

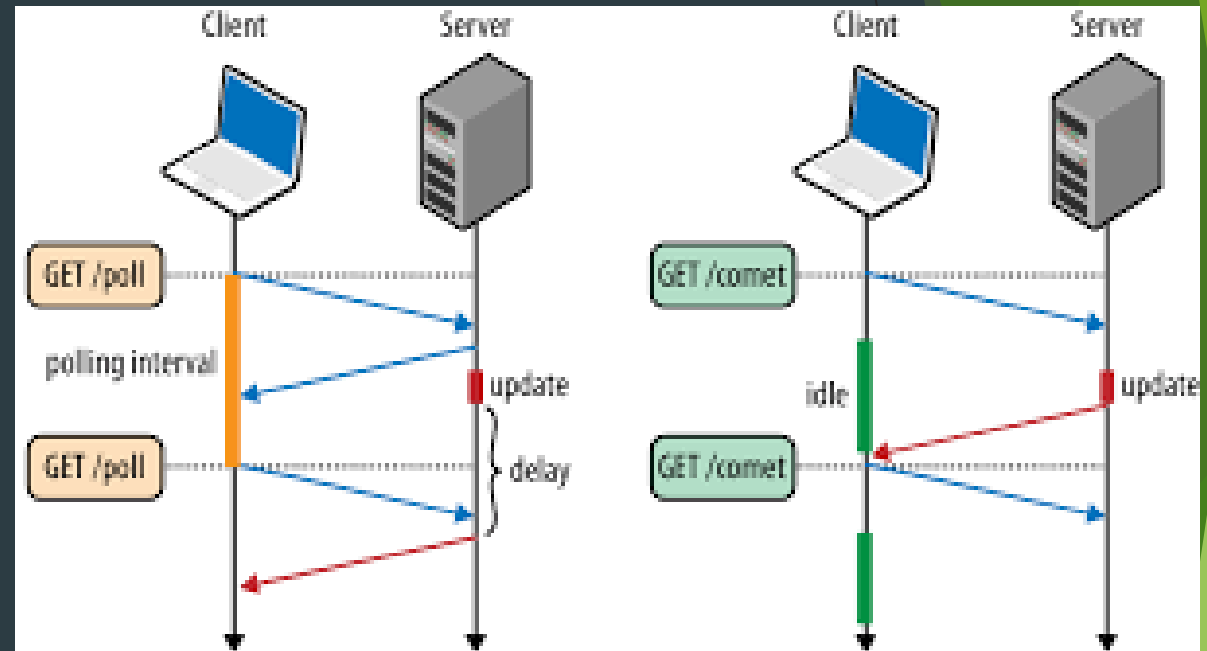
JavaScript

Communication API - 김근형 강사

Communication API

▶ Communication API

- ▶ Communication API 는 HTML5 에 새로 추가된 통신 기법으로 서로 다른 노드에서 제공하는 어플리케이션과 안전하게 통신 할 수 있게 한다.
- ▶ 커뮤니케이션 API는 일반적으로 XMLHttpRequest를 이용하는 Ajax 기술, 다른 노드의 문서 메시징을 지원하는 Cross Document Messaging(또는 Web Messaging) 그리고 서버와의 실시간 통신을 위한 Web Socket, 마지막으로 Server Sent Events (SSE) 라는 기술들을 지칭한다.



XMLHttpRequest

▶ XMLHttpRequest

- ▶ XMLHttpRequest 비 동기로 통신하기 위한 스펙을 가지고 있는 API
- ▶ 1999년 이후 AJAX를 구현하는 핵심 기술
- ▶ IE를 제외한 거의 모든 브라우저에서 제공
- ▶ XMLHttpRequest(XHR) 객체는 서버와 상호작용하기 위하여 사용된다.
- ▶ 전체 페이지의 새로고침 없이도 URL로부터 데이터를 받아올 수 있다. 이는 웹 페이지가 사용자가 하고 있는 것을 방해하지 않으면서 페이지의 일부를 업데이트할 수 있도록 해준다.
- ▶ XMLHttpRequest 는 이름으로만 봐서는 XML 만 받아올 수 있을 것 같아 보이지만, 모든 종류의 데이터를 받아오는데 사용할 수 있다. 또한 HTTP 이외의 프로토콜도 지원한다(file 과 ftp 포함).

XMLHttpRequest

▶ XMLHttpRequest 생성자/속성/함수/이벤트

생성자 이름	설명
XMLHttpRequest	XMLHttpRequest 객체를 생성한다.

속성 이름	설명
readyState	요청의 상태를 0~4 의 정수값으로 표현한다. 내용은 다음과 같다. <ul style="list-style-type: none">- 0 : 아직 open()이 호출되기 이전(UNSENT)- 1 : open()이 호출된 직후(OPENED)- 2 : send()가 호출된 직후로 header 정보 확인 가능(HEADERS_RECEIVED)- 3 : 데이터 받는 중(LOADING)- 4 : 동작 완료(DONE)
response	responseType 속성의 값에 따라 ArrayBuffer, Blob, JSON, 문자열 등 다양한 형태의 응답을 반환한다.
responseText	요청에 대한 응답을 텍스트 형태로 반환한다. 요청이 실패하면 null 이 반환된다.
responseType	응답의 타입으로 “arraybuffer”, “blob”, “document”, “json”, “text” 등을 가질 수 있다. 기본은 “text” 이다.
responseXML	요청에 대한 응답을 XML 형태로 반환한다. 요청이 실패하면 null 이 반환된다.
status	응답의 HTTP 상태 값을 반환한다. 예를 들어 정상적으로 응답을 받은 경우 200 이다.
timeout	서버로부터의 응답을 기다리는 최대 시간이다.
upload	XMLHttpRequestUpload 객체를 반환한다.

XMLHttpRequest

▶ XMLHttpRequest 생성자/속성/함수/이벤트

함수 이름	설명
abort()	이미 전송한 요청을 취소한다.
getAllResponseHeaders()	모든 응답 헤더를 반환한다. 아무런 헤더를 받지 못한 경우 null 이 반환된다.
getResponseHeaders(name)	특정 이름의 응답 헤더를 반환한다. name 에 해당하는 값이 없는 경우 null 이 반환된다.
open(method, url, async [, user, password])	요청을 초기화하는 메서드로 “GET”, “POST”, “PUT”, “DELETE” 중 하나를 사용한다. url 은 요청을 받아서 처리할 서버의 주소이다. async 는 비동기 처리 여부로 동기 처리 시 false , 비동기 처리 시 true 이다. user 와 password 는 옵션으로 인증이 필요한 경우 사용하며 기본은 빈 문자열이다.
overrideMimeType(newType)	기존의 MIME 타입을 새로운 타입으로 변경한다. 이 함수는 send() 함수가 실행되기 전에 호출해야 한다.
send([data])	실제로 서버를 요청하는 메서드이다. 전송할 데이터가 없는 경우 생략하며 필요에 따라 ArrayBufferView , Blob , Document , FormData 문자열을 전송할 수 있다.
setRequestHeader(name, value)	name 의 헤더에 value 를 할당한다. 이 함수는 반드시 open() 과 send() 사이에 호출해야 한다.

