K-Digital Training 웹 풀스택 과정

Socket

TCP/IP

TCP/IP란?

TCP/IP는 컴퓨터 네트워크에서 데이터 통신을 위한 프로토콜 스택으로, 네트워크 간의 데이터 교환을 가능하게 하는 중요한 기술

- 데이터를 분할하여 보냄.
- 정확한 전송을 보장하며 데이터의 경로를 지정하는 역할.

TCP/IP

TCP (Transmission Control Protocol)

TCP는 데이터를 신뢰성 있게 전송하기 위한 프로토콜

특징

- 신뢰성: 데이터의 손실이나 손상을 최소화하고, 데이터의 순서를 보장
- 연결 지향: 데이터를 주고받기 전에 송신자와 수신자 간에 연결
- 흐름 제어: 데이터의 흐름을 제어하여 수신자가 처리할 수 있는 속도에 맞춰 데이터를 전송
- 혼잡 제어: 네트워크의 혼잡 상태를 감지하고 조절하여 네트워크 성능을 유지

TCP/IP

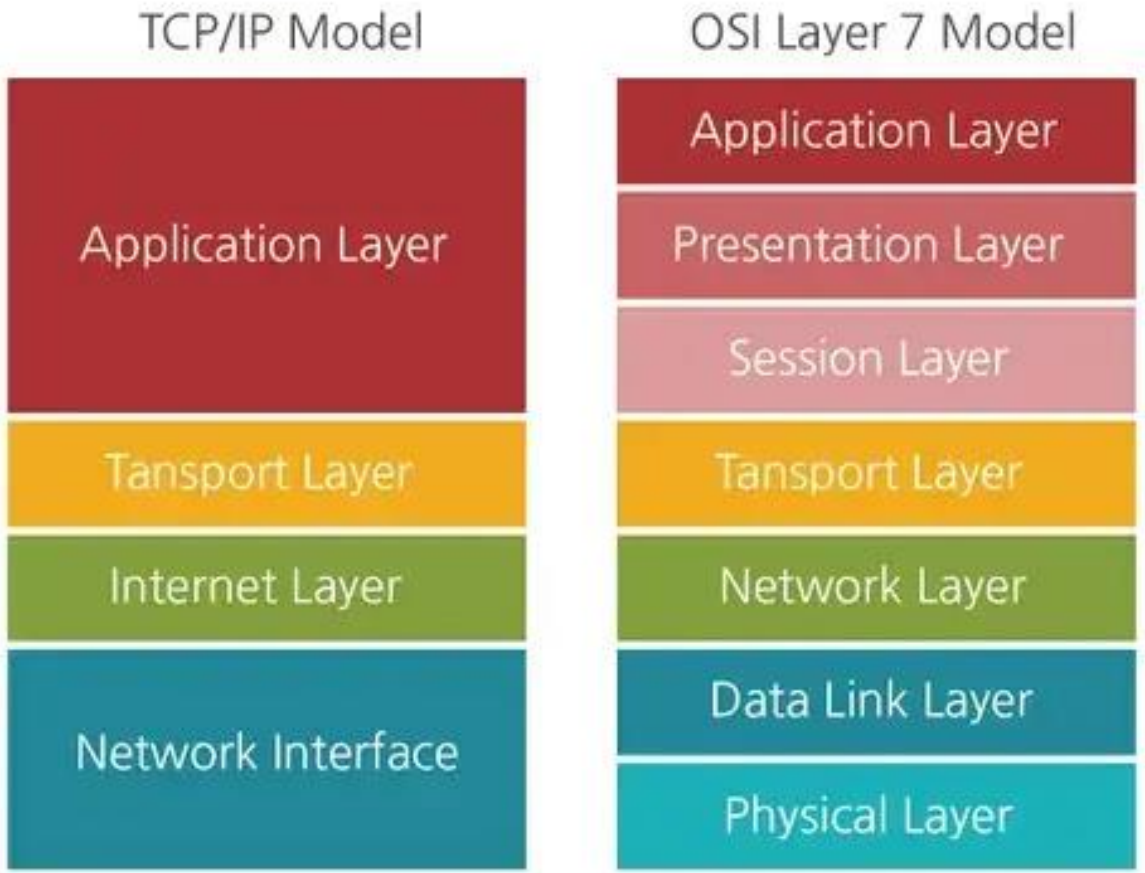
IP (Internet Protocol)

