TCP/IP는 Transmission Control Protocol/Internet Protocol의 약자로 인터넷에서 컴퓨터들이 서로 정보를 주고받을 때 사용되는 통신 규약의 모음이다. 마치 전 세계 사람들이 서로 다른 언어를 사용하지만 특정한 규칙(언어)을 정해 놓고 소통하는 것처럼 컴퓨터들도 TCP/IP라는 공통의 규칙을 사용하여 데이터를 주고받는다.

TCP/IP를 사용하는 이유

1. 세계적인 네트워크 연결 : 다양한 종류의 컴퓨터와 운영체제를 사용하는 네트워크 환경에서도 데이터를 효율적으로 주고받을 수 있도록 해준다.
2. 데이터 전송의 신뢰성 : 데이터가 정확하게 목적지까지 전달될 수 있도록 오류 검출 및 재전송 기능을 제공한다.
3. 데이터 분할 및 재조립 : 큰 데이터를 작은 조각으로 나누어 전송하고 목적지에서 다시 조립하여 원래의 데이터를 복원한다.

TCP/IP 모델의 자세한 계층 구조와 각 계층의 역할

TCP/IP 모델은 네트워크 통신을 4개의 계층으로 나누어 설명한다. 각 계층은 서로 다른 역할을 수행하며 상위 계층은 하위 계층의 서비스를 이용하여 데이터를 전송한다.

1. 애플리케이션 계층 (Application Layer)

   역할 : 사용자와 직접 상호 작용하는 애플리케이션들이 존재하는 계층이다. 사용자 데이터를 생성하고 해석하며 네트워크 서비스를 요청한다.

   대표적인 프로토콜 : HTTP (웹), FTP (파일 전송), SMTP (이메일), DNS (도메인 이름 시스템) 등

   주요 기능 :
   - 사용자 인터페이스 제공
   - 데이터 형식 변환
   - 네트워크 서비스 요청 (예: 웹 페이지 요청, 파일 다운로드)
2. 전송 계층 (Transport Layer)
   역할 : 애플리케이션 계층과 네트워크 계층 사이에서 데이터를 주고받는 서비스를 제공한다. 데이터의 신뢰성, 순서, 흐름 제어 등을 담당한다.

   대표적인 프로토콜 : TCP (Transmission Control Protocol), UDP (User Datagram Protocol)

   주요 기능 :
   - 데이터 세그먼트화
   - 오류 검출 및 재전송
   - 흐름 제어
   - 포트 번호 할당

3. 네트워크 계층 (Internet Layer)
   역할 : 데이터를 패킷으로 분할하고 각 패킷에 목적지 주소를 부여하여 네트워크를 통해 전송한다.

   대표적인 프로토콜 : IP (Internet Protocol)

   주요 기능 :
   - 패킷 라우팅
   - 네트워크 주소 할당
   - 패킷 조각화 및 재조립
4. 링크 계층 (Network Access Layer)
   역할 : 물리적인 네트워크를 통해 데이터를 전송하는 기능을 제공한다.

   대표적인 프로토콜 : Ethernet, Wi-Fi

   주요 기능 :
   - 물리적 매체 접근
   - 데이터 프레임 생성 및 전송
   - 오류 검출 (물리적 계층에서 발생하는 오류)

각 계층의 상호 작용

1. **애플리케이션 계층:** HTTP 요청을 생성하여 전송 계층에 전달합니다.
2. **전송 계층:** HTTP 요청을 TCP 세그먼트로 포장하고, 순서 번호와 확인 번호를 부여하여 네트워크 계층에 전달합니다.
3. **네트워크 계층:** TCP 세그먼트를 IP 패킷으로 포장하고, 목적지 IP 주소를 부여하여 링크 계층에 전달합니다.
4. **링크 계층:** IP 패킷을 Ethernet 프레임으로 포장하여 물리적 매체를 통해 전송합니다.

TCP와 IP의 역할

1. TCP(Transmission Control Protocol)
   연결 지향형 프로토콜
   데이터의 순서를 보장하고 오류 검출 및 재전송을 수행
   신뢰성 있는 데이터 전송을 보장
2. IP(Internet Protocol)
   비연결형 프로토콜
   데이터를 패킷으로 분할하고 목적지 주소를 부여
   패킷을 네트워크를 통해 전달

TCP/IP의 동작 과정

1. 애플리케이션 계층 : 사용자 데이터를 생성하고 TCP/IP 스택에 전달한다.

2. 전송 계층 : 데이터를 TCP 세그먼트로 포장하고 순서 번호와 확인 번호를 부여한다.
3. 네트워크 계층 : TCP 세그먼트를 IP 패킷으로 포장하고 IP 주소를 부여한다.
4. 링크 계층 : IP 패킷을 물리적인 네트워크를 통해 전송한다.
5. 목적지 : 위 과정을 역순으로 수행하여 원래의 데이터를 복원한다.

TCP/IP의 중요성

TCP/IP는 현대 인터넷의 기반이 되는 핵심 기술이다. 웹 서핑, 이메일, 파일 전송 등 우리가 일상적으로 사용하는 많은 인터넷 서비스들이 TCP/IP 프로토콜을 기반으로 동작한다.

UDP(User Datagram Protocol)는 TCP와 함께 인터넷 프로토콜의 주요 프로토콜 중 하나이다. TCP가 연결 지향형 프로토콜이라면 UDP는 비연결형 프로토콜이다. 데이터를 전송하기 전에 별도의 연결을 설정하지 않고 각 데이터 패킷을 독립적으로 전송합니다.

UDP의 특징

1. 비연결형 : 연결을 설정하지 않고 데이터를 바로 전송한다.
2. 데이터그램 : 데이터를 고정된 크기의 데이터그램 단위로 전송한다.
3. 빠른 속도 : 연결 설정 과정이 없어 TCP보다 빠른 전송 속도를 제공한다.
4. 낮은 오버헤드 : TCP에 비해 헤더 정보가 적어 오버헤드가 적다.
5. 신뢰성이 낮음 : 데이터 전송의 순서를 보장하지 않으며 오류 검출 및 재전송 기능이 제한적.
6. 흐름 제어 부재 : 송신 속도를 제어하는 기능이 없어 네트워크 혼잡이 발생할 수 있다.

UDP의 장단점

장점:
1. 빠른 속도 : 실시간성이 요구되는 서비스(예: VoIP, 온라인 게임)에 적합하다.
2. 낮은 오버헤드 : 네트워크 자원을 효율적으로 사용할 수 있다.
3. 단순한 구현 : 프로토콜이 간단하여 구현이 쉽다.

단점:
1. 신뢰성이 낮음 : 데이터 손실이나 순서 혼란이 발생할 수 있다.
2. 흐름 제어 부재 : 네트워크 혼잡을 야기할 수 있다.
3. 오류 검출 기능이 제한적 : 데이터 손상을 감지하기 어렵다.

UDP의 사용

1. 실시간 스트리밍 : 실시간 음성/영상 스트리밍
2. 온라인 게임 : 낮은 지연 시간이 중요한 온라인 게임
3. DNS : 도메인 이름을 IP 주소로 변환하는 서비스
4. SNMP : 네트워크 장비를 관리하는 프로토콜

TCP와 UDP는 인터넷에서 데이터를 전송하는 데 사용되는 두 가지 주요 프로토콜이다. 각각 고유한 특징을 가지고 있으며 어떤 프로토콜을 사용할지는 애플리케이션의 요구 사항에 따라 달라진다.

TCP (Transmission Control Protocol)

1. 연결 지향형 : 데이터를 전송하기 전에 양쪽 끝단 간에 연결을 설정해야 한다.
2. 신뢰성 : 데이터가 손실되거나 순서가 바뀌지 않도록 오류 검출 및 재전송, 순서 제어 기능을 제공한다.
3. 흐름 제어 : 송신 속도를 수신 측의 처리 능력에 맞춰 조절하여 네트워크 혼잡을 방지한다.
4. 느린 속도 : 연결 설정 및 오류 처리 과정 때문에 UDP에 비해 상대적으로 느린 속도를 가지지만 신뢰성이 높다.
5. 대표적인 사용 사례 : 웹 브라우징(컴퓨터나 스마트폰 등의 기기를 통해 인터넷에 연결하여 웹 페이지를 보는 행위), 파일 전송, 이메일 등

UDP (User Datagram Protocol)

1. 비연결형 : 데이터를 전송하기 전에 연결을 설정할 필요가 없다.
2. 신뢰성이 낮음 : 데이터 손실이나 순서 혼란이 발생할 수 있으며 오류 검출 및 재전송 기능이 제한적이다.
3. 빠른 속도 : 연결 설정 과정이 없고 오버헤드가 적어 TCP보다 빠른 속도를 제공한다.
4. 흐름 제어 없음 : 송신 속도를 제어하지 않기 때문에 네트워크 혼잡이 발생할 수 있다.
5. 대표적인 사용 사례 : 실시간 스트리밍, DNS, SNMP 등

TCP와 UDP의 비교

특징	TCP	UDP
연결	연결 지향형	비연결형
신뢰성	높음	낮음
순서 보장	보장	보장하지 않음
흐름 제어	있음	없음
오버헤드	높음	낮음
속도	느림	빠름

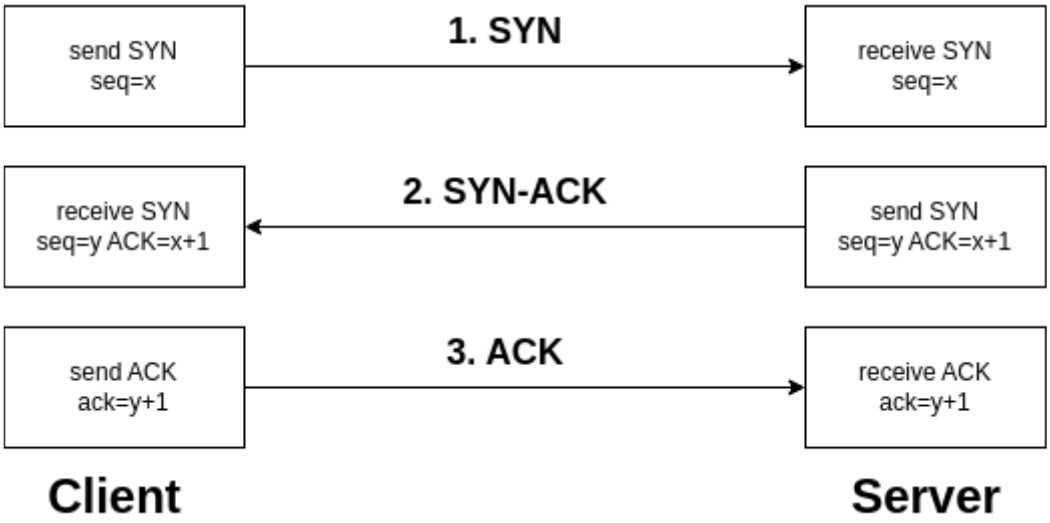
프로토콜을 선택

1. TCP : 데이터의 정확성과 신뢰성이 중요한 경우, 예를 들어 파일 전송, 이메일, 웹 브라우징 등에 사용된다.
2. UDP : 실시간성이 중요하고 데이터 손실이 허용되는 경우, 예를 들어 실시간 스트리밍, 온라인 게임, DNS 등에 사용된다.

결론적으로 TCP와 UDP는 각각 장단점이 있으며 어떤 프로토콜을 사용할지는 애플리케이션의 특성에 따라 적절하게 선택해야 한다.

TCP는 신뢰성 있는 데이터 전송을 보장하기 위해 사용되는 프로토콜이다. 이 프로토콜은 데이터를 주고받기 전에 3-way handshake 라는 과정을 거쳐 연결을 설정한다. 이 과정은 클라이언트와 서버 간에 연결이 안정적으로 이루어졌는지 확인하고 양쪽 모두 데이터 송수신 준비가 완료되었는지 확인하는 역할을 한다.

이미지 자료



출처 : https://www.flamingbytes.com/blog/tcp-3-way-handshake-syn-syn-ack-ack/

3-way handshake 과정

1. SYN 패킷 전송 :

   클라이언트가 서버에 연결 요청을 보낸다. 이때 SYN 플래그가 설정된 패킷을 전송하며 이는 "새로운 연결을 요청합니다"라는 의미이다.

   클라이언트는 자신이 사용할 포트 번호를 지정하고 서버가 사용할 포트 번호를 임의로 선택한다.
2. SYN-ACK 패킷 전송 :

   서버는 클라이언트의 연결 요청을 받고 SYN-ACK 플래그가 설정된 패킷을 클라이언트에게 보낸다.

   이 패킷은 "클라이언트의 연결 요청을 받았으며 나도 연결을 허용한다. 클라이언트가 보낸 포트 번호를 확인했고 내가 사용할 포트 번호는 이것입니다"라는 의미이다.
3. ACK 패킷 전송 :

   클라이언트는 서버의 SYN-ACK 패킷을 받고 ACK 플래그가 설정된 패킷을 서버에게 보낸다.

이 패킷은 "서버의 응답을 잘 받았습니다. 이제 연결이 완료되었으므로 데이터를 주고받을 준비가 되었습니다"라는 의미이다.

3-way handshake의 중요성

1. 신뢰성 있는 연결 확립 : 양쪽 모두 연결에 동의하고 데이터 송수신 준비가 되었다는 것을 확인한다.
2. 포트 번호 협상 : 클라이언트와 서버가 사용할 포트 번호를 결정한다.
3. 오류 감지 : 각 단계에서 패킷이 제대로 전달되지 않으면 연결이 실패하게 된다.

추가 설명

SYN : 연결을 동기화하기 위한 플래그이다.
ACK : 패킷을 정상적으로 받았다는 것을 확인하는 플래그이다.

마무리

TCP 3-way handshake는 인터넷에서 데이터를 안전하고 신뢰성 있게 주고받을 수 있도록 하는 필수적인 과정이다. 이 과정을 통해 클라이언트와 서버는 서로 연결을 설정하고 데이터를 송수신할 준비를 마친다.

OSI(Open Systems Interconnection) 7계층 모델은 컴퓨터 네트워크 통신을 7개의 계층으로 나누진다. 각 계층은 특정한 기능을 담당하며 데이터가 송신원에서 수신원으로 전달될 때 이 계층들을 거치면서 다양한 작업을 수행한다.

7개 계층의 역할

1. 물리 계층 (Physical Layer) :

   가장 하위 계층으로 실제 물리적인 매체(케이블, 광섬유 등)를 통해 비트(0 또는 1) 형태의 데이
   터를 전송하는 역할을 한다.

   전기 신호, 광 신호 등을 이용하여 데이터를 표현하고 물리적인 연결을 관리한다.
2. 데이터 링크 계층 (Data Link Layer) :

   물리 계층 위에 위치하며 데이터를 프레임으로 구성하고 오류 검출 및 재전송 흐름 제어 등을 담당한다.
   MAC 주소를 사용하여 네트워크 상의 장치들을 구별하고 데이터 전송의 신뢰성을 높인다.
3. 네트워크 계층 (Network Layer) :

   패킷이라는 단위로 데이터를 분할하고 목적지까지 전달하기 위한 경로를 설정하는 역할을 한다.
   IP주소를 사용하여 네트워크 상의 각 장치를 고유하게 식별한다.
4. 전송 계층 (Transport Layer) :

   데이터의 신뢰성 있는 전달을 보장하고 흐름 제어를 수행한다.