XMLHttpRequest

▶ XMLHttpRequest 생성자/속성/함수/이벤트

이벤트 이름	설명
readystatechange	readyState 속성이 변경될 때 발생하는 이벤트이다.
timeout	요청에 대한 응답 시간 초과가 발생했을 때 동작하는 이벤트이다.
loadstart	요청을 처음 시작할 때 발생하는 이벤트이다.
progress	데이터를 전송하거나 로딩할 때 주기적으로 발생한다. 이 이벤트를 통해서 응답의 진행 상태를 확인할 수 있다.
abort	요청이 중단될 때 발생하는 이벤트이다.
error	요청 처리 중에 에러가 일어날 때 발생하는 이벤트이다.
load	요청이 성공적으로 완료되었을 때 발생하는 이벤트이다.
loadend	요청이 성공 또는 실패로 종료되었을 때 발생하는 이벤트이다.

XMLHttpRequest

▶ XMLHttpRequest 예제

onreadystatechange 속성

Ajax 요청 보내기!

XMLHttpRequest 객체의 현재 상태는 OPENED 입니다.
XMLHttpRequest 객체의 현재 상태는 DONE 입니다.

```
<h1>onreadystatechange 속성</h1>
<button type="button" onclick="sendRequest()">Ajax 요청 보내기!</button>
<p id="status"></p>
<p id="text"></p>
<script type="text/javascript">
function sendRequest() {
    var httpRequest = new XMLHttpRequest();
    var currentState = "";
    httpRequest.onreadystatechange = function() {
        switch (httpRequest.readyState) {
            case XMLHttpRequest.UNSET:
                currentState += "XMLHttpRequest 객체의 현재 상태는 UNSET 입니다.<br>";
                break;
            case XMLHttpRequest.OPENED:
                currentState += "XMLHttpRequest 객체의 현재 상태는 OPENED 입니다.<br>";
                break;
            case XMLHttpRequest.HEADERS_RECEIVED:
                currentState += "XMLHttpRequest 객체의 현재 상태는 HEADERS_RECEIVED 입니다.<br>";
                break;
            case XMLHttpRequest.LOADING:
                currentState += "XMLHttpRequest 객체의 현재 상태는 LOADING 입니다.<br>";
                break;
            case XMLHttpRequest.DONE:
                currentState += "XMLHttpRequest 객체의 현재 상태는 DONE 입니다.<br>";
                break;
        }
        document.getElementById("status").innerHTML = currentState;

        if (httpRequest.readyState == XMLHttpRequest.DONE && httpRequest.status == 200) {
            document.getElementById("text").innerHTML = httpRequest.responseText;
        }
    };
    httpRequest.open("GET", "ajax_intro_data.txt", true);
    httpRequest.send();
}
</script>
```

XMLHttpRequest

▶ XMLHttpRequest 실행 전 node.js 세팅

```
> node_modules
> public
v router
  JS router1.js
v views
  <> index.ejs
  <> 01-XHR_readyState1.html
  JS 02_main01.js
  ≡ ajax_intro_data.txt
  {} package.json
  {} package-lock.json
```

```
{
  "name": "communicate",
  "version": "0.0.0",
  "private": true,
  "dependencies": {
    "express": "*",
    "ejs": "*",
    "cookie-parser": "*",
    "express-session": "*"
  }
}
```

```
var express = require('express');
var ejs = require("ejs");

var app = express();

app.set("views", __dirname + "/views");
app.set("view engine", "ejs");
app.engine("ejs", ejs.renderFile);

app.use(express.static("public"));
app.use(express.json());

var router1 = require('./router/router1')(app);

var server = app.listen(2000, function(){
  console.log('포트 2000번으로 서버 실행');
});
```

localhost:2000

index.ejs 페이지입니다.

```
module.exports = function(app){
  app.get("/", function(req,res){
    res.render("index.ejs");
  });
}
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <h1>index.ejs 페이지입니다.</h1>
</body>
</html>
```


XMLHttpRequest

- ▶ send를 이용한 간단한 text 보내기 예제

router1.js

```
app.get("/ajax1", function(req,res){
  res.render("ajax1.ejs");
});

app.get("/ajax1_result1", function(req,res){
  res.send("이것은 xmlhttprequest 를 활용한 비동기 통신입니다.");
});
```

localhost:2000/ajax1

Change Content

이것은 xmlhttprequest 를 활용한 비동기 통신입니다.

ajax1.ejs

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <div id="demo">
    <button type="button" onclick="loadXMLDoc()">Change Content</button>
  </div>
  <script>
    function loadXMLDoc() {
      var xhttp = new XMLHttpRequest();
      xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && (this.status == 200 || this.status == 201)) {
          document.getElementById("demo").innerHTML = this.responseText;
        }
      };
      xhttp.open("GET", "ajax1_result1", true);
      xhttp.send();
    }
  </script>
</body>
</html>
```

XMLHttpRequest

- ▶ xmlhttprequest 에서 json 타입을 받아 처리하는 예제 - ajax2.ejs

```
<body>
  <div>
    <p id="method">method</p>
    <p id="des">description</p>
  </div>
  <button type="button" onclick="loadJsonBy('get')">get 호출</button>
  <button type="button" onclick="loadJsonBy('post')">post 호출</button>
  <button type="button" onclick="loadJsonBy('put')">put 호출</button>
  <button type="button" onclick="loadJsonBy('delete')">delete 호출</button>
  <script>
    var xhttp = new XMLHttpRequest();
    xhttp.responseType = 'json'; // 받고자 하는 데이터의 타입을 설정

    function loadJsonBy(method) {
      xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && (this.status == 200 || this.status == 201)) {
          var data = this.response; // response는 json 형태로 받아온다.
          document.getElementById("method").innerHTML = data.method;
          document.getElementById("des").innerHTML = data.des;
        }
      };
      if(method == "get"){
        xhttp.open("GET", "ajax2_get", true);
        xhttp.send();
      }else if(method == "post"){
        xhttp.open("POST", "ajax2_post", true);
        xhttp.send();
      }else if(method == "put"){
        xhttp.open("PUT", "ajax2_put", true);
        xhttp.send();
      }else if(method == "delete"){
        xhttp.open("DELETE", "ajax2_delete", true);
        xhttp.send();
      }
    }
  </script>
</body>
```

XMLHttpRequest

- ▶ xmlhttprequest 에서 json 타입을 받아 처리하는 예제 - router1.js

method

description

get

이것은 get 호출로 얻은 데이터 입니다.

post

이것은 post 호출로 얻은 데이터 입니다.

put

이것은 put 호출로 얻은 데이터 입니다.

delete

이것은 delete 호출로 얻은 데이터 입니다.