인터넷상에서 데이터를 주고받기 위한 통신 규약(약속)

특징

- 패킷 기반: 데이터를 작은 패킷 단위로 나누어 전송하고, 각 패킷은 목적지 주소와 출발지 주소 정보를 포함
- 비연결성: 패킷은 독립적으로 처리되며, 수신자와의 직접적인 연결이 필요하지 않음.
- 라우팅: 각 라우터가 패킷의 경로를 결정하여 목적지까지 전달
- IP 주소: IP는 각 컴퓨터를 식별하기 위한 IP 주소를 사용

TCP/IP 4계층


TCP/IP Model
Application Layer
Tansport Layer
Internet Layer
Network Interface
OSI Layer 7 Model
Application Layer
Presentation Layer
Session Layer
Tansport Layer
Network Layer
Data Link Layer
Physical Layer

네트워크 인터페이스 계층

Network Interface Layer 또는 Network Access Layer

- OSI 7계층의 물리계층과 데이터 링크 계층에 해당
- 이 계층은 물리적인 네트워크와 상호 작용하며, 데이터를 전기 신호로 변환하거나 광 신호로 변환하여 전송
- 데이터를 프레임으로 나누어 전송하고 프레임을 수신하여 물리적인 신호로 변환하는 역할을 수행
- 수신된 프레임에서 오류를 검출
- MAC (Media Access Control) 주소를 관리
- Ethernet이나 Wi-Fi 같은 기술이 이 계층에 해당

인터넷 계층

Internet Layer

- OSI 7계층의 네트워크 계층에 해당
- 이 계층은 데이터 패킷의 라우팅과 논리적인 주소 지정을 담당
- IP (Internet Protocol) 프로토콜이 이 계층에서 작동하며, 패킷의 출발지와 목적지 IP 주소를 사용하여 라우팅을 수행
- IP 주소와 관련된 서비스인 ARP (Address Resolution Protocol)와 같은 프로토콜도 이 계층에서 동작

전송 계층

Transport Layer

- OSI 7계층의 전송 계층에 해당
- 이 계층은 데이터의 신뢰성과 흐름 제어를 관리
- TCP (Transmission Control Protocol)와 UDP (User Datagram Protocol)가 이 계층에서 작동

응용 계층

Application Layer

- OSI 7계층의 세션 계층, 표현 계층, 응용 계층에 해당
- 최종 사용자에게 서비스를 제공하기 위한 응용 프로그램과 사용자 인터페이스가 이 계층에 위치
- 이 계층은 다양한 프로토콜을 포함하며, HTTP, FTP, SMTP, POP3, IMAP, DNS 등의 프로토콜이 이 계층에서 동작
- 각 프로토콜은 특정한 응용 서비스를 제공하기 위한 목적으로 사용