   TCP과 UDP(User Datagram Protocol)이 대표적인 전송 계층 프로토콜이다.
5. 세션 계층 (Session Layer) :

   두 개의 응용 프로그램 간의 통신 세션을 관리하고 연결의 설정, 유지, 해제를 담당한다.
6. 표현 계층 (Presentation Layer) :

   데이터의 형식을 변환하고 암호화, 압축 등을 수행하여 데이터를 표현하는 방식을 통일시킨다.
7. 응용 계층 (Application Layer) :

   사용자가 직접 사용하는 응용 프로그램(웹 브라우저, 이메일 클라이언트 등)이 위치하며 네트워크 서비스를 이용하여 데이터를 주고받는다.

각 계층의 사용 프로토콜 및 예시

1. 물리 계층 : 이더넷 케이블, 광섬유 케이블, Wi-Fi 무선 통신
2. 데이터 링크 계층 : Ethernet, PPP
3. 네트워크 계층 : IP, ICMP
4. 전송 계층 : TCP, UDP
5. 세션 계층 : RPC
6. 표현 계층 : JPEG, MPEG, ASCII
7. 응용 계층 : HTTP, FTP, SMTP

OSI 7계층 모델의 중요도

1. 네트워크의 이해 : OSI 모델은 복잡한 네트워크 시스템을 단순화하여 이해하는 데 도움을 준다.
2. 프로토콜 설계 : 각 계층별 기능을 명확하게 정의하여 새로운 네트워크 프로토콜을 설계하는 기준을 제공한다.
3. 문제 해결 : 네트워크 문제 발생 시, 문제가 발생한 계층을 파악하여 효율적으로 해결할 수 있다.

마무리

OSI 7계층 모델은 네트워크 통신을 이해하는 데 있어 기본적인 틀을 제공한다. 각 계층이 맡은 역할을 명확히 이해하면 네트워크 문제 발생 시 원인을 파악하고 해결하는 데 큰 도움이 된다. 하지만 OSI 모델은 이론적인 모델이며 실제 네트워크 환경에서는 TCP/IP 모델이 더 많이 사용된다고 한다.

OSI 7계층과 TCP/IP 4계층은 모두 네트워크 통신을 계층적으로 나누어 설명하는 모델이지만 몇 가지 중요한 차이점이 있다.

개발 시기와 목적

1. OSI 7계층 : 1970년대 국제 표준화 기구(ISO)에서 제안한 모델로 다양한 네트워크 환경에서의 통신을 표준화하고자 개발되었다. 이론적인 완성도가 높지만 실제 구현에는 어려움이 있어 모든 기능이 구현된 시스템은 드물다.
2. TCP/IP 4계층 : 1970년대 미국 국방성에서 개발된 모델로 인터넷 프로토콜을 중심으로 실제 네트워크 환경에서 사용하기 위해 설계다. OSI 모델보다 실제 구현이 쉽고 인터넷의 기반이 되는 프로토콜이다.

계층 구조

1. OSI 7계층 : 물리 계층, 데이터 링크 계층, 네트워크 계층, 전송 계층, 세션 계층, 표현 계층, 응용 계층으로 총 7개의 계층으로 구성되어 있다. 각 계층은 명확하게 정의된 기능을 수행하며 계층 간에 독립적으로 작동한다.
2. TCP/IP 4계층 : 네트워크 접근 계층, 인터넷 계층, 전송 계층, 응용 계층으로 총 4개의 계층으로 구성되어 있다. OSI 모델에 비해 계층이 간단하며 일부 기능은 여러 계층에 분산되어 있기도 하다.

사용

1. OSI 7계층 : 이론적인 모델로 네트워크의 개념을 이해하고 표준화하는 데 주로 사용된다. 실제 네트워크에서는 모든 계층이 구현되는 경우는 드물고 일부 계층만을 구현하여 사용하는 경우가 많다.
2. TCP/IP 4계층 : 인터넷의 기반이 되는 프로토콜로 전 세계적으로 가장 많이 사용되는 네트워크 모델이다. 실제 네트워크 장비 및 소프트웨어에서 널리 사용되며 다양한 응용 프로그램을 지원한다.

비교

항목	OSI 7계층	TCP/IP 4계층
개발 시기	1970년대	1970년대
목적	표준화, 이론적 완성도	실제 구현, 인터넷 기반
계층 수	7개	4개
실제 사용	이론적 모델	실제 네트워크

마무리

OSI 7계층과 TCP/IP 4계층은 서로 다른 목적으로 개발된 모델이지만, 네트워크 통신의 기본적인

개념을 이해하는 데 모두 중요하다. OSI 7계층은 네트워크의 전체적인 구조를 파악하고 각 계층의 역할을 이해하는 데 도움이 되며, TCP/IP 4계층은 실제 네트워크 환경에서 사용되는 프로토콜을 이해하는 데 도움이 된다.

IP 주소는 인터넷에 연결된 모든 장치(컴퓨터, 스마트폰 등)를 식별하기 위한 고유한 주소이다. 마치 집의 주소처럼 각 장치는 자신만의 고유한 IP 주소를 가지고 있어 다른 장치와 통신할 수 있다.

IP 주소는 크게 IPv4와 IPv6 두 가지 종류가 있다.

IPv4

32비트로 구성된 주소로 점(.)으로 구분된 4개의 숫자로 표현된다. (192.168.0.1) 현재 가장 많이 사용되는 주소 체계지만 고갈 문제로 새로운 IPv6 주소 체계가 등장했다.

IPv6

128비트로 구성된 주소로 콜론(:)으로 구분된 8개의 16진수 숫자로 표현된다. (2001:0db8:85a3:0000:0000:8a2e:0370:7334) IPv4보다 훨씬 더 많은 주소를 지원한다.

서브넷 마스크

서브넷 마스크는 IP 주소를 두 부분으로 나누는 역할을 한다.

1. 네트워크 주소 : 여러 장치가 공유하는 네트워크를 식별하는 부분이다.
2. 호스트 주소 : 네트워크 내에서 각 장치를 구별하는 부분이다.

서브넷 마스크는 IP 주소와 동일한 32비트 길이를 가지며 1과 0으로 구성된다. 1은 네트워크 부분을, 0은 호스트 부분을 나타낸다.

예시

IP 주소 : 192.168.0.100
서브넷 마스크 : 255.255.255.0

위 예시에서 서브넷 마스크를 2진수로 변환하면 11111111.11111111.11111111.00000000이 된다. 따라서 앞의 24비트(11111111.11111111.11111111)는 네트워크 부분, 뒤의 8비트(00000000)는 호스트 부분을 나타낸다. 즉, 192.168.0.100은 192.168.0.0 네트워크에 속하며 해당 네트워크 내에서 100번 주소를 가진 장치이다.

서브넷 마스크의 역할

1. 네트워크 분할 : 하나의 큰 네트워크를 여러 개의 작은 서브넷으로 나눌 수 있다.
2. 트래픽 관리 : 각 서브넷 내에서만 통신을 제한하여 네트워크 성능을 향상시킬 수 있다.
3. 보안 강화 : 서브넷을 이용하여 네트워크를 분리하여 보안을 강화할 수 있다.

서브넷 마스크가 필요성

1. 효율적인 네트워크 관리 : 많은 수의 장치를 효율적으로 관리하기 위해 네트워크를 분할해야 한다.

2. 보안 : 서브넷을 통해 특정 네트워크에 대한 접근을 제한하여 보안을 강화할 수 있다.
3. 주소 공간 활용 : 서브넷 마스크를 적절히 설정하면 주소 공간을 효율적으로 활용할 수 있다.

마무리

IP 주소와 서브넷 마스크는 네트워크에서 중요한 개념이다. 이 두 가지를 이해하면 네트워크의 작동 원리를 이해하고 네트워크 문제를 해결하는 데 도움이 된다.

HTTP(HyperText Transfer Protocol)는 인터넷에서 정보를 주고받는 가장 기본적인 통신 규약이다 웹 브라우저가 웹 서버에 특정 웹 페이지를 요청하고 서버가 그 요청에 맞는 데이터를 보내주는 과정에서 HTTP 프로토콜이 사용된다.

쉽게 말해 웹사이트를 방문할 때마다 브라우저와 서버 간에 HTTP 요청과 응답이 수없이 오가는 것이다.

HTTP의 특징

1. 텍스트 기반 : HTTP 메시지는 사람이 읽을 수 있는 텍스트 형식으로 구성된다.
2. 비연결성 : 각 요청과 응답은 독립적이며 지속적인 연결을 유지하지 않는다.
3. 무상태 : 서버는 이전 요청에 대한 정보를 기억하지 않는다.
4. 클라이언트-서버 구조 : 클라이언트(브라우저)가 서버에 요청하고 서버가 응답하는 구조이다.

HTTP 주요 메서드

1. GET : 특정 자원을 요청 (웹 페이지 조회)
2. POST : 서버에 새로운 데이터를 전송 (회원 가입, 게시글 작성)
3. PUT : 특정 자원을 업데이트
4. DELETE : 특정 자원을 삭제

HTTP 요청과 응답

1. 요청 : 클라이언트가 서버에 보내는 메시지로 요청하는 자원, 사용할 메서드 등의 정보를 포함한다.
2. 응답 : 서버가 클라이언트에게 보내는 메시지로, 요청에 대한 결과(성공/실패), 데이터 등을 포함한다.

HTTP의 역할

1. 웹 페이지 전송 : 우리가 웹사이트를 방문할 때마다 HTTP를 통해 HTML, CSS, JavaScript 등의 파일을 받아와 웹 페이지를 구성한다.
2. 데이터 전송 : 웹 페이지뿐만 아니라 이미지, 동영상, 파일 등 다양한 형태의 데이터를 전송하는 데 사용된다.
3. 웹 서비스 : RESTful API를 통해 서버와 클라이언트 간의 데이터 교환을 가능하게 한다.

HTTP 버전 별 정리

HTTP/1.0

웹 통신의 가장 초기 버전으로 웹의 발전에 중요한 역할을 했지만 현재는 거의 사용되지 않는다.

HTTP/1.0의 특징

1. 비지속 연결 : 한 번의 요청과 응답이 끝나면 TCP 연결이 끊어졌습니다. 매번 새로운 연결을 맺어야 했기 때문에 성능이 좋지 않다.
2. 단순한 요청/응답 : 하나의 요청에 대해 하나의 응답만 처리할 수 있었다.
3. 제한적인 메서드 : GET, HEAD, POST 등 기본적인 메서드만 지원했다.
4. 헤더 필드: HTTP/1.0에서는 요청과 응답에 다양한 헤더를 추가할 수 있게 되어 요청에 대한 추가 정보(예: 콘텐츠 유형, 서버 정보 등)를 전달할 수 있다.

HTTP/1.0의 한계

HTTP/1.0의 주요 한계는 비효율적인 연결 관리로 인해 대규모 트래픽을 처리하는데 적합하지 않다는 점이다. 이후 버전인 HTTP/1.1에서는 이를 개선하여 연결 재사용 및 더 많은 기능을 추가되었다.

HTTP/1.0과 HTTP/1.1의 비교

기능	HTTP/1.0	HTTP/1.1
연결	비지속 연결	지속 연결
요청	단일 요청	여러 요청 (파이프라이닝)
메서드	제한적	다양한 메서드 (PUT, DELETE 등)
캐싱	기본적	강화된 캐싱

HTTP/1.0은 웹 통신의 초기 단계에서 중요한 역할을 했지만 현재는 거의 사용되지 않다.
HTTP/1.1과 HTTP/2 등 더욱 발전된 프로토콜이 등장하면서 HTTP/1.0의 단점은 대부분 해결되었고 웹 성능과 기능이 크게 향상되었다.

HTTP/1.1

웹 통신의 기본 프로토콜인 HTTP의 주요 버전 중 하나이다. HTTP/1.0의 한계를 극복하고 웹 성능을 향상시키기 위해 많은 기능들이 추가되었다.

HTTP/1.1의 특징

1. 지속 연결 :
    1. 하나의 TCP 연결을 유지하여 여러 요청과 응답을 주고받을 수 있다.

    2. 매번 새로운 연결을 맺는 오버헤드를 줄여 응답 시간을 단축시킨다.
    3. `Connection: keep-alive` 헤더를 통해 지속 연결을 요청한다.
2. 파이프라이닝 :
    1. 지속 연결을 활용하여 여러 요청을 순서대로 보낼 수 있다.
    2. 서버는 요청을 받는 순서대로 응답하지 않을 수 있으므로 순서 보장 문제가 발생할 수 있다.
3. 새로운 메서드 :
    1. PUT, DELETE 등 새로운 메서드가 추가되었다.
4. 캐싱 :
    1. 캐싱 메커니즘이 강화되어 동일한 자원에 대한 반복적인 요청을 줄이고 응답 시간을 단축시킨다.
5. 오류 처리 :
    1. 정확하고 다양한 오류 코드를 제공하게 되었다.

HTTP/1.1의 한계

1. 헤더 오버헤드 :
    1. HTTP/1.1의 요청과 응답은 텍스트 기반의 헤더를 포함한다. 이 헤더들은 각 요청마다 전송되며, 종종 동일한 정보가 반복됩니다. 예를 들어, `User-Agent`, `Host`, `Accept-Language` 등의 헤더는 여러 요청에 반복적으로 포함되며, 이러한 반복적인 전송은 네트워크 자원을 낭비하게 만든다. 모바일 네트워크나 대역폭이 제한된 환경에서는 이 오버헤드가 성능 저하의 주요 원인이 될 수 있다.
2. HEAD 블로킹 :
    1. HTTP/1.1에서 클라이언트는 한 번에 하나의 요청만을 처리할 수 있다. 이로 인해 앞선 요청이 완료되지 않으면 뒤따르는 모든 요청이 대기 상태에 놓이게 된다. 이를 "헤드 오브 라인 블로킹(Head-of-Line Blocking)"이라고 부른다. 이미지, 스크립트 등 여러 리소스를 동시에 요청해야 하는 상황에서 첫 번째 요청이 지연되면 이후의 요청들도 연쇄적으로 지연되며 전체 페이지 로딩 시간이 길어진다.
3. TCP 오버헤드 :
    1. HTTP/1.1에서는 기본적으로 각 도메인에 대해 하나의 TCP 연결을 유지한다. 그러나 각 연결에는 설정과 종료 과정에서 TCP의 3-way handshake와 같은 추가적인 오버헤드가 발생한다. 또한 HTTP/1.1에서는 Keep-Alive 기능을 통해 TCP 연결을 재사용할 수 있지만 이 방식에서도 연결 유지와 관련된 비용이 존재한다. 다수의 요청이 있을 때 개별적인 TCP 연결을 관리하는 데 따른 자원 소모가 커지게 된다.
4. 파이프라이닝의 한계 :
   HTTP/1.1에서는 성능을 개선하기 위해 파이프라이닝이라는 기술을 도입했다. 이는 하나의 연결에서 여러 개의 요청을 순차적으로 보내고, 서버로부터 순서대로 응답을 받는 방식이다. 파이프라이닝은 몇 가지 심각한 한계를 가지고 있습니다
   1. 순서 보장 문제 : 파이프라이닝에서는 요청이 순서대로 처리되어야 한다. 만약 앞선 요청

의 응답이 지연되면 뒤따르는 모든 요청의 응답도 지연된다. 이로 인해 성능 저하가 발생할 수 있다.
2. 헤더 차단 문제 : 파이프라이닝 사용 시 앞선 요청의 헤더가 뒤따르는 요청의 처리를 막을 수 있는 문제가 발생한다. 특히 서버가 요청을 처리하는 데 시간이 오래 걸릴 경우 전체 파이프라인의 효율성이 저하된다.
3. 지원 부족 : 파이프라이닝은 이론적으로 성능을 향상시킬 수 있지만 실제로는 많은 서버와 클라이언트가 이를 완전히 지원하지 않거나 비활성화하고 있다. 이는 파이프라이닝의 사용을 제한하고 결국 기대했던 성능 향상이 이루어지지 않는 결과가 된다.