```
app.get("/ajax2", function(req,res){
  res.render("ajax2.ejs");
});

app.get("/ajax2_get", function(req,res){
  var data = {
    method : "get",
    des : "이것은 get 호출로 얻은 데이터 입니다."
  }
  res.json(data);
});

app.post("/ajax2_post", function(req,res){
  var data = {
    method : "post",
    des : "이것은 post 호출로 얻은 데이터 입니다."
  }
  res.json(data);
});

app.put("/ajax2_put", function(req,res){
  var data = {
    method : "put",
    des : "이것은 put 호출로 얻은 데이터 입니다."
  }
  res.json(data);
});

app.delete("/ajax2_delete", function(req,res){
  var data = {
    method : "delete",
    des : "이것은 delete 호출로 얻은 데이터 입니다."
  }
  res.json(data);
});
```

XMLHttpRequest

- ▶ xmlhttprequest 에서 로딩처리 및 데이터를 가져오는 로직 - ajax3..ejs

```
<body>
  <div>
    <p id="state">state</p>
    <p id="method">method</p>
    <p id="des">description</p>
  </div>
  <button type="button" onclick="loadJsonBy()">get 호출</button>
  <script>
    var xhttp = new XMLHttpRequest();
    xhttp.responseType = 'json'; // 받고자 하는 데이터의 타입을 설정

    function loadJsonBy() {
      xhttp.onreadystatechange = function() {
        if (this.readyState == 1 || this.readyState == 2 || this.readyState == 3) {
          document.getElementById("state").innerHTML = '로딩중...';
        } else if (this.readyState == 4 && (this.status == 200 || this.status == 201)) {
          var data = this.response; // response는 json 형태로 받아온다.
          document.getElementById("method").innerHTML = data.method;
          document.getElementById("des").innerHTML = data.des;
        }
      };
      xhttp.open("GET", "ajax3_result1", true);
      xhttp.send();
    }
  </script>
</body>
```

XMLHttpRequest

- ▶ xmlhttprequest 에서 로딩처리 및 데이터를 가져오는 로직 - router1.js

state
method
description

로딩중...
method
description

로딩중...
get
이것은 get 호출로 얻은 데이터 입니다.

```
app.get("/ajax3", function(req,res){
  res.render("ajax3.ejs");
});

app.get("/ajax3_result1", async function(req,res){
  setTimeout(()=>{
    var data = {
      method : "get",
      des : "이것은 get 호출로 얻은 데이터 입니다."
    }
    res.json(data);
  }, 5000);
});
```

XMLHttpRequest

- ▶ xmlhttprequest 에서 데이터를 받아서 가져오는 로직 - router1.js

```
app.get("/ajax4", function(req,res){
    res.render("ajax4.ejs");
});

app.post("/ajax4_result1", async function(req,res){
    var data = {
        id : req.body.id,
        content : req.body.content
    };
    res.json(data);
});
```

XMLHttpRequest

- ▶ xmlhttprequest 에서 데이터를 받아서 가져오는 로직 - ajax4.ejs (1)

```
<body>
  <label>id : <input type="text" id="id" name="id"></label><br>
  <label>content : <input type="text" id="content" name="content"></label><br>
  <button type="button" onclick="loadJsonBy()">전송</button>
  <div>
    <p id="state"></p>
    <p id="outid">id</p>
    <p id="outcontent">content</p>
  </div>
  <script>
    var xhttp = new XMLHttpRequest();
    xhttp.responseType = 'json'; // 받고자 하는 데이터의 타입을 설정
```

XMLHttpRequest

- ▶ xmlhttprequest 에서 데이터를 받아서 가져오는 로직 - ajax4.ejs (2)

id :

content :

로딩완료

abcd

이것은 컨텐츠