UDP

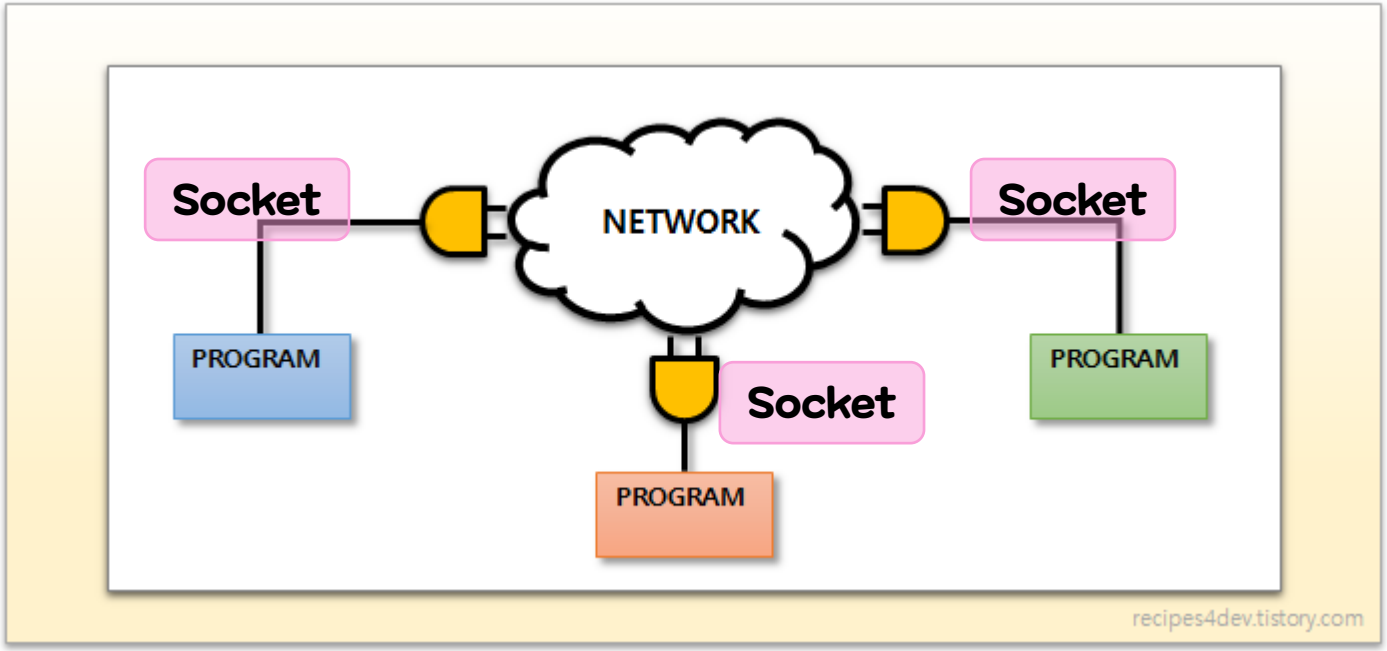
UDP (User Datagram Protocol):

- UDP는 비연결성 프로토콜로, 데이터의 신뢰성은 낮지만 속도가 빠르고 간편
- 데이터그램 단위로 데이터를 전송하며, 순서 보장 및 데이터 신뢰성은 보장되지 않음.
- 데이터 전송에 관련된 작업이 단순
- 흐름 제어나 혼잡 제어 메커니즘이 없어서 오버헤드가 적음
- VOIP, 스트리밍, 온라인 게임 등에서 데이터 전송이 빈번하게 발생하는 경우에 사용됩니다.

소켓

소켓(Socket)