HTTP/1.1과 HTTP/2 비교

기능	HTTP/1.1	HTTP/2
프레임	텍스트 기반	바이너리 기반
헤더	압축 없음	HPACK 압축
멀티플렉싱	파이프라이닝 (제한적)	스트림 기반 멀티플렉싱
서버 푸시	지원하지 않음	지원

HTTP/1.1은 웹 통신의 기본을 다지고 웹 성능 향상에 크게 기여했다. 하지만 HTTP/2의 등장으로 인해 점차 그 중요성이 줄어들고 있다.

HTTP/2

HTTP/1.1의 한계를 극복하고 웹 페이지 로딩 속도를 비약적으로 향상시킨 차세대 HTTP 프로토콜이다. 기존의 텍스트 기반 프로토콜에서 벗어나 이진 형식으로 데이터를 전송하며 다양한 최적화 기술을 통해 웹 성능을 획기적으로 개선됐다.

HTTP/2의 특징

1. 바이너리 프레임 :
    1. HTTP/2는 텍스트 기반의 HTTP/1.1과 달리, 바이너리 프레이밍 계층을 사용한다. 이는 요청과 응답을 바이너리 포맷으로 변환하여 전송함으로써 데이터 처리와 파싱이 더 빠르고 효율적으로 이루어지도록 한다.
2. 헤더 압축 : \
    1. HTTP/2는 헤더 데이터를 효율적으로 전송하기 위해 HPACK이라는 헤더 압축 기법을 사용한다. HPACK은 요청과 응답에서 반복되는 헤더 필드를 효율적으로 압축하여 네트워크 대역폭 사용을 줄인다. 이로 인해 헤더 오버헤드가 크게 감소하며 특히 모바일 환경에서 성능 개선에 기여한다.

3. 멀티플렉싱 :
    1. HTTP/2의 가장 큰 특징 중 하나는 멀티플렉싱 기능이다. HTTP/1.1에서는 하나의 TCP 연결에서 한 번에 하나의 요청만 처리할 수 있었던 반면 HTTP/2는 하나의 연결에서 여러 요청과 응답을 동시에 주고받을 수 있다. 각각의 요청과 응답은 개별적인 스트림으로 처리되며 스트림 간의 차단 없이 병렬로 전송된다. 이는 "헤드 오브 라인 블로킹" 문제를 효과적으로 해결한다.
4. 서버 푸시 :
    1. HTTP/2는 서버 푸시 기능을 지원한다. 서버 푸시는 클라이언트가 요청하지 않은 리소스를 서버가 미리 전송할 수 있게 한다. 클라이언트가 HTML 문서를 요청하면 서버는 클라이언트가 곧 필요로 할 CSS 파일이나 JavaScript 파일을 함께 전송할 수 있다. 이는 페이지 로딩 속도를 크게 개선할 수 있다.
5. 스트림 우선순위 :
    1. HTTP/2는 각 스트림에 우선순위를 설정할 수 있는 기능을 제공한다. 클라이언트는 중요한 리소스(초기 렌더링에 필요한 CSS나 JavaScript 파일)에 높은 우선순위를 부여할 수 있다. 서버는 이 우선순위를 고려하여 리소스를 전송함으로써 사용자 경험을 최적화할 수 있다.
6. 연결 관리 개선 :
    1. HTTP/2는 단일 연결을 통해 모든 데이터를 주고받기 때문에 TCP 연결 수를 최소화할 수 있다. 이는 연결 설정과 종료에 따른 오버헤드를 줄이고 네트워크 자원의 효율적인 사용을 가능하게 한다. 특히 TLS(HTTPS)와 결합했을 때, 성능상의 이점이 더욱 두드러집니다.
7. 보안 :
    1. HTTP/2는 HTTPS와 함께 사용되는 경우가 많으며 TLS를 통해 데이터를 암호화한다. HTTP/2 자체는 보안을 필수로 요구하지 않지만 대부분의 브라우저는 HTTP/2를 HTTPS로만 사용하도록 강제하고 있다. 이는 웹 트래픽의 보안성을 높이는 데 기여한다.

HTTP/2의 한계

1. TCP의 한계 : HTTP/2는 기본적으로 TCP 위에 구축되어 있다. TCP는 연결 지향형 프로토콜로 연결 설정에 시간이 소요되고 네트워크 환경 변화에 민감하게 반응한다. 모바일 환경에서 발생하는 네트워크 끊김이나 혼잡은 HTTP/2의 성능 저하가 될 수 있다.
2. 헤드 오브 라인 블로킹 : 하나의 TCP 연결에서 여러 요청이 순차적으로 처리되기 때문에 앞쪽 요청이 지연되면 뒤쪽 요청도 함께 지연되는 현상이 발생한다 이는 특히 많은 자원을 사용하는 웹 페이지에서 성능 저하를 초래할 수 있다.
3. 연결 설정 오버헤드 : TCP 연결을 설정하는 데 필요한 시간은 무시할 수 없으며 특히 많은 수의 작은 요청을 처리하는 경우 성능에 영향을 미칠 수 있다.

HTTP/2와 HTTP/3의 비교

특징	HTTP/2	HTTP/3
기반 프로토콜	TCP	QUIC (UDP 기반)
헤더 압축	HPACK	HPACK 유사
멀티플렉싱	하나의 TCP 연결 내에서 여러 스트림	하나의 QUIC 연결 내에서 여러 스트림
흐름 제어	흐름 제어 프레임	흐름 제어 프레임
연결 설정	TCP 핸드셰이크	TLS 핸드셰이크 (0-RTT 가능)
오류 복구	TCP 재전송	QUIC의 자체적인 오류 복구 메커니즘
보안	TLS	TLS (기본적으로 암호화)

HTTP/2는 웹 성능을 획기적으로 향상시킨 프로토콜로 웹 개발에서 필수적인 기술이다. HTTP/2를 도입하면 웹 사이트의 로딩 속도를 빠르게 개선하고 사용자 경험을 향상시킬 수 있다.

HTTP/3

HTTP 프로토콜의 최신 버전으로 HTTP/2의 한계를 극복하기 위해 개발되었다. 2018년에 처음 도입되었으며 HTTP/3는 기존의 TCP 대신 QUIC(Quick UDP Internet Connections)라는 새로운 전송 프로토콜을 기반으로 하여 성능과 안정성을 크게 향상시켰다. 특히 모바일 환경에서 발생하는 네트워크 변동에 강하고 더 낮은 지연 시간을 제공하여 사용자 경험을 향상 시켰다.

HTTP/3의 특징

1. QUIC 프로토콜 기반 :
    1. HTTP/3의 가장 큰 특징은 전송 계층에서 TCP 대신 QUIC을 사용하는 것이다. QUIC은 UDP(사용자 데이터그램 프로토콜) 위에서 작동하며 TCP의 기능을 제공하면서도 여러 가지 장점을 추가 되었다.
        1. 빠른 연결 설정 : QUIC은 TCP와 달리 핸드셰이크 과정을 최소화하여 더 빠르게 연결을 설정할 수 있다. 특히 TLS 암호화와 연결 설정을 동시에 처리하여 지연 시간을 줄인다.
        2. 헤드 오브 라인 블로킹 문제 해결 : TCP는 패킷이 손실되면 그 패킷이 복구될 때까지 모든 후속 패킷을 기다려야 한다. 이로 인해 성능 저하가 발생하는데 이를 "헤드 오브 라인 블로킹"이라고 한다. QUIC은 각 스트림이 독립적으로 관리되기 때문에 특정 스트림에서 패킷 손실이 발생하더라도 다른 스트림의 데이터 전송에 영향을 미치지 않는다.
2. 멀티플렉싱과 스트림 관리 :

    1. HTTP/3는 HTTP/2와 마찬가지로 멀티플렉싱을 지원하지만 스트림 관리가 더 효율적이다. QUIC은 각 스트림을 독립적으로 처리하여 데이터가 비동기적으로 전송될 수 있다. 이는 HTTP/2의 멀티플렉싱에서 발생할 수 있는 헤드 오브 라인 블로킹 문제를 해결하는 데 크게 기여한다.
3. 보안과 암호화 :
    1. HTTP/3는 QUIC 위에서 작동하며 QUIC은 기본적으로 TLS 1.3을 사용하여 모든 데이터를 암호화한다. 이 암호화는 기본적으로 활성화되며 QUIC 연결을 통해 주고받는 모든 데이터는 안전하게 보호된다. QUIC의 설계 자체에 보안이 포함되어 있기 때문에 HTTP/3는 추가적인 보안 계층 없이도 높은 수준의 보안을 제공한다.
4. 패킷 손실에 대한 더 나은 처리 :
    1. QUIC은 패킷 손실에 민감하지 않다. 전송된 데이터 중 일부 패킷이 손실되더라도 나머지 패킷은 정상적으로 처리될 수 있다. QUIC은 손실된 패킷만을 재전송하고 나머지 데이터 스트림은 중단 없이 계속 처리되기 때문에 연결 성능이 향상된다.

HTTP/3의 한계

1. 네트워크 환경과의 호환성 : QUIC은 UDP 기반이기 때문에 일부 방화벽이나 네트워크 장비가 UDP 트래픽을 제한하거나 차단할 수 있다.
2. 복잡한 구현 : QUIC과 HTTP/3의 복잡한 구조는 서버와 클라이언트 모두에서 구현과 관리에 많은 노력이 필요한다.
3. 구형 장치와의 호환성 : HTTP/3는 최신 기술이기 때문에 오래된 장치나 소프트웨어에서는 지원되지 않을 수 있다.

HTTP/3는 웹의 요구에 부응하는 프로토콜로 특히 대규모 웹 애플리케이션과 실시간 통신이 필요한 서비스에서 좋은 성능을 발휘한다. QUIC의 도입으로 인해 성능, 보안, 효율성 면에서 많은 개선이 이루어졌으며 HTTP/2의 한계를 효과적으로 극복하고 있다.

HTTPS (HyperText Transfer Protocol Secure)

웹에서 데이터를 안전하게 전송하기 위해 사용되는 프로토콜이다. HTTP의 보안 버전으로 주로 웹사이트와 서버 간의 데이터를 암호화하여 제3자가 이를 읽거나 변조하는 것을 방지한디. HTTPS는 특히 인터넷에서 개인정보를 보호하고 안전한 통신을 보장하는 데 중요한 역할을 한다.

HTTPS의 구성 및 특징

1. TLS (Transport Layer Security) / SSL (Secure Sockets Layer) :

    1. TLS : HTTPS는 TLS프로토콜을 사용하여 데이터를 암호화한다. TLS는 SSL의 후속 버전으로 보안성과 성능이 개선된 프로토콜이다. HTTPS는 서버와 클라이언트 간의 통신을 암호화하고 데이터의 무결성을 보장하기 위해 TLS를 사용된다.
    2. SSL : TLS의 초기 버전이며 현재는 TLS가 대부분의 보안 통신에서 사용되고 있다. SSL의 최신 버전은 SSL 3.0이지만 웹에서는 TLS 1.2와 TLS 1.3이 주로 사용된다.
2. 암호화 :
    1. 대칭 암호화 : 데이터 전송 중 실시간으로 데이터를 암호화하는 방식이다. 암호화와 복호화에 같은 키를 사용된다.
    2. 비대칭 암호화 : 공개 키와 개인 키의 쌍을 사용하여 데이터를 암호화한다. 공개 키로 암호화된 데이터는 개인 키로만 복호화할 수 있다.
3. 인증서 :
    1. HTTPS는 디지털 인증서를 사용하여 서버의 신원을 확인한다. 인증서는 인증 기관에서 발급되며 웹사이트의 도메인과 기업 정보를 포함한다. 클라이언트는 인증서를 통해 서버가 신뢰할 수 있는지 확인할 수 있다.
    2. SSL/TLS 인증서 : 인증서는 공개 키 암호화를 지원하며 웹사이트의 신원을 검증한다. 인증서에는 발급자의 정보, 인증서 소유자의 정보, 유효 기간 등이 포함된다.
4. 핸드셰이크 과정 :
    1. TLS 핸드셰이크 : 클라이언트와 서버 간의 보안 연결을 설정하기 위한 과정이다. 이 과정에서 암호화 방식과 세션 키가 협상되며 서버는 인증서를 클라이언트에게 전송된다. 클라이언트는 인증서를 검증하고 서버와의 안전한 통신을 위한 세션 키를 생성한다.

HTTPS의 작동 방식

1. 클라이언트와 서버 연결 : 클라이언트(웹 브라우저)는 서버에 HTTPS 요청을 보낸다. 서버는 SSL/TLS 인증서를 클라이언트에 제공하여 자신의 신원을 증명한다.
2. TLS 핸드셰이크 : 클라이언트와 서버 간의 핸드셰이크 과정이 시작된다. 이 과정에서 암호화 방식과 세션 키를 협상하며 클라이언트는 서버의 인증서를 검토하여 신뢰성을 확인한다.
3. 암호화된 통신 : 핸드셰이크가 완료되면 클라이언트와 서버 간의 데이터 전송은 암호화된다. 이를 통해 데이터가 중간에서 가로채이거나 변조되는 것을 방지할 수 있다.
4. 세션 종료 : 통신이 종료되면 세션 키와 같은 암호화 정보는 폐기된다. 이후의 통신은 새로운 핸드셰이크 과정으로 시작된다.

HTTPS의 사용 이유

1. 데이터 암호화 : HTTPS는 데이터 전송 시 암호화를 통해 개인 정보와 중요한 데이터를 보호한다. 이로 인해 데이터가 중간에서 도청되거나 변경되는 것을 방지할 수 있다.
2. 서버 인증 : 디지털 인증서를 사용하여 서버의 신뢰성을 검증함으로써 사용자에게 안전한 웹사이트 접속을 보장된다.
3. 데이터 무결성 : 데이터 전송 중에 내용이 변경되지 않았음을 확인할 수 있다. HTTPS는 데이터가 전송되는 동안 손상되거나 변조되는 것을 방지한다.

4. 사용자 신뢰 : 구글과 같은 검색 엔진은 HTTPS를 사용하는 웹사이트에 대해 더 높은 순위를 부여한다. 웹사이트 방문자는 HTTPS를 통해 보안이 강화된 사이트에 더 많은 신뢰를 가진다.

HTTPS와 HTTP의 차이

1. 보안 : HTTP는 데이터를 암호화하지 않지만 HTTPS는 TLS를 사용하여 데이터를 암호화하고 보호한다.
2. 포트 : HTTP는 기본적으로 포트 80을 사용하며 HTTPS는 포트 443을 사용한다.
3. URL 표시 : HTTPS 웹사이트의 URL은 `https://` 로 시작하며 HTTP는 `http://` 로 시작한다. HTTPS 웹사이트는 종종 브라우저 주소창에 자물쇠 아이콘을 표시하여 보안 상태를 나타낸다.