```
function loadJsonBy() {  
    xhttp.onreadystatechange = function() {  
        if (this.readyState == 1){  
            document.getElementById("state").innerHTML = '로딩중...';  
            this.setRequestHeader('Content-type', 'application/json');  
        }else if(this.readyState == 2 || this.readyState == 3){  
            document.getElementById("state").innerHTML = '로딩중...';  
        }else if (this.readyState == 4 && (this.status == 200 || this.status == 201)) {  
            var data = this.response; // response는 json 형태로 받아온다.  
            document.getElementById("state").innerHTML = '로딩완료';  
            document.getElementById("outid").innerHTML = data.id;  
            document.getElementById("outcontent").innerHTML = data.content;  
        }  
    };  
    var data = {  
        id : document.getElementById("id").value,  
        content : document.getElementById("content").value  
    }  
    var data_json = JSON.stringify(data);  
    xhttp.open("POST", "ajax4_result1", true);  
    xhttp.send(data_json);  
}  
</script>  
</body>
```


WebSocket

- ▶ WebSocket



WebSocket

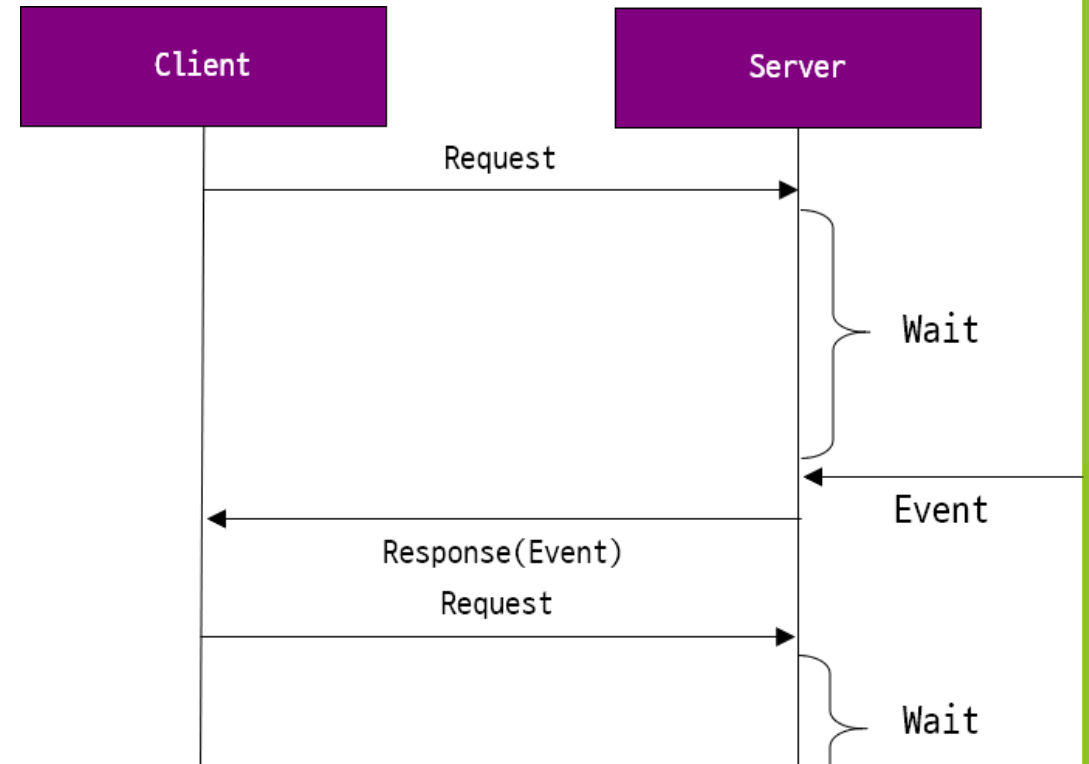
▶ WebSocket

- ▶ 웹소켓은 웹 서버와 클라이언트 간에 지속적인 양방향 통신 방법
- ▶ 즉, 쌍방이 동시에 메시지 데이터를 교환할 수 있다는 의미이며 웹소켓은 진정한 동시성을 제공하고 높은 성능에 최적화 되어 있어서 반응적이고 풍부한 웹 응용 프로그램을 만들 수 있게 해준다.
- ▶ 웹소켓 프로토콜은 IETF(Internet Engineering Task Force)에서 표준화됐고 브라우저의 웹 소켓 API는 W3C에서 표준화가 되었다.

WebSocket

▶ 기존 방식의 문제점 - 폴링(Polling)

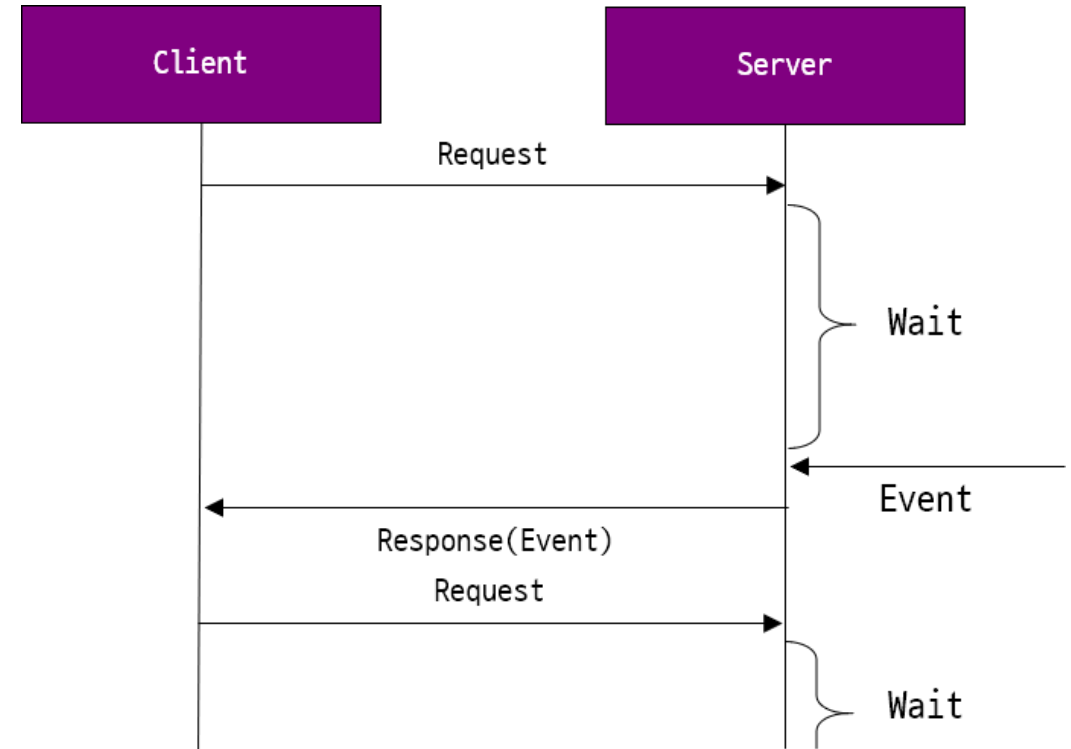
- ▶ 기존 서버와 클라이언트와의 통신 기법 중 하나
- ▶ 전송할 데이터의 유무에 관계없이 주기적으로 요청을 수행하는 동기적 방법
- ▶ 클라이언트는 지정된 시간 간격에 맞춰서 서버에 지속적인 요청을 보낸다.
- ▶ 서버는 각 요청마다 가용 데이터나 데이터가 없는 경우 적절한 경고 메시지로 이에 응답한다.
- ▶ 폴링은 비교적 잘 동작하기는 하지만 대부분의 경우 과도한 연결이 필요하다.
- ▶ 또 웹 응용 프로그램에서 너무 많은 자원을 소비한다



WebSocket

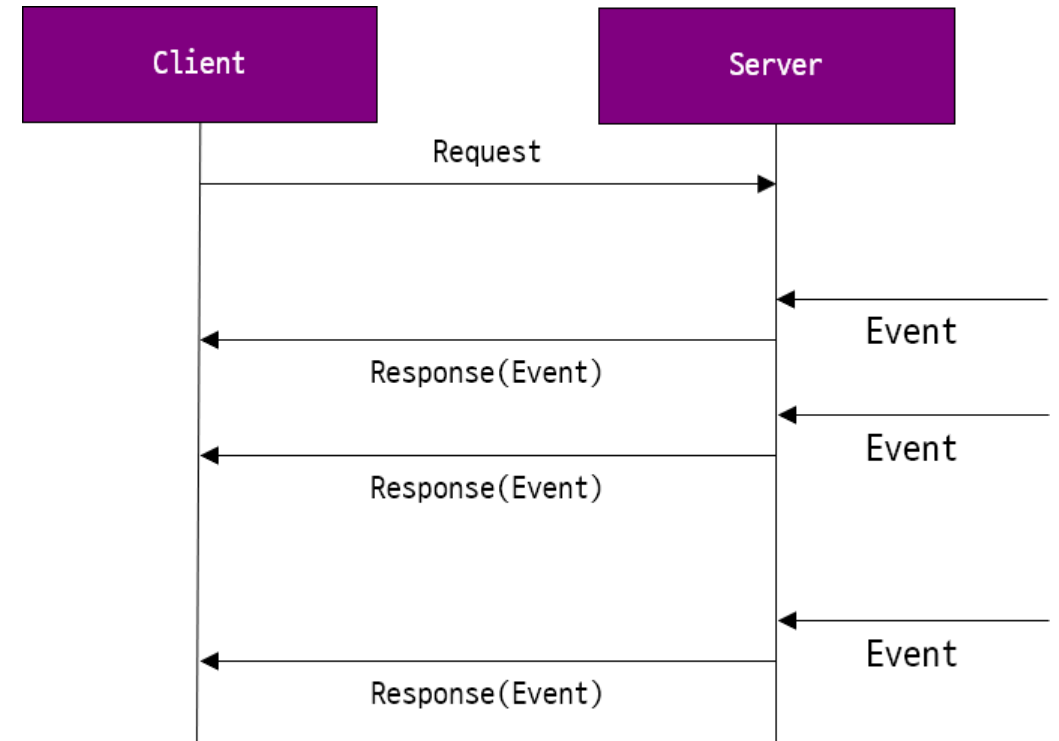
▶ 기존 방식의 문제점 - 롱 폴링(Long Polling)

- ▶ 롱 폴링(long polling)은 폴링과 유사한 방법이지만 클라이언트의 요청에 대해 서버가 전송할 데이터가 있거나 타임아웃이 발생할 때까지 연결을 활성화된 상태로 유지한다.
- ▶ 그런 다음 클라이언트는 다음 단계를 시작하고 순차적으로 요청을 보낸다.
- ▶ 롱 폴링은 폴링보다 성능적으로 향상된 방법이지만 지속적인 요청으로 인해 여전히 프로세스가 늦어질 수 있다.



WebSocket

- ▶ 기존 방식의 문제점 - 스트리밍(Streaming)
 - ▶ 스트리밍은 클라이언트가 요청을 보내면 서버는 무기한 연결을 유지하고 준비가 되면 데이터를 보낸다.
 - ▶ 이전 방법에 비해 크게 개선되기는 했지만, 스트리밍에는 파일 사이즈를 늘리고 불필요한 지연을 야기하는 **HTTP** 헤더가 포함돼 있다.



WebSocket

- ▶ 기존 방식의 문제점 - 포스트 백(postback)과 AJAX
 - ▶ 웹은 **HTTP** 요청/응답 모델을 기반으로 구축됐다. **HTTP**는 **Stateless** 프로토콜이어서 서버와 클라이언트 간의 통신은 각각의 독립적인 요청과 응답의 쌍으로 구성된다.
 - ▶ 클라이언트가 서버에 정보를 요청하면 서버는 적절한 **HTML** 문서로 응답하고 페이지가 새로 고쳐진다. 이러한 방식을 포스트 백 이라 한다.
 - ▶ 사용자가 무엇인가 새로운 행동을 할 때까지 둘 사이에는 아무런 일도 일어나지 않으며 새로운 페이지 로딩은 화면이 깜빡거리는 현상을 수반한다.
 - ▶ 2005년 **AJAX**가 등장하면서 포스트 백의 문제를 해결할 수 있게 됐다.
 - ▶ **AJAX**는 자바 스크립트의 **XmlHttpRequest** 오브젝트로 사용자 인터페이스의 나머지 부분을 방해하지 않고 자바스크립트 코드의 비동기 실행을 가능하게 해준다.
 - ▶ 전체 페이지를 다시 로딩하는 대신 **AJAX**로 웹 페이지의 일부분만 전송할 수 있다.
 - ▶ **AJAX**는 제이쿼리 같은 인기있는 자바스크립트 라이브러리와 함께 사용돼, 최종 사용자 경험을 개선함으로써 모든 웹 사이트에서 반드시 필요한 아이템으로 간주됐다.

WebSocket

- ▶ 기존 방식의 문제점 - 포스트 백(postback)과 AJAX
 - ▶ 앞에 설명한 모든 방법은 실시간 양방향 통신을 제공한다. 하지만 웹소켓에 비하면 세 가지 명백한 단점을 가진다.
 - ▶ http 헤더를 전송해, 전체 파일 크기가 커진다.
 - ▶ 클라이언트/서버 모두 한 쪽이 완료될 때까지 기다려야 하는 반 이중 통신 방식이다.
 - ▶ 웹 서버가 많은 자원을 소모한다.

WebSocket

▶ 웹소켓 프로토콜

- ▶ 웹 소켓 프로토콜은 전 이중 통신을 처음부터 다시 재정의한다.
- ▶ 웹소켓은 웹워커(webworker)와 함께, 데스크톱의 응용프로그램이 제공하는 기능을 웹 브라우저에서도 가능하게 해준다.
- ▶ 동시성과 멀티스레딩은 포스트백 세계에서는 존재하지 않았다. 단지 일부 기능만이 모방돼 제공됐을 뿐이다.
- ▶ 웹 소켓 프로토콜을 통해 위와 같은 동시성과 멀티스레딩이 가능해지며 형식은 다음과 같다.

ws://example.com:8000/chat.php

스키마

호스트

포트

서버

WebSocket

▶ 브라우저 지원

▶ 웹소켓 프로토콜 최신 사양은 **RFC-6455**이며, 모든 최신 웹 브라우저가 이를 지원한다.

- ▶ 인터넷 익스플로러 **10** 이상
- ▶ 모질라 파이어폭스 **11** 이상
- ▶ 구글 크롬 **16** 이상
- ▶ 사파리 **6** 이상
- ▶ 오페라 **12** 이상

WebSocket

▶ 웹 소켓 객체 선언 및 사용 방법