- 프로세스가 네트워크로 **데이터를 내보내거나 데이터를 받기 위한** 실제적인 창구역할을 하는 것



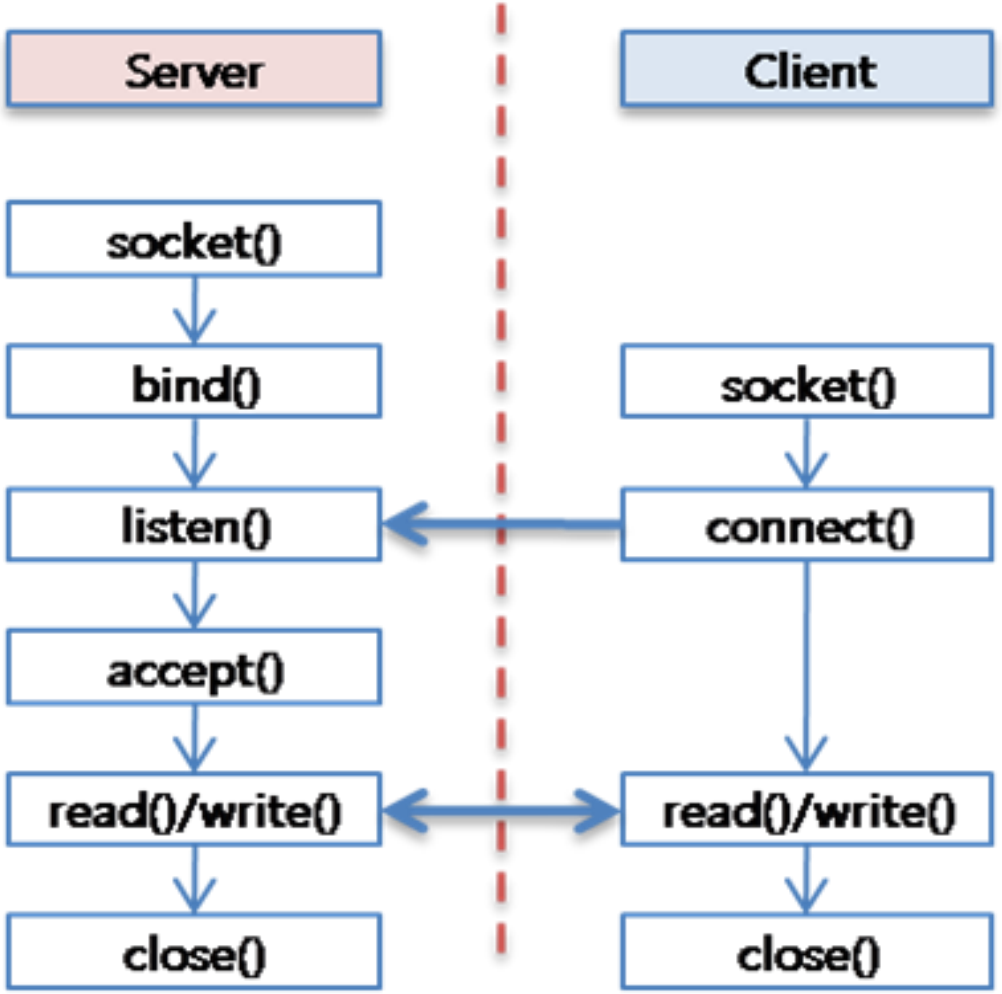
소켓(Socket)

- **서버와 클라이언트를 연결**해주는 도구로써 인터페이스 역할을 하는 것
  - 서버 : 클라이언트 소켓의 연결 요청을 대기하고, 연결 요청이 오면 클라이언트 소켓을 생성해 통신을 가능하게 하는 것
  - 클라이언트 : 실제로 데이터 송수신이 일어나는 곳
- 소켓은 *프로토콜, IP 주소, 포트 넘버로* 정의된다.
- TCP와 UDP 프로토콜을 사용하여 데이터를 전송