웹 기술의 발전에 따라 HTTPS는 계속해서 발전하고 있으며 HTTP/3와 같은 최신 프로토콜은 더욱 향상된 성능과 보안을 제공한다. 앞으로도 HTTPS는 웹의 신뢰성과 보안을 유지하는 데 필수적인 역할을 할 것이다.

추가 정리

HTTP/3는 기본적으로 QUIC 프로토콜 위에서 작동하며 QUIC은 모든 데이터를 전송할 때 TLS 1.3을 사용하여 데이터를 암호화한다. 이로 인해 HTTP/3 통신은 자동으로 암호화되어 안전한 전송이 보장된다.
그러나 HTTP/3가 TLS 1.3을 사용한다는 사실이 HTTPS가 필요 없다는 것을 의미하지는 않는다. 오히려 HTTP/3는 일반적으로 HTTPS를 통해 사용된다고 한다.

이유

1. 보안 표준 : HTTPS는 웹에서 보안 통신을 위한 표준이다. HTTP/3가 TLS 1.3을 사용하더라도 그 통신이 "HTTPS"로 명시되어야 웹 브라우저와 서버가 이를 안전한 연결로 인식하고 처리한다. HTTP/3도 여전히 "https://" 프로토콜을 사용하며 이 표기 방식이 웹에서의 보안 통신을 의미한다.
2. 브라우저 및 서버 호환성 : 대부분의 웹 브라우저와 서버는 HTTP/3를 지원할 때 이를 HTTPS와 함께 사용하도록 설계되어 있다. 웹 서버와 브라우저는 "https://"를 통해 접속 요청이 들어오면 HTTP/3 프로토콜을 사용해 연결을 설정한다.
3. 사용자 경험 : 사용자는 HTTPS를 통해 웹사이트에 접속할 때 URL에 "https://"를 기대한다. 이는 사용자가 연결이 안전하다는 확신을 가지는 데 중요한 역할을 한다. HTTP/3가 암호화를 기본으로 한다고 해도 HTTPS는 여전히 보안의 상징이며 웹 사용자의 신뢰를 유지하는 데 필요한다.

요약

HTTP/3는 TLS 1.3을 통해 모든 데이터를 암호화하지만 여전히 HTTPS가 필요합니다. HTTP/3를 사용할 때 HTTPS는 안전한 웹 통신의 상징으로 계속 사용되며 브라우저와 서버 모두에서 이를 통해 보안 통신이 이루어진다 따라서 HTTP/3 환경에서도 HTTPS는 여전히 필수적이다.

SSL/TLS 인증서는 웹사이트와 사용자의 브라우저 간의 안전한 통신을 보장하기 위해 사용되는 디지털 서명이다. SSL(Secure Sockets Layer)과 TLS(Transport Layer Security)은 웹 보안 프로토콜로 데이터 전송 과정에서 기밀성(정보를 오직 인가된 사람들에게 만 공개하는 것)을 유지하고 데이터가 변조되지 않도록 한다. 이 인증서는 웹사이트가 신뢰할 수 있는지 확인하고 클라이언트와 서버 간의 통신을 암호화한다.

SSL/TLS 인증서의 기능

1. 인증(Authentication) : 인증서는 웹사이트의 신원을 확인한다. 이를 통해 사용자는 자신이 접속한 사이트가 진짜 사이트인지 확인할 수 있다.
2. 암호화(Encryption) : 인증서는 웹사이트와 사용자가 주고받는 데이터를 암호화한다. 이를 통해 네트워크 상에서 데이터를 가로채더라도 내용을 읽을 수 없게 만들어 해킹이나 도청으로부터 안전하게 보호한다.

추가적인 인증서가 필요한 이유

1. 사용자 신뢰도 향상 : HTTPS로 시작되는 웹사이트는 사용자에게 안전하고 신뢰할 수 있다는 인상을 준다.
2. 검색 엔진 순위 상승 : 구글과 같은 검색 엔진은 HTTPS 웹사이트를 더욱 신뢰하고 검색 결과 상위에 노출시키는 경향이 있다.
3. 법적 요구 사항 : 일부 국가에서는 특정 산업 분야에서 SSL/TLS 인증서를 의무화하고 있다.

SSL/TLS 인증서의 종류

1. 도메인 검증 인증서(Domain Validation, DV) : 가장 기본적인 인증서로 도메인 소유권만 확인한다. 발급 절차가 간단하고 비용이 저렴하다.
2. 조직 검증 인증서(Organization Validation, OV) : 도메인 소유권 외에 기업의 법인등기부 등을 통해 조직의 신원을 확인한다. 회사나 기관의 법적 존재를 확인한 후 발급된다.
3. 확장 검증 인증서(Extended Validation, EV) : 가장 높은 수준의 인증서로 조직의 신원을 철저하게 검증하고 브라우저 주소창에 회사명이 녹색으로 표시된다. 금융기관 등 높은 수준의 보안을 요구하는 사이트에서 주로 사용된다.

SSL/TLS 인증서의 실행 과정

1. 핸드셰이크 과정 사용자가 웹사이트에 접속 :사용자가 웹사이트에 접속하면 브라우저가 웹 서버에 SSL/TLS 연결을 요청한다. (클라이언트와 서버는 서로의 인증서를 교환, 웹 서버는 사용할 암호화 방식 및 키 제공)
2. 데이터 암호화 : 브라우저는 핸드셰이크 과정 받은 공개 키를 사용하여 데이터를 암호화하고 웹 서버로 전송한다.
3. 데이터 복호화 : 웹 서버는 자신의 개인 키를 사용하여 받은 데이터를 복호화한다.

4. 세션 종료 : 통신이 끝나면 SSL/TLS 세션이 종료되고 사용된 키는 더 이상 유효하지 않게 된다. 새로운 통신이 시작될 때마다 새로운 핸드셰이크와 키 교환이 이루어진다.

마무리

SSL은 SSL 2.0과 SSL 3.0은 심각한 보안 취약점이 발견되어 현재는 더 이상 사용되지 않고 TLS가 표준으로 자리 잡았다. TLS 1.3은 현재 최신 버전으로 더욱 향상된 보안 기능과 빠른 성능을 제공한다.

도메인은 인터넷 상에서 특정 웹사이트나 서버를 식별하는 고유한 주소이다. 우리가 익숙하게 사용하는 www.naver.com 과 같은 형태로 숫자로 이루어진 복잡한 IP 주소를 사람이 쉽게 기억하고 입력할 수 있도록 만든 일종의 별칭이라고 생각하면 된다.

도메인의 구성

1. 최상위 도메인(Top-Level Domain, TLD) : 도메인 이름의 마지막 부분으로, "`.com`", "`.org`", "`.net`", "`.kr`" 등과 같이 도메인의 가장 오른쪽에 위치한 부분이다. TLD는 일반적으로 도메인의 목적이나 국가를 나타난다.
    1. 일반 최상위 도메인 : ".com", ".org", ".net" 등과 같이 특정 목적이나 조직 유형에 관계없이 전 세계에서 사용 가능한 도메인이다.
    2. 국가 코드 최상위 도메인 : ".kr", ".jp", ".uk" 등과 같이 특정 국가나 지역을 나타내는 도메인이다.
    3. 기관 코드 최상위 도메인 : 특정 기관이나 단체를 나타내는 도메인으로 .edu(교육기관), .gov(정부기관) 등이 있다.
2. 2차 도메인(Second-Level Domain, SLD) : 최상위 도메인 바로 앞에 위치하며 개인이나 기업의 이름 또는 서비스 종류 등을 나타낸다. naver, google 등이 이에 해당한다.
3. 서브 도메인 : 제2차 도메인 앞에 추가로 붙일 수 있는 부분으로 웹사이트 내의 특정 서비스나 콘텐츠를 구분하기 위해 사용된다. 예시 mail.google.com에서 mail이 서브 도메인

도메인 등록

도메인을 사용하려면 도메인 등록 기관(레지스트라)에 등록해야 한다. 가비아, 호스팅 업체 등 다양한 곳에서 도메인을 등록할 수 있다. 도메인 등록 시에는 원하는 도메인 이름을 선택하고 등록 기간과 함께 비용을 지불해야 한다.

도메인 관리

1. 도메인 갱신 : 등록 기간이 만료되기 전에 갱신해야 계속 사용할 수 있다.
2. DNS 설정 : 도메인과 웹 서버를 연결하는 작업으로 도메인을 통해 웹사이트에 접속할 수 있도록 설정해야 한다.
3. Whois 정보 : 도메인 소유주 정보를 조회할 수 있는 서비스이다.
4. 도메인 이전 : 다른 도메인 등록 기관으로 도메인을 이전할 수 있다.

마무리

도메인은 인터넷에서 특정 리소스에 접근할 수 있도록 하는 중요한 주소 체계이다. 웹사이트의 신뢰성을 유지하기 위해 도메인 관리에 신경을 쓰는 것이 중요하다.

DNS(Domain Name System)는 인터넷에서 도메인 이름(www.naver.com)을 IP 주소로 변환해주는 시스템이다. 인터넷의 "전화번호부"라고 생각하면 이해하기 쉽다.

DNS의 역할

1. 이름 해석 : 사용자가 입력한 도메인 이름을 IP 주소로 변환하여 해당 웹사이트에 접속할 수 있게 해준다.
2. 로드 밸런싱 : 하나의 도메인 이름에 여러 IP 주소를 연결해, 웹사이트 트래픽을 분산시켜 성능을 향상시킨다.
3. 도메인 관리 : 도메인 이름과 관련된 정보를 관리한다.

DNS의 구조와 계층

DNS는 계층적인 구조로 구성되어 있으며 크게 4개의 주요 계층이다

1. 루트 네임서버 : DNS 계층의 최상위에 위치한 서버로 모든 도메인 요청의 출발점이다. 루트 서버는 최상위 도메인(TLD) 서버의 위치를 알려준다.
2. 최상위 도메인(TLD) 네임서버 : 루트 네임서버 아래에 위치하며 각 TLD(.com, .net, .org, .kr 등)에 대한 정보를 제공한다. TLD 서버는 해당 도메인의 권한 있는 네임서버의 위치를 반환한다.
3. 권한 있는 네임서버 : 특정 도메인에 대한 최종 정보를 가지고 있는 서버이다. "naver.com"의 IP 주소를 실제로 반환하는 서버가 권한 있는 네임서버이다.
4. 재귀적 네임서버 : 사용자가 도메인 이름을 요청할 때 루트 서버부터 권한 있는 네임서버까지의 모든 단계에서 필요한 정보를 대신 요청하고 결과를 사용자에게 반환하는 역할을 한다.

DNS 작동 원리

1. 사용자가 도메인 이름을 입력 : 사용자가 브라우저에 "www.naver.com" 을 입력한다.
2. 재귀적 네임서버 요청 : 사용자의 로컬 재귀적 네임서버가 이 요청을 처리한다. 이 서버는 루트 네임서버로부터 시작하여 최상위 도메인 네임서버, 권한 있는 네임서버로 요청을 이어가며 필요한 정보를 얻는다.
3. DNS 응답 : 권한 있는 네임서버가 해당 도메인의 IP 주소를 반환하면 재귀적 네임서버는 그 정보를 캐시(Cache)에 저장하고 사용자에게 전달한다.
4. 사용자에게 IP 주소 전달 : 사용자의 브라우저는 반환된 IP 주소를 사용하여 해당 서버에 연결하고 웹사이트를 로드한다.

DNS 캐싱

DNS는 조회된 IP 주소를 캐시(일시 저장)하여 동일한 요청이 있을 때 빠르게 응답한다.

마무리

DNS는 우리가 기억하기 쉬운 도메인 이름을 입력할 때 그것을 컴퓨터가 이해할 수 있는 IP 주소로 바꿔주는 중요한 시스템이다.

RESTful API는 웹 서비스 개발에서 자주 사용되는 아키텍처 스타일로 클라이언트와 서버 간의 통신을 간단하고 효율적으로 만드는 방식이다. REST(Representational State Transfer)는 웹의 기본 원칙에 기반하며 HTTP 프로토콜을 활용해 클라이언트가 서버의 자원에 접근하도록 설계되었다.

REST의 기본 원칙

1. 클라이언트-서버 아키텍처 : 클라이언트와 서버는 서로 독립적으로 동작하며, 클라이언트는 사용자 인터페이스를 관리하고 서버는 데이터 저장과 처리 로직을 관리한다.
2. 무상태성 : 서버는 각 요청을 독립적으로 처리하며 이전 요청에 대한 정보를 저장하지 않는다. 모든 요청에는 필요한 모든 정보가 포함되어야 한다.
3. 캐시 처리 가능 : 응답을 캐싱할 수 있도록 메시지에 캐싱 가능 여부를 명시한다. 캐싱을 통해 네트워크 트래픽을 줄이고 응답 속도를 향상시킬 수 있다.
4. 계층화된 시스템 : 클라이언트는 서버와 직접 통신할 수도 있고 중간 계층(로드 밸런서, 프록시)을 거쳐 통신할 수도 있다.
5. 인터페이스의 일관성 : RESTful 시스템에서는 모든 리소스에 대해 일관된 방식으로 접근할 수 있어야 한다. 이 원칙은 URL 구조, HTTP 메서드, 상태 코드 등이 표준화되어야 함을 의미한다.
6. 요청의 표현 : 클라이언트는 리소스 자체가 아닌 그 표현을 요청한다. 서버의 데이터베이스에 저장된 리소스는 JSON, XML 등 다양한 형식으로 표현될 수 있다.

RESTful API의 구성 요소

RESTful API는 주로 HTTP 프로토콜과 함께 사용된다.

리소스(Resource) : 모든 것(데이터, 객체 등)을 리소스로 간주하며 각 리소스는 고유한 URL(엔드포인트 : 웹 애플리케이션에서 클라이언트가 서버에 요청을 보내는 특정 URL 경로)로 식별된다.
예시 : `https://api.sample.com/users/123` (사용자 ID가 123인 사용자에 대한 리소스)

1. HTTP 메서드 : 리소스에 대해 수행할 작업을 정의하는 데 사용된다.
    1. GET : 리소스를 조회
    2. POST : 새로운 리소스를 생성
    3. PUT : 리소스를 업데이트
    4. DELETE : 리소스를 삭제
    5. PATCH : 리소스의 일부를 업데이트
    6. 그 외 메서드 : HEAD (GET 메서드와 유사하지만 응답 본문 대신 헤더 정보만 받는다), OPTIONS (서버가 지원하는 HTTP 메서드를 확인), TRACE (요청이 서버를 거치면서 어떤 변화를 겪는지 추적, 디버깅 목적), CONNECT (터널링을 설정하여 다른 프로토콜을 통해 통신)

2. HTTP 상태 코드 : 서버가 클라이언트의 요청에 대해 어떤 결과를 반환하는지 나타낸다.
    1. 200 OK : 요청이 성공
    2. 201 Created : 새로운 리소스가 성공적으로 생성
    3. 400 Bad Request : 잘못된 요청
    4. 401 Unauthorized : 인증이 필요
    5. 404 Not Found : 요청한 리소스를 찾을 수 없음
    6. 500 Internal Server Error : 서버에서 문제가 발생
3. URL(Uniform Resource Locator) : RESTful API에서는 각 리소스가 고유한 URL로 식별됩니다. URL은 명사로 표현되며 리소스에 대한 위치를 나타낸다.
    1. 예: `https://api.sample.com/users/` (모든 유저 목록을 가져옴)
4. 헤더(Headers) : 클라이언트와 서버 간의 추가 정보를 전송할 때 사용된다. 콘텐츠 타입(Content-Type)을 지정하여 요청이나 응답 데이터의 형식을 나타낼 수 있다.
    1. Content-Type : `application/json` , `application/xml` 등
5. 페이로드(Payload) : REST API에서 POST, PUT, PATCH와 같은 메서드를 사용할 때 클라이언트(보통 웹 브라우저나 앱)에서 서버로 데이터를 보내야 하는 경우가 많다. 이때 보내는 데이터 자체를 페이로드라고 한다.

