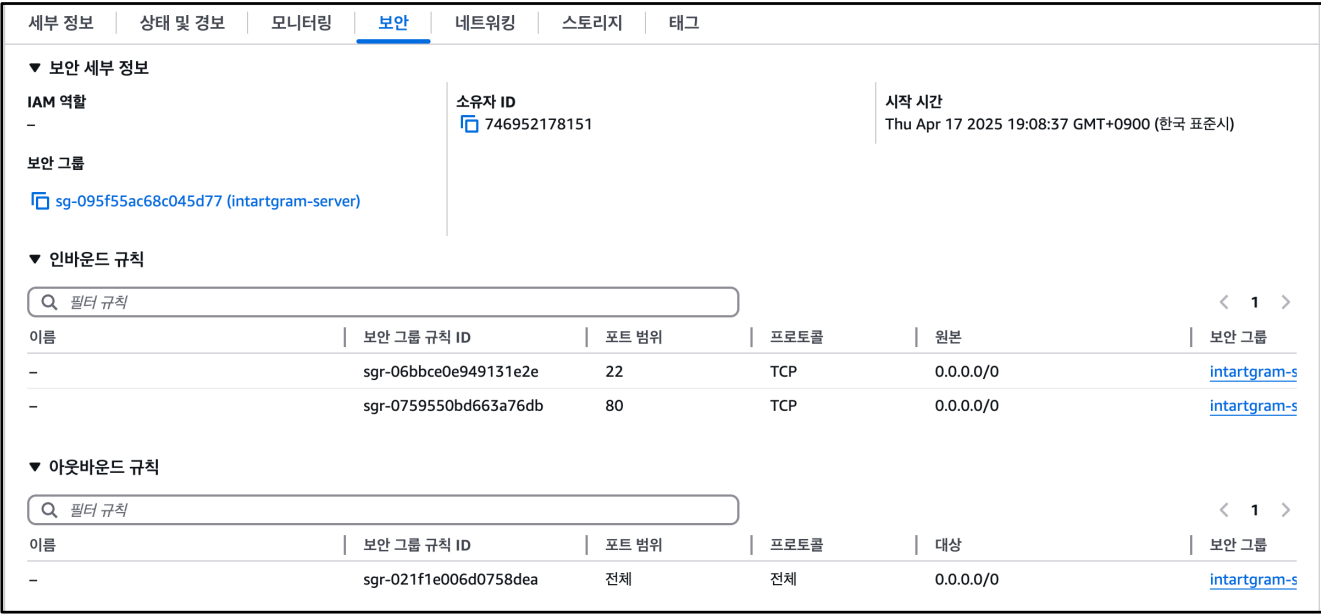
→ 컴퓨터와 같다고 생각하면 된다

- 재부팅: 컴퓨터 재시작
- 중지: 컴퓨터 잠시 꺼놓기
- 종료: 컴퓨터 아예 삭제



EC2 인스턴스를 한 번 종료하면 도중에 취소할 수 없으니 조심해야 한다.

2. 보안그룹



인스턴스 생성 시 설정한 보안 그룹에 대한 정보가 나온다.

6.4.7. EC2 접속하기

3. 네트워킹

세부 정보 | 상태 및 경보 | 모니터링 | 보안 | 네트워킹 | 스토리지 | 태그

▼ 네트워킹 세부 정보 정보

퍼블릭 IPv4 주소
52.79.174.88 | [개방 주소법](#)

퍼블릭 IPv4 DNS
ec2-52-79-174-88.ap-northeast-2.compute.amazonaws.com
| [개방 주소법](#)

서브넷 ID
subnet-0a45ee2dd1fdf6856

가용 영역
ap-northeast-2c

RBN을 게스트 OS 호스트 이름으로 사용
비활성

프라이빗 IPv4 주소
172.31.47.172

프라이빗 IP DNS 이름(IPv4만 해당)
ip-172-31-47-172.ap-northeast-2.compute.internal

IPv6 주소
-

통신 사업자 IP 주소(임시)
-

RBN DNS 호스트 이름 IPv4 응답
활성

VPC ID
vpc-0766069eb1042d7a2

보조 프라이빗 IPv4 주소
-

Outpost ID
-

▼ 네트워크 인터페이스 (1) 정보

네트워크 인터페이스 필터링

인터페이스 ID	디바이스 인덱스	카드 인덱스	설명	퍼블릭 IPv4 주소	프라이빗 IPv4 주소	프라이빗 IPv4 DNS
eni-09b9750896ad09e15	0	0	-	52.79.174.88	172.31.47.172	ip-172-31-47-172.