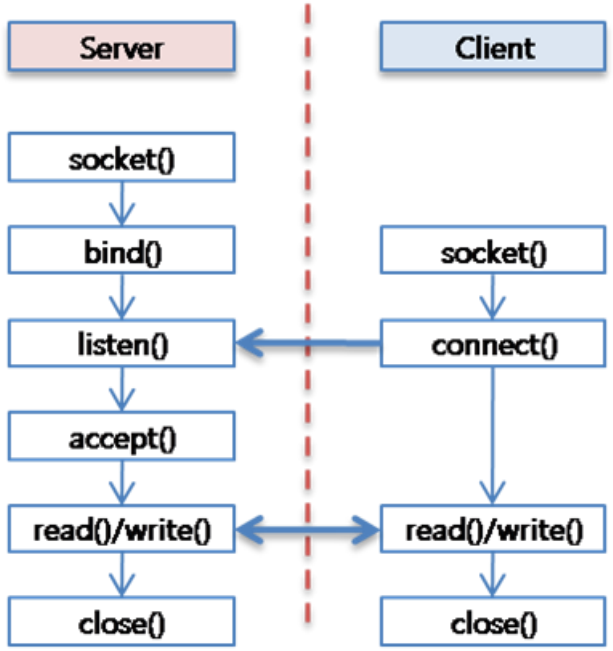
소켓 프로그래밍

- **서버 소켓**:
  - 서버 소켓은 연결 요청을 받아들이는 역할.
  - 클라이언트의 요청을 받아들여 실제 통신을 위한 소켓을 생성.
- **클라이언트 소켓**
  - 클라이언트 소켓은 서버에 연결을 요청하고, 연결이 수락되면 서버와 데이터를 주고받을 수 있는 소켓.
- **포트**
  - 컴퓨터 내에서 소프트웨어 간에 통신을 할 때 사용되는 식별자.
  - 포트를 이용하여 특정 소켓을 찾고 연결.

소켓 프로그래밍 흐름

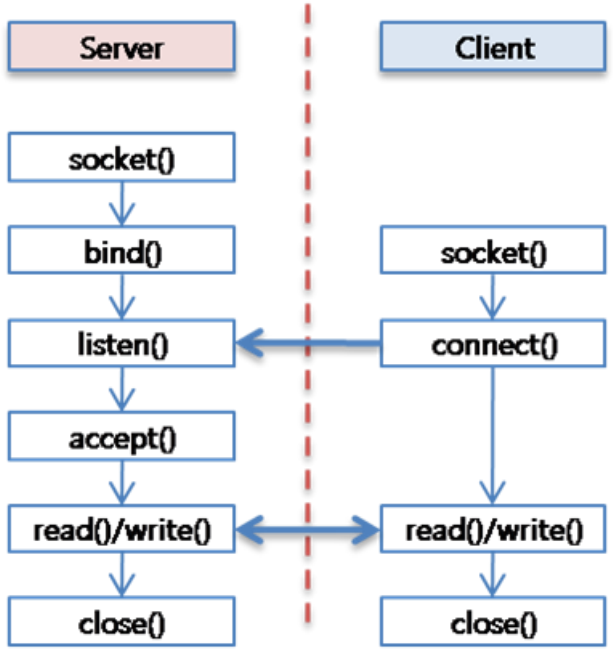

Server
Client
socket()
bind()
listen()
accept()
read()/write()
close()
socket()
connect()
read()/write()
close()

소켓 프로그래밍 흐름



- **서버(Server)**
  - **socket()** : Socket 생성 함수
  - **bind()** : ip와 port 번호 설정 함수
  - **listen()** : 클라이언트의 요청에 수신 대기열을 만드는 함수
  - **accept()** : 클라이언트와의 연결을 기다리는 함수