RESTful API의 설계 원칙

RESTful API는 일관되고 사용하기 쉬운 설계가 중요하다.

1. 명확한 URI(Uniform Resource Identifier): 각 리소스는 명확한 URI로 식별된다. URI는 간결하고 이해하기 쉽게 설계해야 한다.
   예시 : `/customers/456/invoices` (고객 ID가 456인 사용자의 청구서 목록)
2. 일관된 HTTP 메서드 사용 : GET, POST, PUT, DELETE 등의 HTTP 메서드를 일관되게 사용하여 API의 사용성을 높인다.
   예시: `GET /orders/789` (주문 ID가 789인 주문의 세부 정보 조회)
3. 응답 데이터의 구조 : API의 응답 데이터는 클라이언트가 쉽게 이해하고 처리할 수 있는 구조여야 한다. 일반적으로 JSON 형식을 사용한다.
   예시 : `{"id": 789, "status": "shipped", "items": [{"product_id": 321, "quantity": 2}]}`
4. 필터링, 정렬, 페이징 : 대규모 데이터셋에서 유용하게 사용된다. 특정 조건에 맞는 데이터만 가져오거나 데이터를 페이지별로 나누어 가져올 수 있도록 설계한다.
   예시 : `/employees?department=sales&sort=name&limit=20&page=3` (영업 부서 직원들을 이름순으로 정렬한 세 번째 페이지에서 20명 가져오기)
5. 버전 관리 : API의 버전 관리는 중요한 기능이다. 새로운 기능 추가나 변경 사항이 기존 클라이언트에 영향을 미치지 않도록 URL에 버전을 포함하는 방식이 일반적이다.
   예시 : `/api/v2/products/654` (API 버전 2에서 제품 ID가 654인 제품의 정보 조회)

마무리

RESTful API는 현대 웹 개발에서 필수적인 요소이다. 올바른 설계와 구현은 애플리케이션의 확

장성(사용량에 대해 어떻게 대응할 수 있는지를 나타내는 속성)과 유지보수성(지하고 관리하는 데 얼마나 용이한지를 나타내는 속성)을 크게 향상시킬 수 있다.

WebSocket은 클라이언트와 서버 간의 실시간 양방향 통신을 가능하게 하는 프로토콜이다. 기존의 HTTP 프로토콜이 클라이언트가 요청을 보내면 서버가 응답하는 일방적인 방식이었다면 WebSocket은 클라이언트와 서버가 서로 메시지를 주고받으며 실시간으로 데이터를 교환할 수 있다.

특징

1. 양방향 통신 : 클라이언트와 서버가 동시에 메시지를 주고받을 수 있다.
2. 지속적인 연결 : HTTP처럼 요청-응답 방식이 아니라 연결이 한 번 성립되면 계속 유지된다.
3. 낮은 오버헤드 : HTTP 헤더가 필요 없어 데이터 전송량이 적고 효율적이다.
4. 실시간 데이터 전송 : 챗봇, 채팅, 온라인 게임 등 실시간 연결이 요구되는 서비스에 적합하다.

WebSocket의 동작 원리

1. 핸드셰이크 : WebSocket 통신은 먼저 HTTP를 통해 "핸드셰이크"를 수행한다. 클라이언트는 `ws://` 또는 `wss://` (보안 연결)로 시작하는 URL로 서버에 연결을 요청한다. 서버는 특정 HTTP 헤더를 사용해 이 요청을 WebSocket 프로토콜로 업그레이드한다. 핸드셰이크가 성공하면 HTTP 연결이 WebSocket 연결로 전환된다.
   예시 :

```
GET /chat HTTP/1.1
Host: example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Sec-WebSocket-Version: 13
```

핸드셰이크 요청 : 클라이언트는 `GET` 요청을 사용하여 WebSocket 연결을 시작한다.
1. `Upgrade: websocket` : 이 헤더는 서버에게 이 HTTP 연결을 WebSocket으로 업그레이드하려고 함을 알린다.
2. `Connection: Upgrade` : HTTP/1.1 연결에서 업그레이드 요청을 수행하기 위해 필요하다.
3. `Sec-WebSocket-Key` : 클라이언트가 제공하는 임의의 16바이트 키로 서버는 이를 바탕으로 응답 키를 생성한다.
4. `Sec-WebSocket-Version` : 클라이언트가 지원하는 WebSocket 프로토콜의 버전을 명시 일반적으로 13(버전 13이 WebSocket 프로토콜의 표준으로 채택된 최신 안정화 버전)이 전 세계의 주요 브라우저와 서버에서 지원되는 표준이기 때문에 WebSocket 통신의 기본 버전으로 사용된다.
5. `HTTP/1.1` 사용 이유 :
1. `Upgrade` 헤더 : WebSocket은 기존의 HTTP 연결을 WebSocket으로 전환하기 위해 HTTP/1.1의 `Upgrade` 헤더를 사용하다. HTTP/1.0에는 이 기능이 없다.

2. 지속적인 연결 : WebSocket은 클라이언트와 서버 간의 지속적인 양방향 통신을 필요하다. HTTP/1.1은 지속적인 연결을 기본적으로 지원한다.
3. `101 Switching Protocols` : HTTP/1.1은 WebSocket으로 전환되었음을 나타내는 `101 Switching Protocols` 상태 코드를 지원한다.
4. HTTP/2 : HTTP/2는 기본적으로 WebSocket을 지원하지 않으며 HTTP/1.1로 다운그레이드하여 WebSocket을 사용해야 한다.
5. HTTP/3 : HTTP/3는 QUIC 기반으로 WebSocket과의 통합이 복잡하고 현재 표준화되지 않는다.

서버 응답 예시 :