52.79.174.88이 우리가 빌려온 EC2 인스턴스 컴퓨터의 주소다.

4. 스토리지

세부 정보 | 상태 및 경보 | 모니터링 | 보안 | 네트워킹 | 스토리지 | 태그

▼ 루트 디바이스 세부 정보

루트 디바이스 이름
/dev/sda1

루트 디바이스 유형
EBS

EBS 최적화
비활성

▼ 블록 디바이스

블록 디바이스 필터링

☒ 볼륨 ID	디바이스 이름	볼륨 크기(GiB)	볼륨 상태	연결 상태	연결 시간
☒ vol-00a2524d58983964c	/dev/sda1	30	사용 중	연결됨	2025/04/17 19:08 GMT+9

인스턴스 생성 시 설정한 스토리지에 대한 정보가 나온다.

5. 상태검사

이부분은 잘 안 보기 때문에, 넘어가겠다.

289

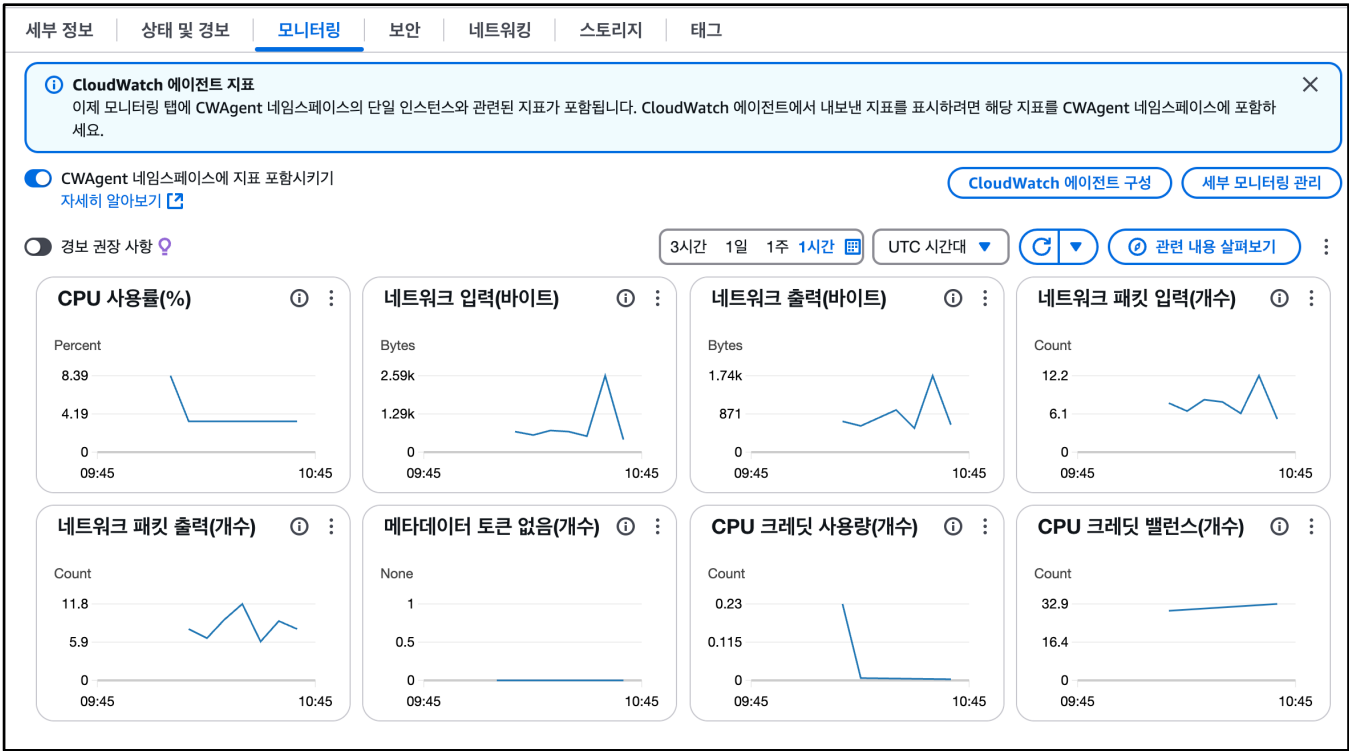
5.4.7. EC2 접속하기

6. 모니터링

EC2 인스턴스에 관련한 여러가지 정보를 볼 수 있는 창이다.