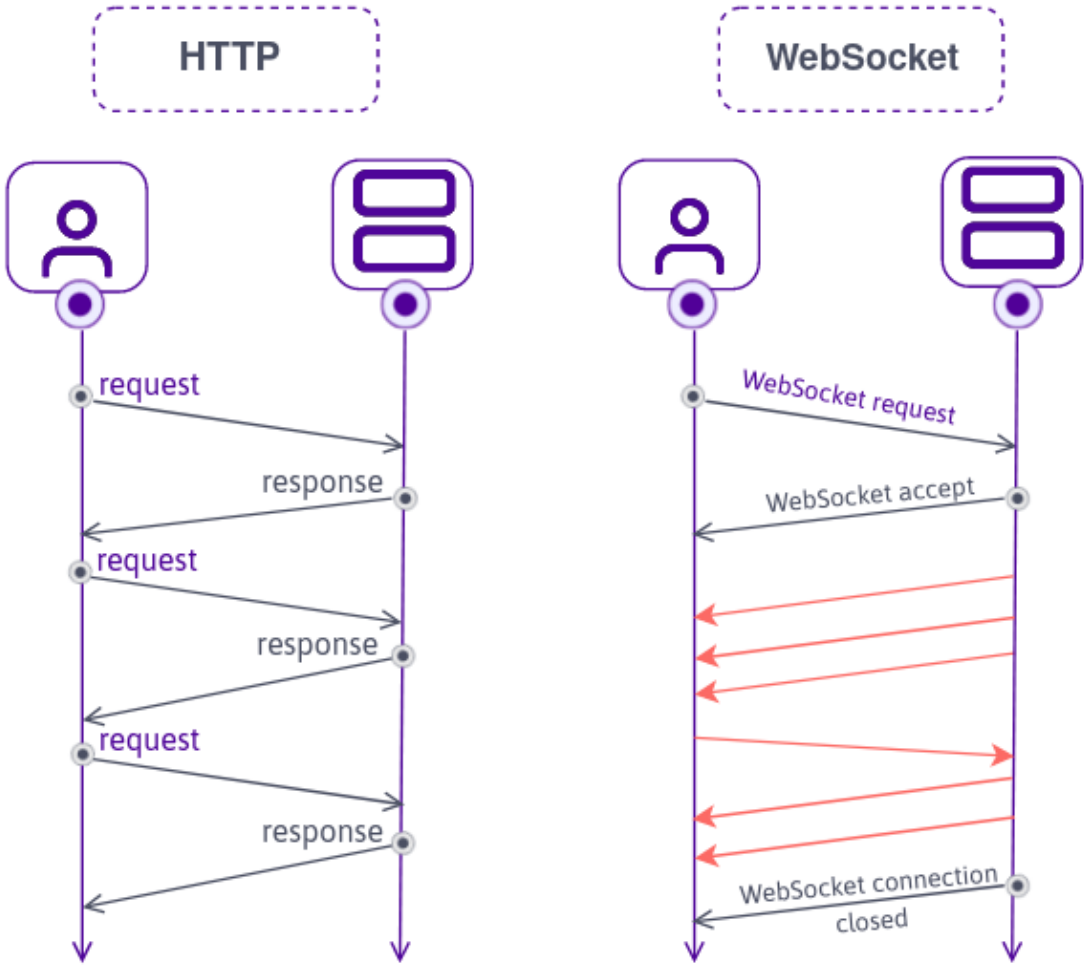
소켓 프로그래밍 흐름



- **클라이언트(client)**
  - **socket()** : 소켓을 여는 함수
  - **connect()** : 통신할 서버의 설정된 ip와 port 번호에 통신을 시도하는 함수
  - 통신 시도 시, 서버가 accept()함수를 이용해 클라이언트의 socket descriptor를 반환
  - 이를 통해 클라이언트와 서버가 서로 read() write()를 반복하며 통신

