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.