AWS를 처음 입문할 때는 자주 볼 일이 없는 창이지만, 나중에 실제 서버를 운영할 때는 자주 보게되는 창이다.

- EC2 인스턴스가 정상적으로 작동하고 있는지, EC2 인스턴스의 성능을 향상시켜주어야 하는 건 아닌 지 아래 지표를 통해 파악할 수 있다.



7. 태그

잘 보지 않는 창이다.

세부 정보	상태 및 경보	모니터링	보안	네트워킹	스토리지	태그
태그						
Key ▾ Value						
Nameintargram-server						

5.4.7. EC2 접속하기

인스턴스에 연결
정보

다음 옵션 중 하나를 사용하여 인스턴스 i-038888c95d0a7d312 (intargram-server)에 연결

EC2 인스턴스 연결
Session Manager
SSH 클라이언트
EC2 직렬 콘솔

인스턴스 ID
i-038888c95d0a7d312 (intargram-server)

연결 유형

☒ EC2 Instance Connect를 사용하여 연결
퍼블릭 IPv4 또는 IPv6 주소가 있는 EC2 인스턴스 연결 브라우저 기반 클라이언트를 사용하여 연결합니다.

☐ EC2 인스턴스 연결 엔드포인트를 사용하여 연결
프라이빗 IPv4 주소 및 VPC 엔드포인트가 있는 EC2 인스턴스 연결 브라우저를 기반 클라이언트를 사용하여 연결합니다.

☒ 퍼블릭 IPv4 주소
52.79.174.88

☐ IPv6 주소
-

사용자 이름
인스턴스를 시작하는 데 사용되는 AMI에 정의된 사용자 이름을 입력합니다. 사용자 지정 사용자 이름을 정의하지 않은 경우 기본 사용자 이름인 ubuntu를(를) 사용합니다.

참고: 대부분의 경우 기본 사용자 이름 ubuntu(는) 정확합니다. 하지만 AMI 사용 지침을 읽고 AMI 소유자가 기본 AMI 사용자 이름을 변경했는지 확인하세요.

취소
연결

AWS EC2 Instance Connect는 AWS 콘솔(웹 브라우저)을 통해 인스턴스에 접속할 수 있는 기능이다.

즉, 별도의 프로그램(예: 터미널, SSH 클라이언트)을 설치하지 않아도 AWS 웹 콘솔 안에서 바로 원격 접속이 가능하다.

```

System load:  0.0          Processes:    104
Usage of /:   6.0% of 28.02GB Users logged in: 0
Memory usage: 21%        IPv4 address for enX0: 172.31.47.172
Swap usage:   0%

Expanded Security Maintenance for Applications is not enabled.

0 updates can be applied immediately.

Enable ESM Apps to receive additional future security updates.
See https://ubuntu.com/esm or run: sudo pro status

The list of available updates is more than a week old.
To check for new updates run: sudo apt update

The programs included with the Ubuntu system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent permitted by
applicable law.

To run a command as administrator (user "root"), use "sudo <command>".
See "man sudo_root" for details.

ubuntu@ip-172-31-47-172:~$

i-038888c95d0a7d312 (intargram-server)
PublicPs: 52.79.174.88  PrivatePs: 172.31.47.172

```

5.4.8 탄력적 IP 연결하기

왜 탄력적 IP가 필요할까?

EC2 인스턴스를 생성하면 자동으로 퍼블릭 IP가 할당된다.

그러나 이 IP는 임시적이어서, 인스턴스를 중지 후 다시 실행하면 새로운 IP가 부여된다.