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
```

서버의 응답 : 서버가 요청을 수락하면 HTTP 상태 코드 `101 Switching Protocols` 와 함께 아래 내용을 보낸다.
1. `Upgrade: websocket` : 서버가 WebSocket 프로토콜로 업그레이드를 승인했음을 나타낸다.
2. `Connection: Upgrade` : 업그레이드가 성공적으로 수행되었음을 나타낸다.
3. `Sec-WebSocket-Accept` : 서버는 클라이언트가 보낸 `Sec-WebSocket-Key` 에 `258EAFA5-E914-47DA-95CA-C5AB0DC85B11` 문자열을 결합하고 이를 SHA-1 해시 후 Base64로 인코딩한 값을 이 헤더에 포함하여 응답한다.
핸드셰이크 완료 : 클라이언트는 서버의 응답을 확인하고 WebSocket 연결이 성공적으로 업그레이드되었는지 판단한다. 이 단계가 완료되면 클라이언트와 서버는 HTTP 연결에서 WebSocket 연결로 전환하여 양방향 통신을 시작할 수 있다.

2. 메시지 프레임 : 핸드셰이크가 완료되면 클라이언트와 서버는 메시지를 "프레임" 단위로 주고 받는다. WebSocket 프레임은 작은 데이터 블록으로 이를 통해 텍스트 또는 바이너리 데이터(컴퓨터가 이해하는 0과 1로만 이루어진 데이터)를 전송할 수 있다. 프레임은 최소한의 헤더 정보만을 포함해 매우 효율적이다.
3. 서버 푸시 : WebSocket의 중요한 기능 중 하나는 서버가 클라이언트에 데이터를 "푸시"할 수 있다는 것이다. 클라이언트가 요청하지 않더라도 서버는 언제든지 클라이언트로 데이터를 보낼 수 있다.
4. 연결 종료 :
    1. 종료 요청 : WebSocket 연결을 종료하기 위해서는 클라이언트 또는 서버에서 연결 종료 프레임을 보낸다. 이 프레임은 연결이 종료될 이유를 설명하는 코드를 포함할 수 있다.
    2. 종료 코드 : 종료 코드는 연결이 왜 종료되었는지를 나타낸다. 정상적인 종료의 경우 코드 `1000` , 프로토콜 오류의 경우 코드 `1002` 등을 사용할 수 있다.

3. 정상 종료 과정 : 한쪽에서 종료 요청을 보내면 다른 쪽에서도 동일한 종료 요청을 전송해 응답해야 한다. 이를 통해 양측 모두 정상적으로 연결이 종료되었음을 확인하고 필요한 정리 작업을 수행한다.

장점

1. 낮은 지연시간 : 지속적인 연결을 유지하기 때문에 데이터 전송의 지연이 적다.
2. 네트워크 효율성 : 연결을 유지하고 있는 동안 데이터 프레임만 전송되므로 네트워크 오버헤드가 감소한다.
3. 서버 푸시 : 서버는 클라이언트의 요청 없이도 데이터를 보낼 수 있다.

단점

4. 브라우저 호환성 : 모든 브라우저에서 완벽하게 지원되지 않을 수 있다.
5. 복잡성 : HTTP보다 구현이 복잡할 수 있다.
6. 보안 : WebSocket 연결을 악용한 해킹에 취약할 수 있다.

WebSocket과 다른 프로토콜 비교

WebSocket vs HTTP

특징	WebSocket	HTTP
연결 방식	지속적인 연결	요청-응답 방식
데이터 전송	양방향 실시간	단방향 (클라이언트 → 서버)
오버헤드	낮음	높음 (헤더 정보 등)
사용 사례	채팅, 게임, 실시간 알림 등	웹 페이지 로딩, API 호출 등

WebSocket vs Server-Sent Events (SSE)

SSE는 서버에서 클라이언트로 데이터를 일방적으로 전송하는 기술

특징	WebSocket	SSE
연결 방식	지속적인 연결	지연된 응답
데이터 전송	양방향 실시간	단방향 (서버 → 클라이언트)
오버헤드	낮음	중간
사용 사례	채팅, 게임, 실시간 알림 등	실시간 알림, 뉴스 피드 등

결론 프로토콜 선택시

1. 실시간 양방향 통신이 필요한 경우 : WebSocket
2. 단방향 통신으로 충분한 경우 : SSE
3. 간단한 요청-응답 방식이면 충분한 경우 : HTTP

마무리

WebSocket은 실시간 양방향 통신이 필요한 웹 애플리케이션 개발에 매우 유용한 기술이다. 하지만 단점도 존재하므로 프로젝트의 특성에 맞게 적절하게 사용해야 한다.

HTTP 상태 코드는 클라이언트와 서버 간의 요청-응답 사이에서 발생한 상황을 나타내는 3자리 숫자 코드이다. 이 코드는 클라이언트의 요청이 성공적으로 처리되었는지 아니면 어떤 종류의 오류가 발생했는지를 알려준다.

1xx: 정보 응답

1. 100 Continue : 클라이언트의 요청이 일부분만 수신되었으나 서버는 나머지를 계속해서 전송하도록 요청하다.
2. 101 Switching Protocols : 클라이언트가 프로토콜 전환을 요청하고 서버가 이에 동의함.
3. 102 Processing : 서버가 요청을 수신했으며 아직 처리 중임을 나타낸다.

2xx: 성공

1. 200 OK: 가장 일반적인 코드로, 요청이 성공적으로 처리되었음을 의미한다.
2. 201 Created : 요청이 성공적이며, 새로운 리소스가 생성되었다.
3. 202 Accepted : 요청이 수락되었지만 아직 처리되지 않았다. 비동기 처리의 경우에 많이 사용된다.
4. 204 No Content : 요청이 성공적으로 처리되었지만 응답 본문이 없다.

3xx: 리다이렉션

1. 301 Moved Permanently : 요청한 리소스가 영구적으로 다른 URL로 이동되었음을 의미 클라이언트는 앞으로 새로운 URL을 사용해야 한다.
2. 302 Found : 요청한 리소스가 일시적으로 다른 URL로 이동되었음을 나타낸다. 클라이언트는 동일한 URL로 다시 요청할 수 있다.
3. 304 Not Modified : 클라이언트가 캐시된 리소스를 사용할 수 있음을 알린다. 리소스가 변경되지 않았기 때문에 다시 전송할 필요가 없다.

4xx: 클라이언트 오류

1. 400 Bad Request : 클라이언트의 요청이 잘못되어 서버가 이해할 수 없음 잘못된 문법이나 유효하지 않은 요청으로 발생한다.
2. 401 Unauthorized : 인증이 필요함 클라이언트가 올바른 인증을 제공하지 않았거나 제공된 인증 정보가 유효하지 않음.
3. 403 Forbidden : 인증은 되었지만 권한이 없어 접근할 수 없음을 의미한다.
4. 404 Not Found : 요청한 리소스를 찾을 수 없음 존재하지 않는 URL에 대한 요청일 경우 주로 발생한다.
5. 405 Method Not Allowed : 요청한 HTTP 메서드가 지원되지 않음 POST 요청만 허용된 리소스에 GET 요청을 보낸 경우.

5xx: 서버 오류

1. 500 Internal Server Error : 서버에서 알 수 없는 오류가 발생하여 요청을 처리할 수 없음.
2. 502 Bad Gateway : 서버가 게이트웨이 또는 프록시 역할을 하는 동안 잘못된 응답을 수신했을 때 발생한다.
3. 503 Service Unavailable : 서버가 현재 요청을 처리할 수 없음. 주로 서버 과부하나 유지 보수로 인해 발생한다.
4. 504 Gateway Timeout : 서버가 게이트웨이 또는 프록시로 동작하는 동안 다른 서버로부터 응답을 기다리는 시간이 초과되다.

위와 같은 상태 코드로 서버가 요청을 어떻게 처리했는지에 대한 중요한 정보를 제공하며 클라이언트와 서버 간의 통신에서 발생하는 문제를 파악할 수 있다.

캐싱(Caching)은 데이터를 미리 저장해두고 동일한 데이터 요청 시 저장된 데이터를 활용하여 서버와의 통신을 최소화하고 성능을 향상시키는 기술이다. 이를 통해 웹 페이지 로딩 속도를 개선하고 서버 부하를 줄이는 효과를 얻는다.

브라우저 캐싱

웹 사이트 방문 시 브라우저가 서버에서 받은 리소스(이미지, CSS, JavaScript 파일 등)를 사용자의 로컬 저장소에 임시로 저장해 두는 방식이다. 이 캐싱을 통해 동일한 웹 페이지나 리소스를 다시 요청할 때 서버에 재요청하지 않고 브라우저에 저장된 데이터를 사용하여 페이지 로딩 속도를 빠르게 할 수 있다. 이는 웹 성능을 크게 개선하고 서버 부하를 줄이는 효과가 있다.

브라우저 캐싱의 동작 방식

1. 초기 요청 : 사용자가 웹 페이지를 처음 방문할 때 브라우저는 서버에 요청을 보내고 필요한 리소스(HTML, CSS, JavaScript, 이미지 등)를 다운로드한다.
2. 캐시 저장 : 서버에서 받은 리소스는 브라우저의 로컬 캐시에 저장된다. 이 캐시는 사용자의 하드 드라이브나 메모리에 저장되며, 다음 요청에 재사용할 수 있다.
3. 재요청 시 캐시 사용 : 동일한 리소스를 다시 요청할 경우 브라우저는 먼저 로컬 캐시를 확인한다. 캐시된 리소스가 유효하다면 서버에 다시 요청하지 않고 캐시에서 리소스를 불러온다. 이를 캐시 히트(cache hit)라고 하며 서버에 다시 요청할 필요가 없으므로 네트워크 트래픽을 줄이고 로딩 속도를 개선한다.
4. 캐시 미스: 만약 캐시된 리소스가 없거나 유효하지 않다면 브라우저는 서버에 재요청을 보내 최신 리소스를 받아온다 이를 캐시 미스(cache miss)라고 하며 새로운 리소스는 다시 브라우저 캐시에 저장된다.

브라우저 캐싱 설정 방법

브라우저 캐싱은 주로 서버에서 설정한 HTTP 헤더를 통해 관리되며, 중요한 헤더로는 `Cache-Control`, `Expires`, `ETag` 등이 있다. 각각의 헤더는 캐시의 유효 기간이나 업데이트 조건 등을 제어한다.

1. `Cache-Control` 헤더
   `Cache-Control`은 캐시 정책을 정의하는 주요 HTTP 헤더이다. 이를 통해 브라우저가 리소스를 얼마나 오래 캐시해야 하는지 캐시된 데이터를 사용할지 여부 등을 결정할 수 있다.
   1. `max-age` : 캐시의 유효 기간을 초 단위로 지정하다. `Cache-Control: max-age=3600`은 브라우저가 리소스를 1시간 동안(3600초) 캐시해야 한다는 의미이다.

    2. `no-cache` : 캐시된 리소스를 사용할 수 있지만 매번 서버에 확인 요청을 보내 최신 버전인지 체크한다.
    3. `no-store` : 리소스를 캐시하지 않고 매번 서버에서 데이터를 받아온다.
    4. `public` : 리소스가 모든 캐시(브라우저뿐만 아니라 중간 서버도 포함)에서 캐시될 수 있음을 나타낸다.
    5. `private` : 리소스가 브라우저 캐시에만 저장되며 중간 서버에서는 캐시되지 않음을 나타낸다.
2. `Expires` 헤더
   `Expires`는 특정 시점을 기준으로 캐시된 리소스가 만료되는 시간을 설정하는 헤더이다. `Expires: Wed, 21 Oct 2023 07:28:00 GMT`는 해당 시간 이후에는 브라우저가 캐시를 사용하지 않고 서버에서 다시 리소스를 요청하게 한다. 그러나 현재는 `Cache-Control`의 `max-age`가 더 많이 사용되며 `Expires`는 구식 방식이다.
3. `ETag` 헤더
   `ETag`는 서버가 리소스의 고유 식별자를 제공하여 브라우저가 캐시된 리소스가 여전히 최신 상태인지 확인하는 데 사용된다. 브라우저가 서버에 캐시된 리소스가 여전히 유효한지 요청할 때 서버는 `ETag` 값을 비교하여 동일한 경우 캐시된 리소스를 그대로 사용하고 변경된 경우 새로운 리소스를 제공할 수 있다.
4. `Last-Modified` 헤더
   `Last-Modified`는 리소스가 마지막으로 수정된 시간을 서버가 브라우저에 알려주는 헤더이다. 브라우저는 이 값을 기억하고 서버에 재요청할 때 `If-Modified-Since` 헤더로 확인 요청을 보낸다. 만약 리소스가 변경되지 않았다면 서버는 304 Not Modified 응답을 보내고 브라우저는 캐시된 리소스를 계속 사용한다.

브라우저 캐싱은 웹 성능을 개선하고 사용자 경험을 향상시키는 중요한 기술이다. 적절한 HTTP 헤더 설정을 통해 캐싱 정책을 잘 관리하면 웹 페이지의 응답 속도를 크게 높이고 서버 부하를 줄일 수 있다.

CDN 캐싱(Content Delivery Network Caching)

콘텐츠 전달 네트워크(CDN)를 활용해 데이터를 사용자에게 더 가까운 위치에 캐시하는 방식이다. CDN은 전 세계 여러 지역에 분산된 서버 네트워크로 구성되며 사용자와 가까운 서버(에지 서버)에 콘텐츠를 저장해 두었다가 빠르게 제공함으로써 성능을 최적화한다.

CDN 캐싱의 동작 방식

1. 콘텐츠 요청 : 사용자가 웹 사이트에 접속하면, 브라우저는 필요한 리소스(이미지, CSS, JavaScript 파일 등)를 요청합니다.

2. 에지 서버 확인 : 사용자의 요청은 CDN 네트워크 내에서 가장 가까운 서버 에지 서버로 전달된다. 이 서버는 요청된 콘텐츠가 캐시되어 있는지 확인한다.
3. 캐시 히트 : 에지 서버에 해당 콘텐츠가 이미 캐시되어 있는 경우 캐시 히트(cache hit)가 발생하며 서버는 빠르게 캐시된 콘텐츠를 사용자에게 제공한다. 이때 원래 웹 서버까지 가지 않고 가까운 위치에서 콘텐츠가 제공되므로 대기 시간(Latency)이 줄어든다.
4. 캐시 미스 : 만약 에지 서버에 콘텐츠가 없는 경우 캐시 미스(cache miss)가 발생한다. 에지 서버는 원본 서버로부터 콘텐츠를 받아오고 이를 사용자에게 전달하면서 동시에 자신의 캐시에 저장해 둔다. 이후 같은 콘텐츠를 요청하는 사용자에게는 에지 서버가 캐시된 콘텐츠를 제공한다.

CDN 캐싱의 장점

1. 속도 향상: 사용자와 가까운 에지 서버에서 데이터를 제공하기 때문에 웹 사이트 로딩 속도가 크게 향상된다.
2. 서버 부하 감소 : 원본 서버로의 요청이 줄어들어 서버에 가해지는 부하가 감소한다. 이는 대규모 트래픽을 처리할 때 서버 다운이나 속도 저하를 방지하는 데 유리하다.
3. 전 세계적 접근성 향상 : CDN은 다양한 지역에 분산된 서버를 통해 글로벌 사용자에게도 빠른 서비스 제공이 가능하다. 지역 간의 네트워크 지연을 최소화한다.
4. 대역폭 비용 절감 : CDN을 활용하면 서버로 가는 직접적인 트래픽이 줄어들어 대역폭 비용이 절감될 수 있다.
5. 보안 강화 : 많은 CDN 제공업체는 DDoS 방어, SSL 인증서 지원 등 다양한 보안 기능을 제공하여 원본 서버를 보호하고 안전한 콘텐츠 전송을 보장한다.

CDN 캐싱 설정

CDN 캐싱은 서버의 설정이나 콘텐츠 제공 방법에 따라 다양한 방식으로 구현된다.

1. TTL(Time To Live) : 콘텐츠의 유효 기간을 설정하는 방식이다. CDN에 캐시된 콘텐츠는 1시간 동안 유효하도록 설정할 수 있다. 유효 기간이 지나면 CDN 서버는 본 서버로부터 새로운 콘텐츠를 요청한다.
2. `Cache-Control` 헤더 : 이 헤더를 사용하여 CDN과 브라우저에 캐시 정책을 정의할 수 있다. 캐시의 최대 유효 시간을 지정하거나 캐시하지 말아야 할 리소스를 설정할 수 있다.

CDN 캐싱은 사용자 경험을 개선하고 서버의 성능을 높이는 중요한 기술이다. 특히 전 세계적으로 분산된 트래픽을 효율적으로 처리해야 하는 웹 사이트나 애플리케이션에서 필수적인 역할을 한다.

`Cache-Control`은 HTTP 헤더 중 하나로 웹 브라우저나 CDN 같은 캐시 서버가 어떻게 리소스를 캐싱해야 하는지에 대한 지침을 제공한다. 이를 통해 서버가 웹 페이지 또는 리소스의 캐싱 동작을 제어할 수 있으며 클라이언트(브라우저)와 서버 간의 효율적인 데이터 전송을 할 수 있다. `Cache-Control` 헤더는 캐싱 기간, 캐싱 여부, 캐시된 데이터를 재검증하는 방식 등 다양한 캐싱 정책을 정의할 수 있다.

`Cache-Control`의 주요 디렉티브

`Cache-Control` 헤더는 여러 디렉티브를 포함할 수 있으며 각각 캐시의 동작 방식을 제어을 할 수 있다.

1. `max-age=<seconds>`
    1. 설명 : 리소스를 캐시할 최대 기간(초 단위)을 정의한다. 이 기간 동안 브라우저는 서버에 재요청하지 않고 캐시된 데이터를 사용한다.
    2. 예시 : `Cache-Control: max-age=3600`은 1시간 동안(3600초) 캐시된 데이터를 사용하도록 설정한다.
2. `s-maxage=<seconds>`
    1. 설명 : 공유 캐시(CDN, 프록시 서버 등)에서의 캐싱 유효 기간을 지정한다. 브라우저의 `max-age`와는 별도로 공유 캐시 서버의 캐싱 정책을 설정할 수 있다.
    2. 예시 : `Cache-Control: s-maxage=86400`은 CDN이나 프록시 서버가 24시간(86400초) 동안 캐시된 데이터를 사용할 수 있게 한다.
3. `public`
    1. 설명 : 리소스가 공유 캐시(CDN, 프록시 서버 등)과 브라우저 캐시에 저장될 수 있음을 나타낸다. 캐시가 모든 사용자에게 공유될 수 있음을 의미한다.
    2. 예시 : `Cache-Control: public`은 해당 리소스가 공유 캐시와 브라우저 캐시에 저장될 수 있도록 허용한다.
4. `private`
    1. 설명 : 리소스가 브라우저 캐시에만 저장되고 공유 캐시에는 저장되지 않음을 나타낸다. 개인화된 데이터나 사용자별 데이터를 캐시할 때 주로 사용된다.
    2. 예시 : `Cache-Control: private`은 해당 리소스가 브라우저에만 캐시되고 프록시나 CDN 같은 공유 캐시에는 저장되지 않도록 한다.
5. `no-cache`
    1. 설명 : 캐시된 리소스를 사용하기 전에 서버에 재검증을 요청해야 함을 나타낸다. 서버에서 최신 리소스인지 확인한 후 캐시된 리소스를 사용할 수 있다.
    2. 예시 : `Cache-Control: no-cache`는 캐시된 데이터를 사용할 수는 있지만 매번 서버에 확인을 요청해야 한다.
6. `no-store`
    1. 설명 : 리소스를 캐시하지 않도록 설정한다. 브라우저나 중간 캐시 서버에 데이터를 저장하지 않으며 매번 서버에서 리소스를 다시 받아온다. 민감한 정보(예: 개인 정보, 금융 정보)를 다룰 때 사용된다.

    2. 예시 : `Cache-Control: no-store` 는 캐시를 완전히 금지하여 리소스가 브라우저나 중간 서버에 저장되지 않도록 한다.
7. `must-revalidate`
    1. 설명 : 캐시된 리소스의 유효 기간이 끝나면 서버에 재검증을 요청해야 한다. 브라우저가 만료된 캐시를 사용할 수 없도록 한다.
    2. 예시 : `Cache-Control: must-revalidate` 는 캐시된 데이터가 만료된 경우 서버에 재검증을 요청해야 한다.
8. `proxy-revalidate`
    1. 설명 : 프록시 서버가 캐시된 리소스를 재검증하도록 설정한다. `must-revalidate` 와 비슷하지만 이 설정은 브라우저가 아닌 공유 캐시(프록시 서버)에만 적용된다.
    2. 예시 : `Cache-Control: proxy-revalidate` 는 프록시 서버가 캐시된 리소스를 사용할 때 재검증을 요구한다.
9. `no-transform`
    1. 설명 : 중간 캐시 서버나 프록시 서버가 리소스를 변환하지 못하도록 설정한다. 이미지의 포맷을 변경하거나 데이터를 압축하지 않도록 한다.
    2. 예시 : `Cache-Control: no-transform` 은 이미지나 리소스가 중간 서버에 의해 변경되지 않도록 보장한다.
10. `immutable`
11. 설명 : 리소스가 변경되지 않음을 보장한다. 리소스는 절대 변경되지 않으므로 브라우저는 서버에 재검증 요청 없이 항상 캐시된 리소스를 사용할 수 있다. 주로 버전이 명확하게 관리되는 파일(CSS, JS)에서 사용된다.
12. 예시 : `Cache-Control: immutable` 은 브라우저가 리소스를 캐시에 저장하고 절대 서버에 다시 요청하지 않도록 설정한다.
13. `stale-while-revalidate=<seconds>`
    1. 설명 : 캐시된 리소스가 만료되었더라도 새로운 리소스를 받아오는 동안 잠시 만료된 캐시를 사용할 수 있도록 허용한다. 서버에서 데이터를 다시 받는 동안 사용자는 캐시된 데이터를 받아볼 수 있어 로딩 시간을 줄이는 데 도움이 된다.
    2. 예시 : `Cache-Control: stale-while-revalidate=60` 은 리소스가 만료된 후 60초 동안은 만료된 캐시를 사용하며 백그라운드에서 새로운 데이터를 받아온다.
14. `stale-if-error=<seconds>`
    1. 설명 : 서버에서 오류가 발생할 경우 캐시된 리소스가 만료되었더라도 오류 발생 시 캐시된 데이터를 사용할 수 있도록 허용한다. 서버 다운이나 네트워크 장애 시 유용하게 사용된다.
    2. 예시 : `Cache-Control: stale-if-error=300` 은 서버에서 오류가 발생할 경우 5분간은 만료된 캐시를 사용할 수 있게 한다.

`Cache-Control` 의 예시

1. 일반 캐싱

```
Cache-Control: public, max-age=3600
```

이 설정은 리소스를 1시간 동안 캐시하며 브라우저와 CDN 같은 공유 캐시 서버에서도 사용할 수 있도록 허용한다.

2. 민감한 데이터 캐싱 금지

```
Cache-Control: no-store
```

이 설정은 브라우저나 공유 캐시 서버에 절대 리소스를 저장하지 않고 매번 서버에서 새롭게 데이터를 가져오도록 강제한다.

3. 재검증 설정