```
new WebSocket(  
    DOMString url  
    [, DOMString protocols]  
);
```

```
new WebSocket(  
    DOMString url  
    [, DOMString[] protocols]  
);
```

websocket 매개변수	설명
url	연결할 URL 주소. 웹소켓 서버가 응답할 수 있는 위치의 주소이어야 한다.
protocol	단일 프로토콜 문자열, 또는 프로토콜 문자열의 배열. 이 문자열들은 서브 프로토콜을 지정하는데 사용된다. 이를 통해 하나의 웹소켓 서버가 여러 개의 웹 소켓 서브 프로토콜을 구현할 수 있도록 해준다. (예를 들어, 하나의 서버가 두 개 이상의 커뮤니케이션 방식을 가지고 싶도록 하고 싶을 때). 만약 지정하지 않으면 빈 문자열을 넣은 것으로 간주된다.

WebSocket

▶ 웹 소켓 속성

속성	타입	설명
binaryType	<u>DOMString</u>	연결에 의해 전송되는 이진 데이터의 유형을 나타내는 문자열. DOM Blob 객체를 사용하는 경우 "blob" 또는 ArrayBuffer 객체를 사용하는 경우 "arraybuffer"여야 한다.
bufferedAmount	unsigned long	send() 에 대한 호출을 사용하여 대기열에 있지만 아직 네트워크로 전송되지 않은 데이터 바이트 수입니다. 연결이 닫혔을 때 이 값은 0으로 재설정되지 않는다. send() 를 계속 호출하면 이 값은 계속 상승한다. 읽기 전용
extensions	<u>DOMString</u>	서버에서 선택한 확장명. 이것은 현재 연결에 의해 협의된 빈 문자열 또는 확장자 목록이다.
onclose	<u>EventListener</u>	WebSocket 인터페이스의 연결상태가 readyState 에서 CLOSED 로 바뀌었을 때 호출되는 이벤트 리스너. 이 이벤트 리스너는 "close" 라는 이름의 <u>CloseEvent</u> 를 받는다.
onerror	<u>EventListener</u>	"error" 라는 이름의 이벤트가 발생하면 처리할 핸들러. 이는 에러가 발생하는 상황에 호출된다.
onmessage	<u>EventListener</u>	"message" 이름의 <u>MessageEvent</u> 이벤트가 발생하면 처리할 핸들러. 이는 서버로부터 메시지가 도착했을 때 호출된다.
onopen	<u>EventListener</u>	WebSocket 인터페이스의 연결상태가 readyState 에서 OPEN 으로 바뀌었을 때 호출되는 이벤트 리스너. 연결 상태가 OPEN 으로 바뀌었다는 말은 데이터를 주고 받을 준비가 되었다는 뜻이다. 이 리스너가 처리하는 이벤트는 "open" 이라는 이벤트 하나이다.
protocol	<u>DOMString</u>	서버에 의해 선택된 서브 프로토콜을 가리킨다. 이 값은 객체를 생성하면서 protocols 파라미터에 전달했던 값 들 중 하나이다.
readyState	unsigned short	연결의 현재 상태. 값은 <u>Ready state constants</u> 중에 하나이다. 읽기 전용.
url	<u>DOMString</u>	생성자에 의해 해석된 URL . 이는 항상 절대 주소를 가리킨다. 읽기 전용.

WebSocket

▶ 웹 소켓 속성(Ready state constants)

Constant	Value	Description
CONNECTING	0	연결이 수립되지 않은 상태.
OPEN	1	연결이 수립되어 데이터가 오고 갈 수 있는 상태.
CLOSING	2	연결이 닫히는 중.
CLOSED	3	연결이 종료되었거나, 연결에 실패한 경우.

WebSocket

▶ 웹 소켓 메서드 - close()

- ▶ 맺어진 연결, 또는 연결 시도를 종료한다. 이미 종료된 경우에는 아무 동작도 하지 않는다.

```
void close(  
    [unsigned short code]  
    [, DOMString reason]  
);
```

매개변수	설명
code	연결이 종료되는 이유를 가리키는 숫자 값. 지정되지 않을 경우 기본값은 1000으로 간주된다.
reason	연결이 왜 종료되는지를 사람이 읽을 수 있도록 나타내는 문자열. 이 문자열은 UTF-8 포맷이며, 123 바이트를 넘을 수 없다.

WebSocket

- ▶ 웹 소켓 메서드 - send()
 - ▶ 웹소켓 연결을 통해 데이터를 서버로 전송한다.