인스턴스의 IP가 실행할 때마다 바뀌면, 사용자는 접속할 주소를 계속 수정해야 하므로 불편하다.

따라서 운영 환경에서는 고정된 퍼블릭 IP를 설정하는 것이 필요하기 때문에 사용하는 것이 바로 탄력적 IP(Elastic IP)이다.

탄력적 IP란?

탄력적 IP는 중지/재시작해도 변하지 않는 고정 퍼블릭 IPv4 주소이다.

- 인스턴스에 할당하면 언제나 동일한 주소로 접속할 수 있다.
- 서버를 교체하거나 재배포하더라도, 탄력적 IP를 새 인스턴스에 연결하면 같은 주소를 그대로 사용할 수 있다.

현업에서는 EC2 인스턴스를 생성할 때 탄력적 IP를 필수적으로 설정하는 경우가 많다.



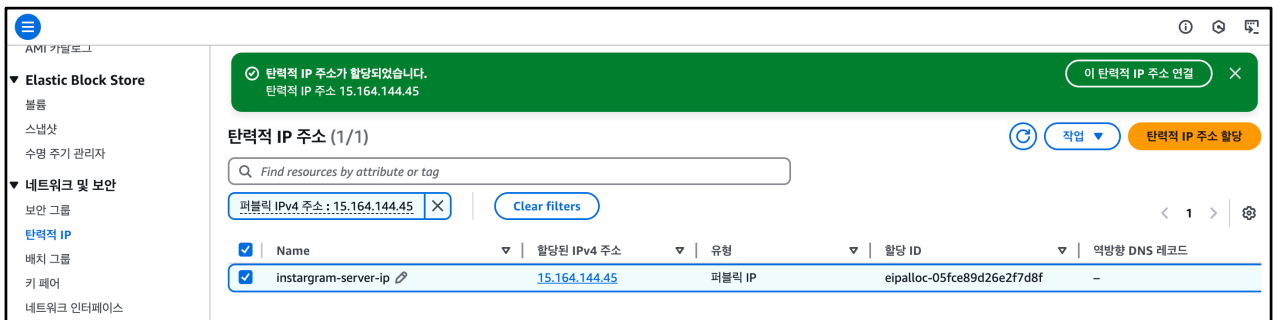
그렇다면 왜 AWS는 기본 IP를 고정하지 않고 임시로 부여할까?”라는 의문이 생긴다.

IPv4 주소는 한정된 자원이다.

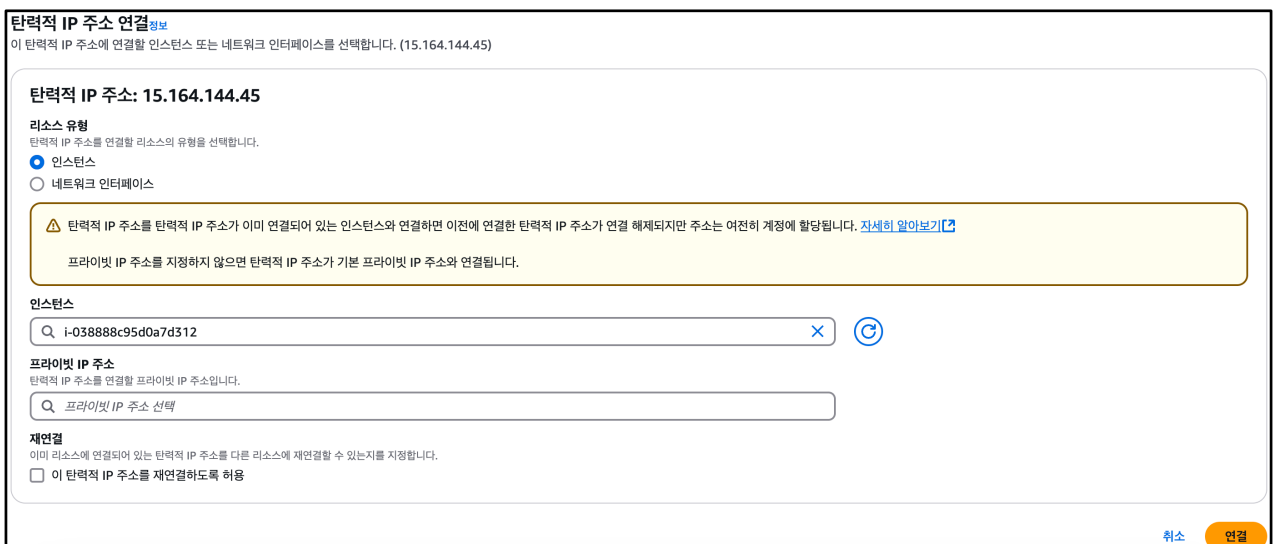
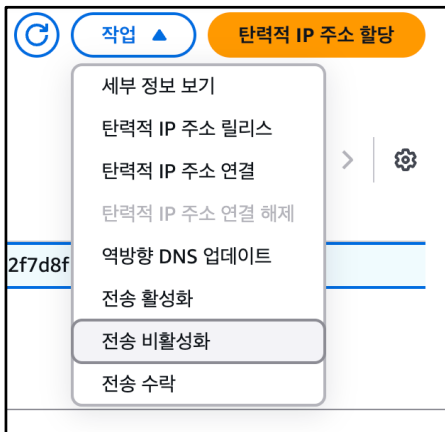
전 세계적으로 인터넷 사용이 증가하면서, 모든 컴퓨터와 기기에 IP를 고정적으로 할당하기에는 수량이 부족하다.