WebSocket

HTTP vs WebSocket



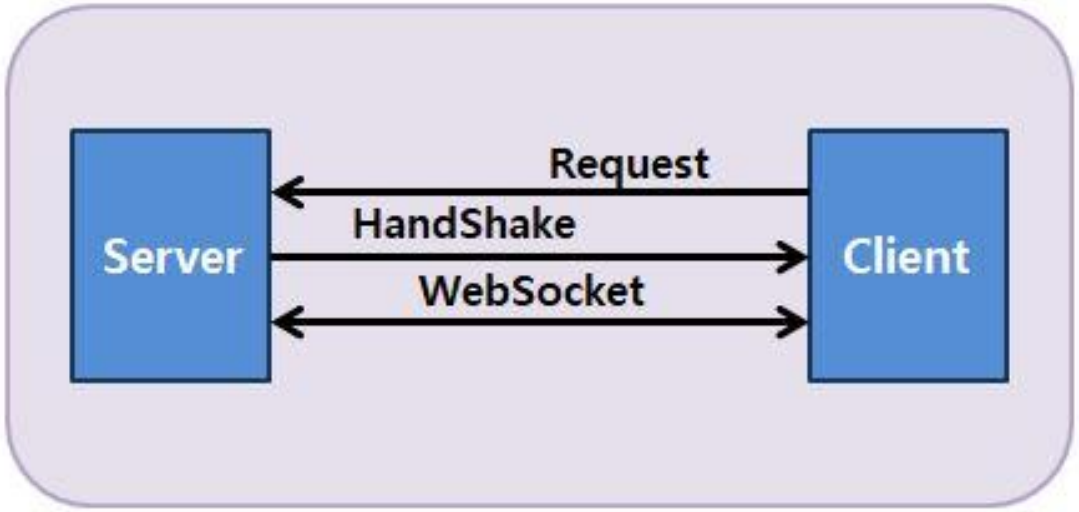
HTTP vs WebSocket

- HTTP 통신
  - 클라이언트의 요청이 있을 때만 서버가 응답하여 해당 정보를 전송하고 곧바로 연결을 종료하는 방식
  - 서버로부터 응답을 받은 후에는 연결이 바로 종료
  - 실시간 연결이 아니고, 필요한 경우에만 서버로 요청을 보내는 상황에 유용

- Socket 통신
  - 서버와 클라이언트가 특정 PORT를 통해 실시간으로 양방향 통신을 하는 방식
  - 실시간으로 데이터를 주고 받는 상황이 필요한 경우에 사용

WebSocket이란?

양방향 소통을 위한 프로토콜(약속)



- HTML5 웹 표준 기술
- 빠르게 작동하며 통신할 때
  아주 적은 데이터를 이용
- 이벤트를 단순히 듣고, 보내는 것만 가능
- Handshake(핸드셰이크): 클라이언트가 서버로 웹소켓 연결을 요청할 때, 서버와 클라이언트 간에 초기 핸드셰이크가 이루어지며 이 핸드셰이크 과정을 통해 웹소켓 연결
- 클라이언트 측에서는 브라우저의 WebSocket 객체를 사용하여 웹소켓 연결을 생성하고 관리

WebSocket 이벤트

- open: 웹소켓 연결이 성공적으로 열렸을 때 발생
- message: 웹소켓을 통해 데이터를 주고받을 때 발생
- error: 웹소켓 연결 중 오류가 발생했을 때 발생. 연결 실패, 통신 오류 등이 해당
- close: 웹소켓 연결이 종료되었을 때 발생

클라이언트


```
<script>
    const socket = new WebSocket("ws://localhost:8000");

    socket.addEventListener("open", (event) => {
        console.log("서버에 연결되었습니다.");
    });

    socket.addEventListener("message", (event) => {
        console.log(`서버로부터 받은 메시지: ${event.data}`);
    });

    socket.addEventListener("error", (event) => {
        console.error("오류가 발생했습니다:", event.error);
    });

    socket.addEventListener("close", (event) => {
        console.log("서버와 연결이 종료되었습니다.");
    });
</script>
```

백엔드

브라우저 환경에서는 WebSocket API를 사용하여 웹소켓 클라이언트를 만들 수 있지만, 서버를 만들려면 별도의 라이브러리나 모듈이 필요

```
npm install ws
```

https://www.npmjs.com/package/ws

ws 모듈 이벤트

- connection: 클라이언트가 웹소켓 서버에 연결되었을 때 발생.
  이 이벤트의 콜백 함수는 새로운 클라이언트 연결마다 실행
- message: 클라이언트로부터 메시지를 받았을 때 발생
- error: 웹소켓 연결 중 오류가 발생했을 때 발생
- close: 클라이언트와의 연결이 종료되었을 때 발생

백엔드


```
const ws = require("ws");
const PORT = 8080;

const wss = new ws.Server({ port: PORT });

wss.on("connection", (socket) => {
    console.log("클라이언트가 연결되었습니다.");

    socket.on("message", (message) => {
        console.log(`클라이언트로부터 받은 메시지: ${message}`);
        // 클라이언트로 응답 메시지 전송
        socket.send(`서버에서 받은 메시지: ${message}`);
    });
    socket.on("error", (error) => {
        console.error("오류가 발생했습니다:", error);
    });
    socket.on("close", () => {
        console.log("클라이언트와 연결이 종료되었습니다.");
    });
});
```