```
void send(  
    DOMString data  
);
```

```
void send(  
    ArrayBuffer data  
);
```

```
void send(  
    Blob data  
);
```

매개변수	설명
data	서버로 전송할 텍스트 메시지

WebSocket

▶ 웹 소켓 예제 -1

```
// Node.js socket server script
const net = require('net');

// Create a server object
const server = net.createServer((socket) => {
  socket.on('data', (data) => {
    console.log(data.toString());
  });
  socket.write('SERVER: 이것은 서버에서 보내는 메시지입니다.<br>');
  socket.end('SERVER: Closing connection now.<br>');
}).on('error', (err) => {
  console.error(err);
});

// 서버 port 4000
server.listen(4000, () => {
  console.log('서버 오픈 포트 : ', server.address().port);
});
```

server

```
--
서버 오픈 포트 : 4000
CLIENT: Hello this is client!
```

```
// Node.js socket client script
const net = require('net');
// Connect to a server @ port 4000
const client = net.createConnection({ port: 4000 }, () => {
  console.log('CLIENT: I connected to the server.');
```

```
  client.write('CLIENT: Hello this is client!');
});
client.on('data', (data) => {
  console.log(data.toString());
  client.end();
});
client.on('end', () => {
  console.log('CLIENT: I disconnected from the server.');
```

```
});
```

client

```
CLIENT: I connected to the server.
SERVER: 이것은 서버에서 보내는 메시지입니다.<br>SERVER: Closing connection now.<br>
CLIENT: I disconnected from the server.
```

WebSocket

- ▶ 웹 소켓 예제 -2 (echo서버 만들기)
 - ▶ 여기에서는 node.js 에서 제공하는 websocket 라이브러리인 ws를 사용한다.
 - ▶ github 주소 : <https://github.com/websockets/ws>

```
// Node.js WebSocket server script
const WebSocket = require('ws');

const wss = new WebSocket.Server({ port: 4000 });

wss.on('connection', function connection(ws) {
  ws.on('message', function incoming(message) {
    console.log('received: %s', message);
  });

  ws.send('Hi this is WebSocket server!');
});
```

received: Hi this is web client.

server

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script>
    const ws = new WebSocket('ws://localhost:4000/');
    ws.onopen = function() {
      console.log('WebSocket Client Connected');
      ws.send('Hi this is web client.');
```

client

```
    };
    ws.onmessage = function(e) {
      console.log("Received: '" + e.data + "'");
    };
  </script>
</body>
</html>
```

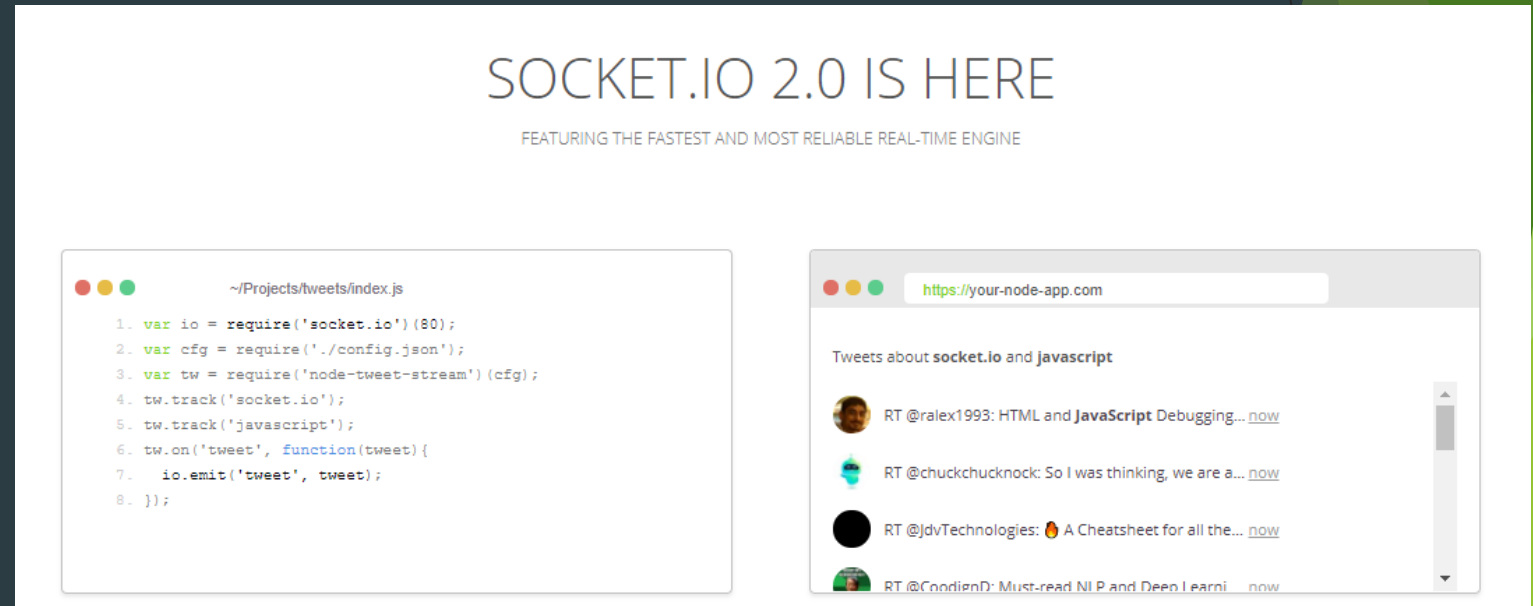
WebSocket Client Connected

Received: 'Hi this is WebSocket server!'

Socket.io

▶ socket.io

- ▶ HTML5 WebSocket은 매우 유용한 기술이지만 오래된 브라우저의 경우 지원하지 않는 경우가 있다.
- ▶ 브라우저 간 호환이나 이전 버전 호환을 고려하여 Node.js를 위한 강력한 Cross-platform WebSocket API인 Socket.io를 사용하는 것이 좋다.
- ▶ <https://socket.io/>



Socket.io

▶ socket.io

▶ 실제 socket.io 관련된 api 자료들은 관련 홈페이지에서 더 자세하게 다루므로 참고할 것

▶ <https://socket.io/docs/>

The screenshot shows the Socket.IO documentation page. On the left is a sidebar with a 'Documentation' section containing links to 'Introduction', 'Server', and 'Client'. The main content area is titled 'What Socket.IO is' and explains that it is a library for real-time, bidirectional communication between a browser and a server. It lists two main components: a Node.js server and a JavaScript client library for the browser. A diagram illustrates this with a box labeled 'Browser / Node.js client' connected by a double-headed arrow to a box labeled 'Node.js server'. Below the diagram, it mentions several community-maintained client implementations in other languages like Java, C++, Swift, Dart, Python, and .Net, each with a corresponding GitHub link.

socket.io

Documentation

- Introduction
 - What Socket.IO is
 - What Socket.IO is not
 - Minimal working example
 - Features
 - Emit cheatsheet
 - Logging and debugging
- Server
 - Installation
 - Initialization
 - Namespaces
 - Rooms
 - Using multiple nodes
- Client
 - Installation
 - Initialization
 - Connection lifecycle
- Internals

What Socket.IO is

Socket.IO is a library that enables real-time, bidirectional and event-based communication between the browser and the server. It consists of:

- a Node.js server: [Source](#) | [API](#)
- a Javascript client library for the browser (which can be also run from Node.js): [Source](#) | [API](#)