따라서 AWS는 임시 IP 방식을 통해 사용하지 않는 IP를 다른 사용자에게 재활용한다.

5.4.8 탄력적 IP 연결하기



탄력적 IP 주소를 연결할 땐 → 탄력적 IP 주소 연결을 누르고



만든 EC2 인스턴스를 누르고 연결하면 끝이다.

5.4.9 Express 서버를 EC2에 배포하기

1. Ubuntu 환경에서 Node.js 설치하는 법

Ubuntu 환경에서 Express 서버를 실행시키려면 Node.js가 설치되어 있어야 한다.

```
$ sudo su
$ apt-get update && /
apt-get install -y ca-certificates curl gnupg && /
mkdir -p /etc/apt/keyrings && /
curl -fsSL https://deb.nodesource.com/gpgkey/nodesource-repo.gpg.key | sudo gpg --dearmor -
o /etc/apt/keyrings/nodesource.gpg && /
NODE_MAJOR=20 && /
echo "deb [signed-by=/etc/apt/keyrings/nodesource.gpg]
https://deb.nodesource.com/node_${NODE_MAJOR}.x nodistro main" | sudo tee
/etc/apt/sources.list.d/nodesource.list && /
apt-get update && /
apt-get install nodejs -y
```

2. 잘 설치됐는 지 확인하기

```
$ node -v
```

```
root@ip-172-31-47-172:/home/ubuntu# node -v
v20.19.0
```

3. 80번 포트에서 프로젝트 실행하기

app.js

```
require('dotenv').config();
const express = require('express');
const app = express();
const port = 80;

app.get('/', (req, res) => {
  res.send(`.env 테스트 : ${process.env.DATABASE_NAME}`);
})

app.listen(port, () => {
  console.log(`Example app listening on port ${port}`)
})
```

5.4.9 Express 서버를 EC2에 배포하기

4 .env 파일 직접 만들기

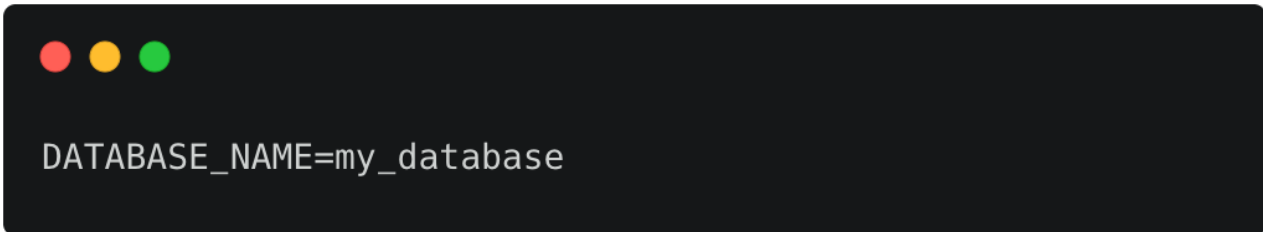
.env와 같은 민감한 파일은 Git으로 버전 관리를 하지 않는게 일반적이다. 따라서 .env 파일은 별도로 EC2 인스턴스에 올려주어야 한다.