```
Cache-Control: no-cache, must-revalidate
```

이 설정은 브라우저가 캐시된 리소스를 사용하기 전에 항상 서버에 재검증을 요청하고 만료된 리소스는 반드시 재검증해야 한다는 의미이다.

`Cache-Control` 헤더는 웹 페이지 성능 최적화에서 매우 중요한 역할을 한다. 이를 통해 리소스가 얼마나 오래 어디에서 어떻게 캐시되어야 하는지를 명확하게 정의할 수 있으며 적절한 캐싱 정책을 적용함으로써 웹 성능을 크게 개선하고 서버 부하를 줄일 수 있다. 캐시 정책은 리소스의 성격 업데이트 빈도 민감도에 따라 세밀하게 설정되어야 한다.

로드 밸런싱(Load Balancing)은 여러 대의 서버에 들어오는 트래픽을 분산하여 시스템의 성능을 향상시키고 장애 발생 시에도 서비스의 가용성을 높이는 기술이다.

로드 밸런싱의 개념

1. 로드 밸런서(Load Balancer) :
    1. 클라이언트의 요청을 받아 여러 서버에 분산하는 역할을 하는 장비나 소프트웨어이다.
    2. 하드웨어 로드 밸런서와 소프트웨어 로드 밸런서가 있다.
2. 로드 밸런싱 알고리즘 :
    1. 라운드 로빈(Round Robin) : 요청을 순서대로 각 서버에 분산하는 방식이다.
    2. 최소 연결(Least Connections) : 현재 연결 수가 가장 적은 서버에 요청을 분산하는 방식이다.
    3. IP 해시(IP Hash) : 클라이언트의 IP 주소를 해싱하여 특정 서버에 요청을 전달하는 방식이다.
    4. 가중치 기반(Weighted) : 서버의 능력에 따라 가중치를 부여하고 그 가중치에 따라 요청을 분산한다.
3. 세션 유지(Session Persistence) :
    1. 특정 클라이언트의 요청이 항상 동일한 서버로 전달되도록 하는 기능이다. 이를 통해 세션 상태를 유지할 수 있다. 방법으로는 쿠키 기반, IP 기반 등이 있다.
4. 장애 조치(Failover) :
    1. 서버 중 하나가 다운되었을 때 로드 밸런서가 자동으로 다른 서버로 트래픽을 전환하여 서비스를 계속 유지하는 기능이다.
5. 스케일링(Scaling) :
    1. 수요 증가에 따라 서버를 추가하거나 제거하여 시스템의 성능을 조절할 수 있다. 로드 밸런서가 자동으로 이 작업을 지원하기도 한다.

로드 밸런서의 종류

1. 하드웨어 로드 밸런서 :
    1. 물리적인 장비로 제공되는 로드 밸런서이다. 고성능을 제공하지만 비용이 높은 경우가 많다.
2. 소프트웨어 로드 밸런서 :
    1. 소프트웨어로 구현된 로드 밸런서이다. 유연성과 비용 측면에서 유리하며 클라우드 환경에서 많이 사용된다.
3. 클라우드 기반 로드 밸런서 :
    1. AWS의 Elastic Load Balancing(ELB), Azure의 Azure Load Balancer, Google Cloud의 Cloud Load Balancing 등 클라우드 서비스 제공자가 제공하는 로드 밸런서이다.

로드 밸런싱은 대규모 웹 애플리케이션이나 서비스에서 성능과 신뢰성을 유지하는 데 필수적인 기술이다. 구현 방법과 필요성은 시스템의 요구 사항에 따라 달라질 수 있다.

OAuth(Open Authorization)는 제3자 애플리케이션이 사용자 자격 증명을 직접 알지 못하면서 사용자 리소스에 접근할 수 있도록 허용하는 인증 및 권한 부여하는 방식이다. 주로 소셜 미디어나 웹 서비스에서 많이 사용되며 보안이 중요한 API와 애플리케이션 간의 데이터 접근을 제어하는 데 활용된다.

OAuth는 주로 두 가지 버전이 존재하는데 OAuth 1.0과 OAuth 2.0이 있다. 현재는 더 간단하고 보안이 강화된 OAuth 2.0이 표준으로 사용된다.

OAuth 2.0 구성 요소

1. 클라이언트(Client)
    1. 정의 : 사용자로부터 권한을 받아 리소스 서버에 접근하려는 애플리케이션이다.
    2. 예시 : 모바일 앱, 웹 애플리케이션, 데스크톱 애플리케이션 등
2. 리소스 소유자(Resource Owner)
    1. 정의 : 자신의 데이터에 대한 권한을 가지고 있는 사용자이다. 사용자가 자신의 데이터를 제3자에게 제공할 수 있는 권한을 부여한다.
    2. 예시 : 구글 계정 사용자, 페이스북 계정 사용자 등
3. 인증 서버(Authorization Server)
   3. 정의 : 사용자의 인증을 처리하고 클라이언트가 적절한 권한을 부여받았는지 검증하는 서버이다. 인증 서버는 액세스 토큰을 발급한다.
   4. 예시 : 구글 OAuth 서버, 페이스북 OAuth 서버 등
4. 리소스 서버(Resource Server)
    1. 정의 : 사용자의 데이터가 저장된 서버로, 액세스 토큰을 통해 요청된 데이터를 클라이언트에 제공합니다. 이 서버는 클라이언트가 제출한 액세스 토큰의 유효성을 확인한 후 데이터 접근을 허용한다.
    2. 예시 : 구글 API 서버, 페이스북 API 서버 등
5. 액세스 토큰(Access Token)
   3. 정의 : 클라이언트가 리소스 서버에 대한 접근 권한을 증명하는 증표이다.
   4. 특징 :
   1. 비밀 유지가 중요하며 일반적으로 만료 시간이 설정된다.
   2. 리소스 서버는 액세스 토큰을 검증하여 클라이언트의 권한을 확인한다.

OAuth 2.0의 개념과 동작 방식

OAuth에는 다양한 흐름이 있지만 주로 권한 부여 코드 그랜트(Authorization Code Grant) 방식을

사용된다.

1. 사용자(리소스 오너)가 클라이언트(앱)에 로그인을 요청한다.
2. 클라이언트는 인증 서버로 사용자를 리디렉션한다.
3. 사용자는 인증 서버에서 자신의 아이디와 비밀번호를 입력하고 클라이언트에게 특정 정보에 대한 접근을 허용한다.
4. 인증 서버는 클라이언트에게 권한 부여 코드를 발급한다.
5. 클라이언트는 권한 부여 코드를 이용하여 인증 서버에 액세스 토큰을 요청한다.
6. 인증 서버는 액세스 토큰을 발급하고 클라이언트는 이 토큰을 이용하여 리소스 서버에 접근한다.
7. 리소스 서버는 액세스 토큰의 유효성을 검증하고 요청된 리소스(예: 사용자 정보, 파일 등)를 클라이언트에게 제공한다.

예시 :

1. **A 앱**에서 **구글 드라이브**의 파일을 열려고 한다고 가정해 보자.
2. 사용자가 A 앱에서 구글 로그인을 클릭하면 A 앱은 사용자를 구글의 인증 서버로 이동시킨다.
3. 사용자가 구글 계정으로 로그인하고 A 앱에 자신의 구글 드라이브에 접근할 수 있는 권한을 부여한다.
4. 구글은 A 앱에게 **권한 부여 코드**를 발급한다.
5. A 앱은 이 코드를 이용하여 구글에 **액세스 토큰**을 요청한다.
6. 구글은 A 앱에게 액세스 토큰을 발급하고 A 앱은 이 토큰을 이용하여 구글 드라이브에 접근하여 파일을 열 수 있다.

추가 설명

1. 액세스 토큰 : 일정 시간이 지나면 만료되며 비밀 유지가 매우 중요하다.
2. 리프레시 토큰 : 액세스 토큰이 만료되었을 때 새로운 액세스 토큰을 발급받기 위해 사용하는 토큰이다.
3. 스코프 : 클라이언트가 요청할 수 있는 권한의 범위를 정의한다. 예시 사용자의 프로필만 읽을 수 있는 권한, 이메일을 읽고 보낼 수 있는 권한 등이 있다.

OAuth 1.0 구성 요소

1. 소비자 (Consumer)

    1. 정의 : 사용자의 자원에 접근하려는 애플리케이션이다.
    2. 예시 : 트위터 클라이언트 애플리케이션, 타사 웹 애플리케이션 등.
2. 서비스 제공자 (Service Provider)
    1. 정의 : 사용자의 자원을 호스팅하고 이를 보호하는 서비스다. Consumer에게 인증 및 권한을 부여한다.
    2. 예시 : 트위터, 구글, 페이스북 등의 API 제공자
3. Request Token
    1. 정의 : Consumer가 사용자의 권한을 얻기 전에 발급받는 임시 토큰이다. 이 토큰은 사용자의 승인을 받기 위한 과정에서 사용된다.
    2. 특징 :
        1. 임시 토큰으로, 사용자가 승인하면 Access Token으로 변환된다.
        2. 서비스 제공자가 발급한다.
4. Access Token
    1. 정의 : 사용자가 승인 절차를 거친 후 Consumer에게 발급되는 최종 토큰으로 이를 통해 서비스 제공자의 API에 접근할 수 있다.
    2. 특징 :
        1. Request Token을 승인 후 교환하여 발급된다.
        2. 각 요청마다 서명과 함께 사용되며, 만료되거나 재사용이 제한된다.
5. 서명 (Signature)
    1. 정의 : 각 API 요청마다 암호화된 서명을 포함하여 Consumer가 요청의 무결성을 보장하는 방식이다.
    2. 특징 :
        1. 요청의 변조를 방지하며, Consumer Key, Secret, Request Token, 요청 메서드, URL 등 다양한 요소를 조합해 생성된다.
        2. 서명을 통해 서비스 제공자는 요청이 올바른지 확인할 수 있다.

OAuth 1.0의 개념과 동작 방식

1. Request Token 요청
    1. Consumer는 Service Provider에 Request Token을 요청한다. 이 과정에서 Consumer Key와 Consumer Secret이 함께 전송된다.
2. 사용자 승인
    1. Request Token을 발급받은 후, 사용자는 Consumer가 자신의 자원에 접근할 수 있도록 Service Provider를 통해 승인을 요청받는다.
3. Access Token 발급
    1. 사용자가 승인을 완료하면 Service Provider는 Request Token을 Access Token으로 변환하여 Consumer에게 전달한다.
4. 서명을 통한 요청

    1. Consumer는 Access Token을 사용하여 Service Provider의 자원에 접근할 때마다 서명된 요청을 전송한다. 서명은 요청의 변조 여부를 확인하기 위해 사용된다.
5. 자원 접근
    1. Access Token을 소유한 Consumer는 서비스 제공자의 API를 통해 사용자의 자원에 안전하게 접근할 수 있다.

예시 :

1. 트위터 애플리케이션이 사용자 A의 트윗 목록에 접근하려고 한다고 가정하자.
2. 트위터 애플리케이션은 트위터 서버(Service Provider)에 Request Token을 요청한다.
3. 사용자 A는 트위터 로그인 화면을 통해 애플리케이션에 권한을 부여한다.
4. 트위터 서버는 애플리케이션에 Access Token을 발급한다.
5. 애플리케이션은 Access Token을 사용하여 사용자 A의 트윗 목록을 불러온다.

추가 설명

1. OAuth 1.0은 요청의 무결성을 보장하기 위해 서명을 필수적으로 요구하며 이를 통해 높은 보안성을 제공한다.
2. 하지만 서명 방식과 Request/Access Token 처리 과정이 복잡해서 OAuth 2.0에서는 이를 간소화한 방식이 도입되었다.

OAuth 1.0과 OAuth 2.0 차이

OAuth 1.0과 OAuth 2.0은 웹 애플리케이션과 API 간의 인증을 위한 프로토콜로 사용자 자격 증명 없이 안전하게 권한을 부여할 수 있도록 돕는 방식이다. 하지만 두 버전 사이에는 중요한 차이점들이 있다.

1. **복잡성 및 설계 철학**
    1. OAuth 1.0 : 복잡한 인증 프로세스를 사용한다. OAuth 1.0은 주로 암호화된 서명을 사용해 요청의 무결성을 보장하며 각 요청에 서명을 생성하고 이를 검증하는 과정이 포함되어 있다. 따라서 구현이 복잡하고 관리할 요소가 많다.
    2. OAuth 2.0 : OAuth 2.0은 사용하기 쉽게 설계되었으며 보안은 HTTPS(SSL/TLS)에 의존한다. 암호화된 서명보다는 HTTPS로 트래픽을 보호하는 방식을 사용하기 때문에 OAuth 1.0보다 더 단순하고 개발자들이 쉽게 사용할 수 있다.
2. **서명(Signature) 방식**
    1. OAuth 1.0 : OAuth 1.0에서는 각 요청이 반드시 서명되어야 한다. 이 서명은 클라이언트 비밀키를 사용하여 암호화 알고리즘을 통해 생성되며 요청 내용과 매개변수 등이

포함된다. 서명 생성 과정이 복잡하고 이를 검증하는 데 추가적인 계산이 필요하다.

2. OAuth 2.0 : OAuth 2.0은 서명을 사용하지 않는다. 대신 HTTPS를 사용하여 데이터 전송 중 보안을 유지하며 클라이언트 인증에는 액세스 토큰을 사용한다. 클라이언트 비밀키는 토큰을 발급받을 때만 사용된다. 그 외의 요청은 단순히 액세스 토큰을 포함하여 서버와 통신한다.

3. **액세스 토큰 발급 방식**
    1. OAuth 1.0 : OAuth 1.0에서는 **Request Token**과 **Access Token**이라는 두 단계의 토큰을 사용한다. 먼저 리소스 소유자가 인증 서버에서 요청 토큰을 발급받고 이 토큰을 사용해 액세스 토큰을 교환하는 방식이다. 이러한 이중 토큰 방식은 좀 더 복잡하다.
    2. OAuth 2.0 : OAuth 2.0에서는 권한 부여 흐름에 따라 **Access Token**을 직접 발급받는다. 토큰 발급 과정이 단순화되었으며 Refresh Token을 사용하여 만료된 Access Token을 새로 발급받을 수 있다.
4. **토큰 유형 및 갱신**
    1. OAuth 1.0 : OAuth 1.0에서는 토큰 갱신(refresh)이 없다. 액세스 토큰이 만료되면 새로운 토큰을 발급받아야 한다.
    2. OAuth 2.0 : OAuth 2.0에서는 **Refresh Token**이 도입되었다. 이는 액세스 토큰이 만료되었을 때 새로 인증을 하지 않고도 액세스 토큰을 다시 발급받을 수 있도록 한다. 따라서 사용자 경험이 더 매끄럽다.
5. **권한 부여 방식**
    1. OAuth 1.0 : OAuth 1.0은 기본적으로 인증 코드 흐름만을 사용한다. 사용자 자격 증명을 활용해 인증 서버로부터 권한을 부여받고 그에 따라 토큰을 발급받는 단일 방식만을 지원한다.
    2. OAuth 2.0 : OAuth 2.0에서는 다양한 권한 부여 방식을 지원한다. 각 상황에 맞는 권한 부여 방식을 선택할 수 있다.
        1. Authorization Code Grant : 일반적인 웹 애플리케이션 인증에 사용.
        2. Implicit Grant : 브라우저 기반 애플리케이션에 사용.
        3. Resource Owner Password Credentials Grant : 사용자가 애플리케이션에 직접 아이디와 비밀번호를 제공하는 방식.
        4. Client Credentials Grant : 서버 간 통신에 사용.
        5. Device Code Grant : 제한된 입력을 허용하는 장치(예: TV, IoT 기기)에서 사용.
6. **보안**

1. OAuth 1.0 : OAuth 1.0은 모든 요청에 서명이 포함되므로 트래픽이 평문으로 전달되더라도 서명 검증을 통해 요청이 위조되지 않았는지 확인할 수 있다. 그러나 암호화 서명을 사용하는 방식이 복잡하여 개발자가 구현하는 데 어려움이 있다.

2. OAuth 2.0 : OAuth 2.0은 모든 트래픽을 HTTPS로 보호하는 것을 전제로 한다. HTTPS를 통해 기밀성(허가되지 않은 사람이나 시스템이 정보에 접근하는 것을 방지하는 것)과 무결성(정보가 생성되고 전송되는 과정에서 변조나 손상 없이 원본 그대로 유지되는 것)을 보장하며 토큰 발급 및 사용 과정에서 추가적인 보안 메커니즘을 도입할 수 있다. 그러나 OAuth 1.0과 비교했을 때 보안이 HTTPS에만 의존하므로 HTTPS 설정이 제대로 되어 있지 않으면

보안 문제가 발생할 수 있다.

결론

OAuth 1.0 : 보안은 강력하지만 복잡하고 구현이 어렵다.

OAuth 2.0 : 더 단순하고 유연하며, HTTPS 보안에 의존하여 더 광범위한 사용을 지원한다.

JWT(Json Web Token) 인증 방식은 클라이언트와 서버 간의 안전한 인증을 위한 토큰 기반 인증 방법이다. JWT는 주로 사용자 인증과 권한 부여에서 사용되며 세션을 서버에 저장하지 않고 사용자 정보를 담은 토큰을 클라이언트 측에 저장해 사용하는 방식이다.

JWT의 구조

1. 헤더(Header) : 토큰의 형식(JWT)과 사용된 알고리즘 등에 대한 정보를 담고 있다.
2. 페이로드(Payload) : 실제 사용자 정보를 담는 부분이다. 사용자 ID, 이름, 권한 등 다양한 정보를 포함할 수 있다.
3. 서명(Signature) : 헤더와 페이로드를 연결하고 토큰의 무결성을 보장하기 위한 부분이다. 비밀 키를 사용하여 해시 함수를 적용하여 생성된다.

JWT 형식 예시

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9
```

JWT의 저장 위치

1. 로컬 스토리지 vs 쿠키
    1. 로컬 스토리지 : 브라우저의 로컬 스토리지에 저장되면 서버와의 통신에 자동으로 포함되지 않아 사용자가 명시적으로 토큰을 추가해야 한다. 하지만 로컬 스토리지는 XSS(크로스 사이트 스크립팅) 공격에 취약할 수 있다.
    2. 쿠키 : 쿠키에 저장할 경우 `HttpOnly` 속성과 `Secure` 속성을 설정해 XSS와 CSRF(크로스 사이트 요청 위조) 공격을 방지할 수 있다. 하지만 CSRF에 취약할 수 있어 CSRF 방어 전략을 함께 적용하는 것이 중요하다.