Diagram:

```
graph LR; A["Browser / Node.js client"] <--> B["Node.js server"]
```

There are also several client implementation in other languages, which are maintained by the community:

- Java: <https://github.com/socketio/socket.io-client-java>
- C++: <https://github.com/socketio/socket.io-client-cpp>
- Swift: <https://github.com/socketio/socket.io-client-swift>
- Dart: <https://github.com/rikulo/socket.io-client-dart>
- Python: <https://github.com/miguelgrinberg/python-socketio>
- .Net: <https://github.com/Quobject/SocketIoClientDotNet>

Socket.io

▶ socket.io 예제 - Server1

```
var app = require('express')();
var server = require('http').createServer(app);
// http server를 socket.io server로 upgrade한다
var io = require('socket.io')(server);

// localhost:3000으로 서버에 접속하면 클라이언트로 index.html을 전송한다
app.get('/', function(req, res) {
  res.sendFile(__dirname + '/05_mainclient03.html');
});

// connection event handler
// connection이 수립되면 event handler function의 인자로 socket인 들어온다
io.on('connection', function(socket) {

  // 접속한 클라이언트의 정보가 수신되면
  socket.on('login', function(data) {
    console.log('Client logged-in:\n name:' + data.name + '\n userid: ' + data.userid);

    // socket에 클라이언트 정보를 저장한다
    socket.name = data.name;
    socket.userid = data.userid;

    // 접속된 모든 클라이언트에게 메시지를 전송한다
    io.emit('login', data.name );
  });
});
```

```
// 클라이언트로부터의 메시지가 수신되면
socket.on('chat', function(data) {
  console.log('Message from %s: %s', socket.name, data.msg);

  var msg = {
    from: {
      name: socket.name,
      userid: socket.userid
    },
    msg: data.msg
  };
});
```

```
// 메시지를 전송한 클라이언트를 제외한 모든 클라이언트에게 메시지를 전송한다
// socket.broadcast.emit('chat', msg);
```

```
// 메시지를 전송한 클라이언트에게만 메시지를 전송한다
// socket.emit('s2c chat', msg);
```

```
// 접속된 모든 클라이언트에게 메시지를 전송한다
io.emit('chat', msg);
```

```
// 특정 클라이언트에게만 메시지를 전송한다
// io.to(id).emit('s2c chat', data);
});
```

```
// force client disconnect from server
socket.on('forceDisconnect', function() {
  socket.disconnect();
})
```

```
socket.on('disconnect', function() {
  console.log('user disconnected: ' + socket.name);
});
```

```
server.listen(4000, function() {
  console.log('Socket IO server listening on port 4000');
});
```

Socket.io

▶ socket.io 예제 - Client1_1

```
<body>
  <div class="container">
    <h3>Socket.io Chat Example</h3>
    <div class="form-inline">
      <div class="form-group">
        <label for="msgForm">Message: </label>
        <input type="text" class="form-control" id="msgForm">
      </div>
      <button type="button" id="sendMsg" class="btn btn-primary">Send</button>
    </div>
    <div id="chatLogs"></div>
  </div>
  <script src="/socket.io/socket.io.js"></script>
  <script>
    // socket.io 서버에 접속한다
    var socket = io();

    // 서버로 자신의 정보를 전송한다.
    socket.emit("login", {
      name: "kgh",
      userid: "kgh@gmail.com"
    });

    // 서버로부터의 메시지가 수신되면
    socket.on("login", function(data) {
      let block1 = document.createElement("div");
      let block2 = document.createElement("strong");
      let text1 = document.createTextNode(data );
      let text2 = document.createTextNode("님이 입장하셨습니다");
      block2.appendChild(text1);
      block1.appendChild(block2);
      block1.appendChild(text2);

      document.getElementById("chatLogs").appendChild(block1);
    });
  </script>
</body>
```

Socket.io

▶ socket.io 예제 - Client1_2

Socket.io Chat Example

Message:

kgh님이 입장하셨습니다

kgh:안녕하세요

kgh:이번 시간은 websocket에 대해 공부해 봅시다

```
// 서버로부터의 메시지가 수신되면
socket.on("chat", function(data) {
    var block1 = document.createElement("div");
    var block2 = document.createElement("strong");
    var text1 = document.createTextNode(data.from.name+": " );
    var text2 = document.createTextNode(data.msg);
    block2.appendChild(text1);
    block1.appendChild(block2);
    block1.appendChild(text2);

    document.getElementById("chatLogs").appendChild(block1);
});

// 버튼 클릭 시 서버로 메시지 전송
document.getElementById("sendMsg").onclick = () =>{
    let message = document.getElementById("msgForm").value;
    socket.emit("chat", { msg: message });
    document.getElementById("msgForm").value = "";
}

// input 에서 enter시 서버로 메시지 전송
document.getElementById("msgForm").addEventListener('keydown', function(event) {
    if (event.keyCode === 13) {
        event.preventDefault();
        let message = document.getElementById("msgForm").value;
        socket.emit("chat", { msg: message });
        document.getElementById("msgForm").value = "";
    }
}, true);
</script>
</body>
```

Socket.io

▶ socket.io 예제 - Server2

```
var app = require('express')();
var server = require('http').createServer(app);
// http server를 socket.io server로 upgrade한다
var io = require('socket.io')(server);

// localhost:3000으로 서버에 접속하면 클라이언트로 index.html을 전송한다
app.get('/', function(req, res) {
  res.sendFile(__dirname + '/06_mainclient04.html');
});

// namespace /chat에 접속한다.
var chat = io.of('/chat').on('connection', function(socket) {
  socket.on('chat message', function(data){
    console.log('message from client: ', data);

    var name = socket.name = data.name;
    var room = socket.room = data.room;

    // room에 join한다
    socket.join(room);
    // room에 join되어 있는 클라이언트에게 메시지를 전송한다
    var send_data = {
      name : name,
      msg : data.msg
    };

    chat.to(room).emit('chat message', send_data);
  });
});

server.listen(3000, function() {
  console.log('Socket IO server listening on port 3000');
});
```

Socket.io

▶ socket.io 예제 - Client2_1

```
<body>
  <div class="container">
    <h3>Socket.io Chat Example</h3>
    <!-- <form class="form-inline"> -->
    <form id="form" class="form-horizontal">
      <div class="form-group">
        <label for="name" class="col-sm-2 control-label">Name</label>
        <div class="col-sm-10">
          <input type="text" class="form-control" id="name" placeholder="Name">
        </div>
      </div>
      <div class="form-group">
        <label for="room" class="col-sm-2 control-label">Room</label>
        <div class="col-sm-10">
          <input type="text" class="form-control" id="room" placeholder="Room">
        </div>
      </div>
      <div class="form-group">
        <label for="msg" class="col-sm-2 control-label">Message</label>
        <div class="col-sm-10">
          <input type="text" class="form-control" id="msg" placeholder="Message">
        </div>
      </div>
      <div class="form-group">
        <div class="col-sm-offset-2 col-sm-10">
          <button type="submit" class="btn btn-default">Send</button>
        </div>
      </div>
    </form>
    <ul id="chat"></ul>
  </div>
  <script src="/socket.io/socket.io.js"></script>
```

Socket.io

▶ socket.io 예제 - Client2_2

Socket.io Chat Example

Name	<input type="text" value="김근형"/>
Room	<input type="text" value="new"/>
Message	<input type="text" value="이번시간에는 socket.io의 활용법에 대해 배워볼겁니다"/>
	<input type="button" value="Send"/>