하지만 .env 파일을 EC2 인스턴스에 올리는 작업보다는, .env 파일을 직접 만드는 게 훨씬 간단하다.



```
$ ubuntu@ip-172-31-47-172:~/ec2-express-sample$ ls
$ app.js  node_modules  package-lock.json  package.json
$ ubuntu@ip-172-31-47-172:~/ec2-express-sample$ vi .env
```

.env



```
DATABASE_NAME=my_database
```

5.4.9 Express 서버를 EC2에 배포하기

5. pms2 설치해서 서버 실행시키기

Node 기반의 서버는 'pm2'를 활용해서 많이 실행시킨다.

서비스를 운영하는 데 있어서 유용한 기능들을 'pm2'가 많이 가지고 있기 때문이다.

```
$ sudo npm i -g pm2
$ sudo pm2 start app.js
```

동작화면

```
[PM2] Spawning PM2 daemon with pm2_home=/root/.pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /home/ubuntu/ec2-express-sample/app.js in fork_mode (1 instance)
[PM2] Done.
```

id	name	namespace	version	mode	pid	uptime	⌵	status	cpu	mem	user	watching
0	app	default	1.0.0	fork	2904	0s	0	online	0%	29.1mb	root	disabled

6. 잘 작동하는 지 확인하기

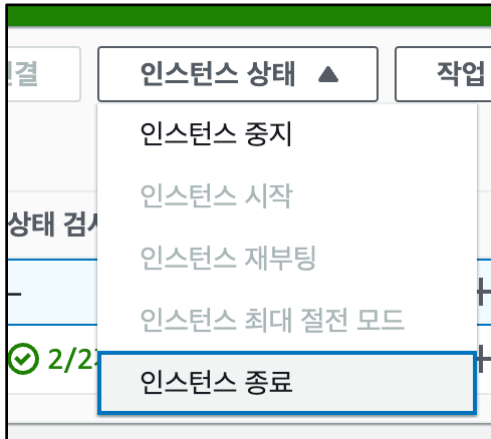
← → ↻ ⚠ 주의 요함 15.164.144.45

.env 테스트 : undefined

내 주소를 검색창에 치면 잘 동작하는 것을 확인할 수 있다.

5.4.10. 비용 나가지 않게 EC2 깔끔하게 종료하기

1. EC2 인스턴스 종료하기



2. 탄력적 IP 릴리스하기.

탄력적 IP 주소가 연결되었습니다.
탄력적 IP 주소 15.164.144.45(가) 인스턴스 i-038888c95d0a7d312에 연결되었습니다.

탄력적 IP 주소 (1/1)

Find resources by attribute or tag

☒	Name	할당된 IPv4 주소	유형	할당 ID
☒	instagram-server-ip	15.164.144.45	퍼블릭 IP	eipalloc-05fce89d26e2f7d8f

세부 정보 보기
탄력적 IP 주소 릴리스
탄력적 IP 주소 연결
탄력적 IP 주소 연결 해제
역방향 DNS 업데이트

탄력적 IP 주소 릴리스

릴리스 예정

다음 탄력적 IP 주소를 릴리스할 경우, 해당 IP 주소는 더 이상 계정에 할당되지 않으며 리소스에 연결할 수 없습니다.

이름	IPv4 주소	할당 ID	주소 유형
instagram-server-ip	15.164.144.45	eipalloc-05fce89d26e2f7d8f	Public

취소 릴리즈

5-5 보충: 서버가 잘 돌아가는지 확인하기

서버가 잘 돌아가는지 확인하기 위해 아래 6가지 조건을 테스트 해보려고 한다.

1. DB에 값 넣어보기
2. 임의 api 만들어서 테스트 해보기
3. 도커 build,
4. 서버에 해당 build된 거 서버에 적용시키기
5. 잘 작동하는지 확인하기
6. 데이터 안 날라갔는지, DB 괜찮은지

5-5 보충: 서버가 잘 돌아가는지 확인하기

1. DB에 값 넣어 보기

Test user를 넣을 것이다.