JWT 무효화 전략

1. JWT는 자체적으로 상태를 저장하지 않기 때문에 토큰 발급 후 서버에서 무효화하기 어렵다. 이를 해결하기 위해 몇 가지 방법이 있다
    1. 블랙리스트 : 무효화할 JWT를 서버에 저장하고 매 요청마다 토큰을 블랙리스트와 비교하여 차단한다. 그러나 이 방식은 서버의 상태를 저장하게 되어 JWT의 무상태 성격과 모순된다.
    2. 짧은 만료 시간과 리프레시 토큰 사용 : JWT의 만료 시간을 짧게 설정하고 리프레시 토큰을 이용하여 만료된 JWT를 갱신하는 방식이다. 이 방법은 리프레시 토큰을 서버에 저장하여 보다 유연하게 무효화할 수 있다.

JWT 암호화

기본적으로 JWT는 서명을 통해 무결성(변조 방지)을 보장하지만 Payload에 담긴 정보는 누구나

열람할 수 있다. 기밀 데이터가 포함된 경우 JWT를 JWE(Json Web Encryption) 형식으로 암호화하여 전송하는 것이 필요할 수 있다.

알고리즘 선택

1. HS256 (HMAC with SHA-256) : 비밀 키를 사용한 대칭 암호화 방식으로 성능이 좋고 사용이 간단하지만 키가 유출되면 쉽게 공격받을 수 있다.
2. RS256 (RSA Signature with SHA-256) : 비대칭 암호화 방식으로 공개 키와 비밀 키를 사용한다. 서명 검증은 공개 키로 할 수 있어 보안성이 뛰어나지만 성능이 떨어질 수 있습니다.

OAuth 2.0과의 관계

1. JWT는 주로 OAuth 2.0과 함께 사용된다. OAuth 2.0은 권한 부여를 위한 프로토콜이고 JWT는 이러한 프로토콜에서 인증을 위한 토큰으로 사용된다. OAuth 2.0에서 JWT는 **Access Token**이나 **ID Token**으로 활용된다.
2. OAuth 2.0의 **Implicit Flow**(브라우저 기반 애플리케이션에 사용)나 **Authorization Code Flow**(일반적인 웹 애플리케이션 인증에 사용)에서 JWT를 발급하여 클라이언트가 권한을 사용할 수 있도록 하며 이때 리프레시 토큰과 함께 사용해 보안성을 강화할 수 있다.

JWT와 OAuth 2.0 사용 시 보안 고려사항

1. 리프레시 토큰 보안 : 리프레시 토큰은 더 긴 만료 시간을 갖고 새로운 액세스 토큰을 발급하는 데 사용되기 때문에 안전하게 관리해야 한다. 리프레시 토큰도 탈취되면 동일한 위험이 발생할 수 있다.
2. Scope와 권한 관리 : JWT에 담긴 권한(scope) 정보가 잘못 관리되면 불필요하게 높은 권한을 부여받은 토큰이 발급될 수 있다. 따라서 Scope를 적절히 설정하고 관리해야 한다.

JWT 활용 예시

1. 마이크로서비스 아키텍처 : 각 서비스가 분산되어 있을 때, JWT는 무상태 특성 덕분에 각 서비스 간 인증 정보를 공유하는 데 용이하다.
2. 소셜 로그인 : Google, Facebook 등의 소셜 로그인 서비스에서 JWT를 Access Token으로 활용하여 인증을 통합적으로 처리할 수 있다.

JWT 서드 파티 라이브러리

1. 다양한 언어에서 JWT를 다루기 위한 라이브러리가 있다. jsonwebtoken (Node.js), pyjwt (Python), jwt-go (Go), jjwt (Java JWT) 등이 있다.

JWT 와 OAuth 차이점

항목	JWT	OAuth 2.0
개념	토큰 포맷(인증 정보를 담아 전송)	권한 부여 프로토콜(제3자 권한 부여 관리)

항목	JWT	OAuth 2.0
역할	인증 정보와 권한을 포함한 토큰 전달	클라이언트에게 자원에 대한 권한 부여
토큰 형식	JWT 자체가 토큰 형식으로 사용	OAuth 2.0은 JWT와 같은 토큰을 사용하거나 다른 토큰 형식도 사용 가능
목적	사용자 인증 및 권한 정보 전달	제3자에게 특정 자원에 대한 접근 권한 부여
토큰 저장	클라이언트가 JWT를 로컬 스토리지나 쿠키에 저장	OAuth는 서버에서 액세스 토큰을 발급하고 관리
사용 사례	API 인증, 사용자 로그인	소셜 로그인, 외부 앱의 API 권한 부여
무효화	JWT는 자체적으로 상태를 관리하지 않아서 무효화가 어려움	OAuth 2.0은 리프레시 토큰으로 토큰 갱신 가능
서드 파티	JWT는 서버와 클라이언트 간의 인증에 사용	OAuth 2.0은 제3자 클라이언트에 권한 부여를 중점으로 함
보안	JWT는 암호화되지 않고, 서명을 통해 변조를 방지	OAuth 2.0은 토큰을 관리하며, 토큰 만료 및 갱신을 통해 보안을 강화
권한 관리	JWT는 권한 정보(scope)를 직접 포함할 수 있음	OAuth 2.0은 권한 범위(scope)를 명시적으로 관리

1. JWT는 인증 정보나 사용자 권한을 포함한 토큰 형식으로 주로 사용되며 인증 과정에서 서버와 클라이언트 간에 정보를 주고받는 데 유용
2. OAuth 2.0은 사용자로부터 권한을 받아 제3자 클라이언트가 자원에 접근할 수 있게 해주는 권한 부여 프로토콜이다. OAuth 2.0 자체가 토큰 형식을 정의하지는 않지만 JWT와 같은 토큰을 활용할 수 있다.

JWT 인증 과정

1. 사용자 로그인 : 사용자가 로그인 요청을 보내면 서버는 사용자의 자격 증명을 확인하고 확인이 완료되면 사용자 정보와 만료 시간 등의 정보를 담은 JWT를 생성한다.
2. 토큰 발급 : 서버는 JWT를 클라이언트에게 응답으로 반환한다. 이때 서버는 JWT를 자체적으로 저장하지 않는다.
3. 클라이언트 저장 : 클라이언트는 발급받은 JWT를 로컬 스토리지나 쿠키에 저장한다.
4. 인증 요청 : 클라이언트는 이후 인증이 필요한 요청(예시 : API 요청) 시, JWT를 HTTP 헤더(보통 `Authorization: Bearer <토큰>` 형식)로 서버에 전송한다.
5. 토큰 검증 : 서버는 요청을 받을 때 JWT의 유효성을 검증한다. 검증 과정은 Header와 Payload의 정보가 변경되지 않았는지 서명이 올바른지 만료 시간이 지났는지를 확인하는 과정이다.
6. 요청 처리 : 토큰이 유효하면 서버는 해당 사용자의 정보를 추출하여 요청을 처리하고 유효하지 않으면 인증 오류를 반환한다.

JWT의 장점

1. 상태 비저장 : 서버 측에서 세션을 저장할 필요가 없으므로 서버 확장이 용이하다.
2. 확장성 : API 간의 인증에서 특히 유용하며 분산 시스템에서도 쉽게 사용할 수 있다.
3. 편리한 데이터 전송 : Payload에 사용자 권한 정보나 만료 시간 등을 포함할 수 있어 추가적인 DB 조회 없이도 클라이언트 요청을 처리할 수 있다.

JWT의 단점

1. 크기 : JWT는 자체적으로 사용자 정보를 담고 있기 때문에 일반적인 세션 ID보다 크기가 크다.
2. 보안 : 클라이언트 측에 저장된 토큰이 유출될 경우 유효 기간 동안 악용될 수 있다. 따라서 HTTPS와 같은 안전한 통신 채널을 통해 전송해야 하며 토큰의 만료 시간 설정도 중요하다.
3. 토큰 무효화 어려움 : 서버에서 JWT를 무효화하는 것은 어렵다. 서버는 JWT 자체를 저장하지 않기 때문에 유효 기간 내에서는 무효화할 수 있는 방법이 제한적이다.

보안 고려 사항

1. HTTPS 사용 : JWT는 클라이언트에 저장되기 때문에 토큰을 탈취당하지 않도록 반드시 HTTPS를 통해 통신해야 한다.
2. 토큰 만료 시간 설정 : 짧은 만료 시간을 설정하여 만료 후 새로운 토큰을 발급하는 것이 좋다.
3. 리프레시 토큰 사용 : 만료된 JWT를 갱신하기 위해 별도의 리프레시 토큰을 사용하여 보안을 강화할 수 있다.

이렇게 JWT는 서버 확장성, API 기반 인증에서 매우 유용하지만 보안적인 측면에서 주의해야 할 부분도 많다.

JWT와 OAuth 2.0은 서로 다른 개념이지만 주로 OAuth 2.0에서 JWT를 **Access Token**으로 사용하는 방식으로 함께 사용된다. OAuth 2.0은 인증과 권한 부여를 처리하는 **프로토콜**이고 JWT는 권한을 담은 **토큰 형식**으로 사용되기 때문에 두 가지를 결합하면 효율적이고 확장 가능한 인증 시스템을 만들 수 있다.

OAuth 2.0에서 JWT의 역할

OAuth 2.0에서 클라이언트 애플리케이션이 API에 접근할 수 있도록 **Access Token**을 발급받는다. 이 Access Token을 JWT 형식으로 사용할 수 있다. Access Token에는 권한(scope)과 사용자 정보 등이 포함되어 있으며 이를 통해 자원 서버(API 서버)는 클라이언트가 올바른 권한을 가지고 있는지 확인할 수 있다.

사용자 인증 요청

클라이언트 애플리케이션이 사용자를 대신하여 자원 서버(API 서버)에 접근하려고 할 때 먼저 OAuth 2.0의 **Authorization Server**에 인증 요청을 한다. 이 과정에서 사용자는 로그인 절차를 거친다.

Access Token 발급 (JWT 형식)

사용자가 성공적으로 로그인하면 **Authorization Server**는 클라이언트에게 **Access Token**을 발급한다. 이 Access Token이 JWT 형식으로 발급될 수 있다. JWT는 사용자 정보, 권한 범위(scope), 만료 시간(exp) 등의 정보를 담고 있으며 해당 정보는 서버에서 서명되어 보호된다.

클라이언트가 Access Token을 사용하여 API 호출

클라이언트는 JWT 형식의 Access Token을 가지고 자원 서버(API 서버)에 요청을 보낸다. 이때 요청 헤더에 `Authorization: Bearer <JWT 토큰>` 형식으로 Access Token을 포함시킨다.

자원 서버가 JWT를 검증

자원 서버(API 서버)는 전달된 JWT Access Token의 서명(Signature)을 검증하고 만료 시간(exp), 권한(scope) 등의 정보를 확인한다. JWT가 올바르게 서명되었고 만료되지 않았다면 자원 서버는 클라이언트가 요청한 자원에 대한 접근을 허용한다.

JWT와 OAuth 2.0 사용 시 고려 사항

1. 토큰 유효성 검사 : 자원 서버(API 서버)는 JWT 토큰의 서명 검증을 통해 토큰의 무결성을 확인하고 발급된 토큰이 변조되지 않았는지 확인한다.
2. JWT 만료 시간 : JWT에는 만료 시간이 포함되어 있어 만료된 토큰은 더 이상 유효하지 않는다. 만료 시간이 지나면 사용자는 다시 인증을 받아 새로운 토큰을 발급받아야 한다.
3. 리프레시 토큰 사용 : OAuth 2.0에서는 Access Token이 만료되면 **리프레시 토큰**을 사용하여 새로운 Access Token(JWT)을 발급받을 수 있다. 리프레시 토큰은 주로 OAuth 2.0의 **Authorization Code Flow**에서 사용되며 클라이언트는 리프레시 토큰을 통해 사용자가 다시 로그인하지 않아도 토큰을 갱신할 수 있다.

4. Access Token과 Refresh Token의 구분 : Access Token(JWT)은 만료 시간이 짧고 자원 서버에 전달되지만 리프레시 토큰은 좀 더 긴 만료 시간을 갖고 주로 Authorization Server에서 사용된다.

JWT를 사용하는 OAuth 2.0의 장점

1. 무상태(stateless) : JWT는 서버에 세션 정보를 저장할 필요가 없고 토큰만으로도 인증 및 권한 부여가 가능하므로 서버 확장이 용이하다.
2. 보안성 : JWT는 서명을 통해 변조를 방지할 수 있다. RSA나 HMAC 등의 알고리즘을 사용하여 서명을 검증하면 클라이언트에서 토큰이 변경되지 않았음을 확인할 수 있다.
3. 권한 범위 관리 : JWT의 `scope` 필드를 통해 어떤 자원에 접근할 수 있는지 명확하게 정의할 수 있다.

OAuth 2.0과 JWT를 함께 사용할 때 흐름 요약

1. **클라이언트가 사용자 인증 요청** → 사용자가 인증 서버에서 인증.
2. **Authorization Server가 JWT 형식의 Access Token 발급**.
3. **클라이언트가 자원 서버(API 서버)에 Access Token(JWT)을 전송**.
4. **자원 서버는 JWT의 서명 및 권한 정보 검증 후 요청 처리**.

OAuth 2.0은 권한 부여를 위한 프로토콜이고 **JWT**는 그 권한을 증명하는 데 사용되는 **토큰 포맷**이다. OAuth 2.0에서 JWT는 주로 **Access Token**으로 사용되어 클라이언트가 자원 서버에 안전하게 요청을 보낼 수 있게 한다.
OAuth 2.0과 JWT의 결합은 확장성 있고 효율적이며 무상태적인 인증 및 권한 부여 시스템을 제공한다.