- 김근형:안녕하세요
- 김근형:이번시간에는 socket.io의 활용법에 대해 배워볼겁니다

```
<script src="/socket.io/socket.io.js"></script>
<script>
  // 지정 namespace로 접속한다
  var chat = io('http://localhost:3000/chat');

  const form = document.getElementById('form');
  form.addEventListener('submit', function(e) {
    e.preventDefault();

    // 서버로 자신의 정보를 전송한다.
    chat.emit("chat message", {
      name: document.getElementById("name").value,
      room: document.getElementById("room").value,
      msg: document.getElementById("msg").value
    });
  });

  // 서버로부터의 메시지가 수신되면
  chat.on("chat message", function(data) {
    var block1 = document.createElement("li");
    var text1 = document.createTextNode(data.name + ":" + data.msg);
    block1.appendChild(text1);
    document.getElementById("chat").appendChild(block1);
  });
</script>
```


SSE(Server Sent Event)

- ▶ SSE(Server Sent Event)



Server Sent Events

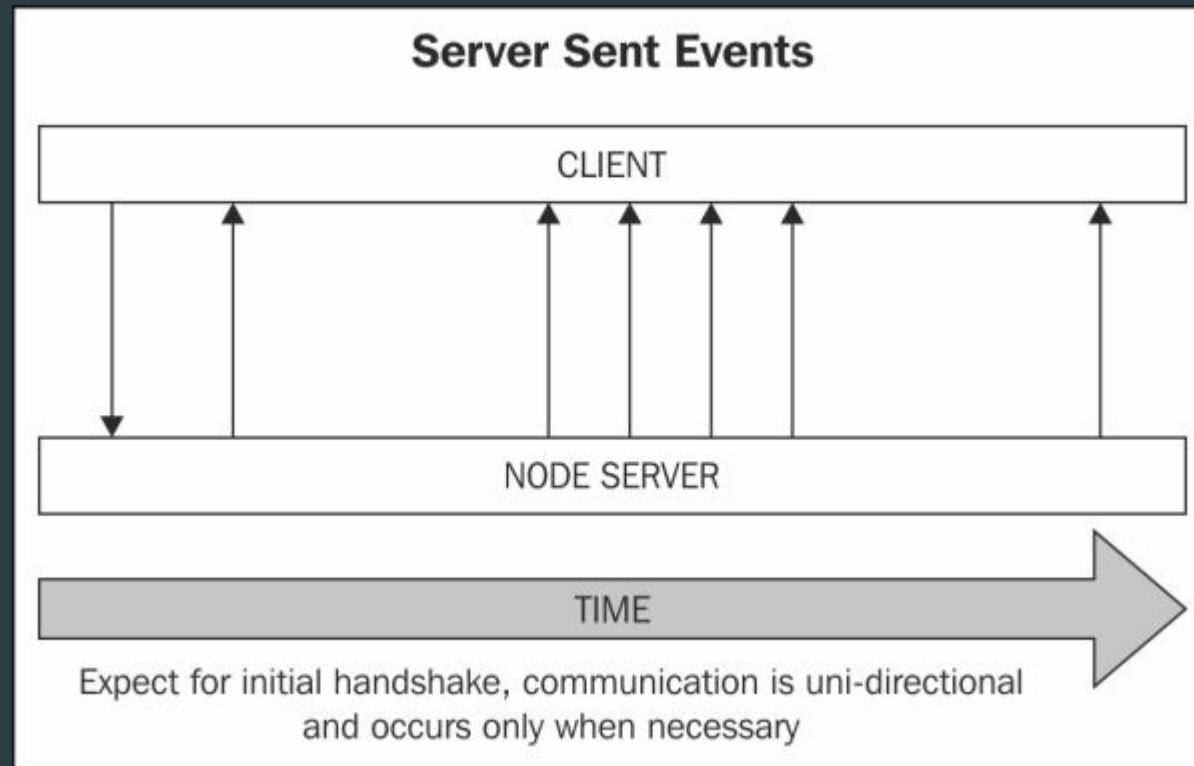
SSE(Server Sent Event)

▶ SSE(Server Sent Event)

- ▶ SSE 는 클라이언트가 HTTP 연결을 통해 서버에서 자동 업데이트를 받을 수 있게하는 서버 푸시 기술
- ▶ 서버 전송 이벤트 EventSource API는 W3C에 의해 HTML5의 일부로 표준화가 됨.
- ▶ Server-Sent Events는 초기 클라이언트 연결이 설정되면 서버가 클라이언트로 데이터 전송을 시작하는 방법을 설명하는 표준이다.
- ▶ 일반적으로 메시지 업데이트 또는 연속 데이터 스트림을 브라우저 클라이언트에 보내는 데 사용되며 클라이언트가 이벤트 스트림을 수신하기 위해 특정 URL을 요청하는 EventSource라는 JavaScript API를 통해 기본 브라우저 간 스트리밍을 향상 시키도록 설계되었다.
- ▶ 기존의 websocket 에서 제공하는 프로토콜을 사용해야 하는 것이 아닌 기존의 프로토콜을 그대로 사용할 수 있다는 장점이 존재한다.

SSE(Server Sent Event)

▶ SSE(Server Sent Event) 동작 원리



SSE(Server Sent Event)

▶ SSE(Server Sent Event) 예제 - 1(server)

```
const express = require('express')
const app = express()

app.get('/', function(req, res) {
  res.sendFile(__dirname + '/07_sseclient01.html');
});

// sse 관련 헤더로 설정
app.get('/countdown', function(req, res) {
  res.writeHead(200, {
    'Content-Type': 'text/event-stream',
    'Cache-Control': 'no-cache',
    'Connection': 'keep-alive'
  });
  countdown(res, 10)
})

// 전송되는 count가 10부터 1씩 줄어드는 로직
function countdown(res, count) {
  res.write("data: " + JSON.stringify({id : 'kgh', cnt : count}) + "\n\n");
  if (count)
    setTimeout(() => countdown(res, count-1), 1000);
  else
    res.end();
}

app.listen(3000, () => console.log('SSE app listening on port 3000!'));
```

SSE(Server Sent Event)

▶ SSE(Server Sent Event) 예제 - 1(client)

```
<body>
  <h1>SSE: <span id="state"></span></h1>
  <h3>Id: <span id="id"></span></h3>
  <h3>Data: <span id="data"></span></h3>
  <script>
    // /countdown으로 sse 연결을 요청한다.
    var source = new EventSource('/countdown');

    // 데이터를 받아 처리하는 부분
    source.addEventListener('message', function(e) {
      var data = JSON.parse(e.data);
      document.getElementById('id').innerHTML = data.id;
      document.getElementById('data').innerHTML = data.cnt;
    }, false);

    // 이벤트 관련 처리 부분
    source.addEventListener('open', function(e) {
      document.getElementById('state').innerHTML = "Connected";
    }, false);

    source.addEventListener('error', function(e) {
      const id_state = document.getElementById('state');
      if (e.eventPhase == EventSource.CLOSED)
        source.close();
      if (e.target.readyState == EventSource.CLOSED) {
        id_state.innerHTML = "Disconnected"
      }
      else if (e.target.readyState == EventSource.CONNECTING) {
        id_state.innerHTML = "Connecting..."
      }
    }, false);
  </script>
</body>
```