POST: <http://43.200.201.172/api/users/signup>

```
{
  "email": "test@example.com",
  "password": "password123~",
  "username": "TestUser",
  "phoneNumber": "01012345678",
  "birthDate": "1990-01-01",
  "gender": "MALE"
}
```

Nginx → Spring Boot로 프록시



5-5 보충: 서버가 잘 돌아가는지 확인하기

임의의 api 만들어서 테스트 해보기

다음으로 임의의 API를 만들어서 테스트 해보겠다.

그냥 간단한 api 만들 것이다.

- GET 요청으로 문자열 응답
- Nginx 경유해서 브라우저, Postman에서 확인

별도의 테스트 폴더를 하나 만들어 주고 api 코드를 작성한다.

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class SampleController {

    @GetMapping("/api/hello")
    public String hello() {
        return "Hello Docker World!";
    }
}
```


5-5 보충: 서버가 잘 돌아가는지 확인하기

임의의 api 만들어서 테스트 해보기

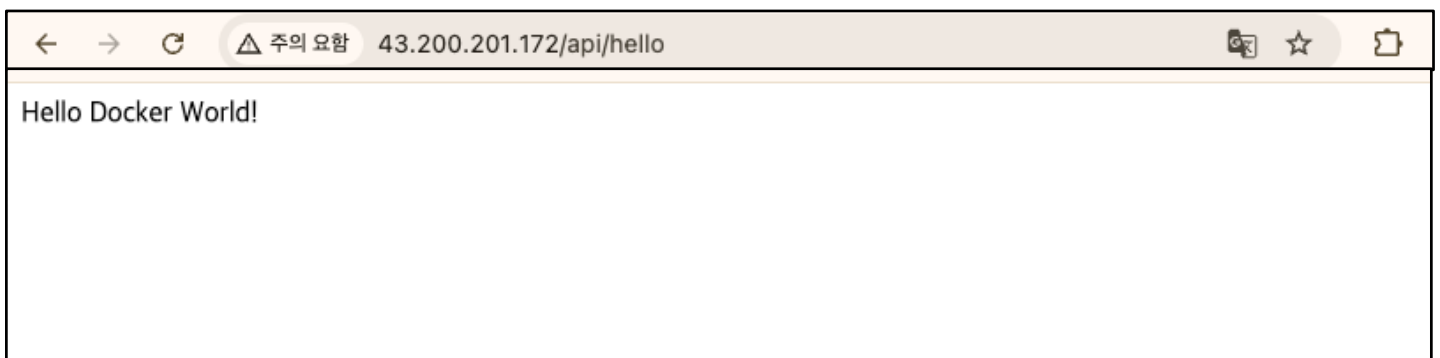
api를 작성했으면, 빌드를 해준다.

```
cd ~/project
#Gradle 빌드
./gradlew build
## 만약 테스트 없이 JAR만 새로 빌드하고 싶다면
./gradlew build -x test

#Docker Compose 빌드
cd ~/project-root
docker compose build springboot

#컨테이너 재시작
docker compose up -d

#로그 확인
docker logs springboot
```



정상적으로 실행이 완료됐다.

5-5 보충: 서버가 잘 돌아가는지 확인하기

마지막 스텝

데이터가 날라가진 않았는지, DB 괜찮은지

먼저 MySQL CLI에 들어간다.

```
docker exec -it mysql mysql -u root -p
```

```
#DB 선택
USE tcar;
```

```
#테이블 목록 확인
SHOW TABLES;
```

```
#user 내용 조회
SELECT * FROM users;
```

```
#레코드 카운트 확인 - 데이터 개수 체크
SELECT COUNT(*) FROM users;
```

```
mysql> SELECT * FROM user;
```

id	birth_date	email	gender	nickname	password	phone_number	role	username
1	1990-01-01	test@example.com	MALE	TestUser	\$2a\$10\$yDPRaHBkAI1QEDqgtu7p.OPON.ilqaUNon6HKNF.74vztywkIu4Hy	01012345678	1	TestUser

```
1 row in set (0.00 sec)
```

정상적으로 잘 들어와 있는 것을 확인할 수 있다.