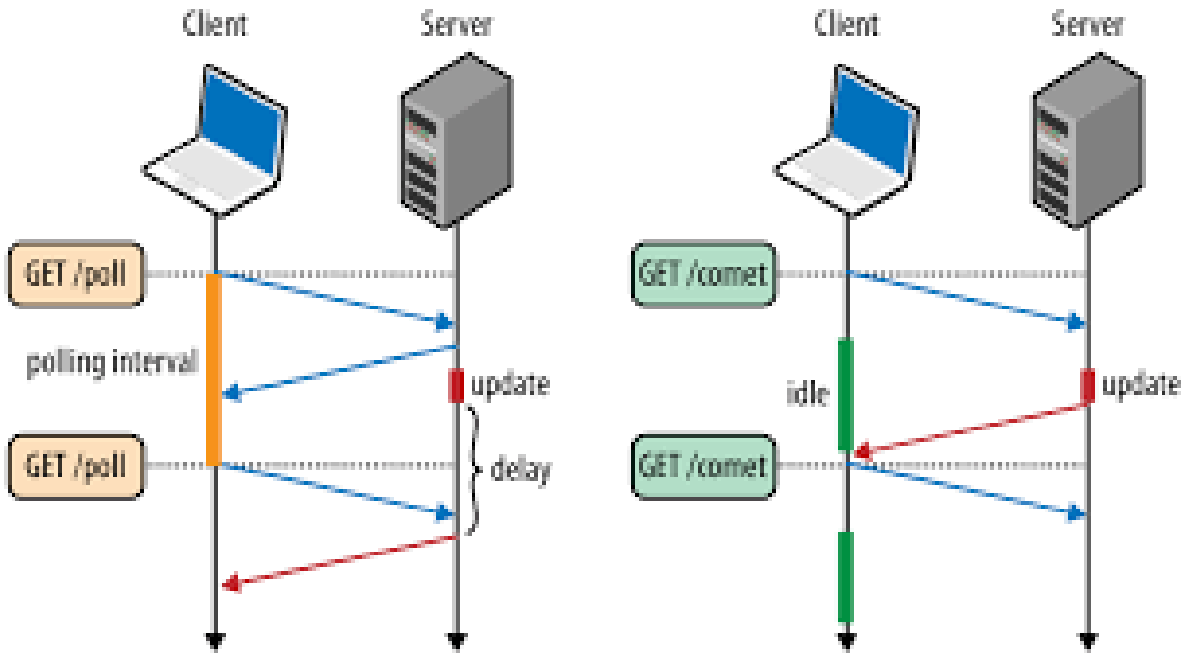
JavaScript

Communication API – 김근형 강사

Communication API

- Communication API
  - Communication API 는 HTML5 에 새로 추가된 통신 기법으로 서로 다른 노드에서 제공하는 어플리케이션과 안전하게 통신할 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 URL 로부터 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. - 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 null이 반환된다.
getResponseHeaders(name)	특정 이름의 응답 헤더를 반환한다. name에 해당하는 값이 없는 경우 null이 반환된다.
open(method, url, async [, user, password])	요청을 초기화하는 메서드로 "GET", "POST", "PUT", "DELETE" 중 하나를 사용한다. url은 요청을 받아서 처리할 서버의 주소이다. async는 비동기 처리 여부로 동기 처리 시 false, 비동기 처리 시 true 이다. user와 password는 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)	name의 헤더에 value를 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.

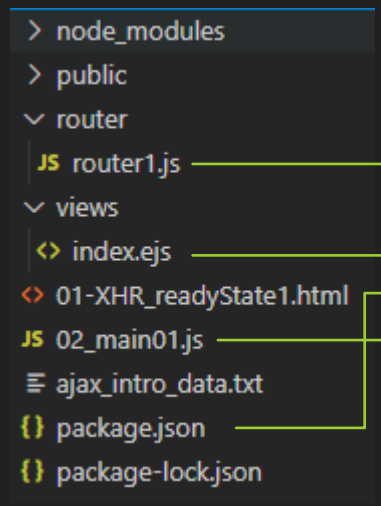
XMLHttpRequest

- XMLHttpRequest 예제


```
<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




```
{
    "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');
});
```

```
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>
```

localhost:2000

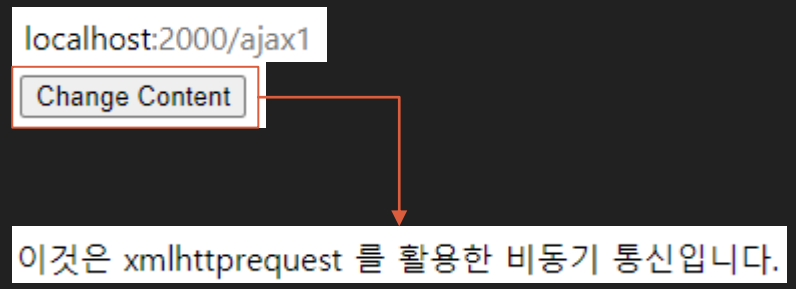
index.ejs 페이지입니다.

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.");
});
```



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>
```

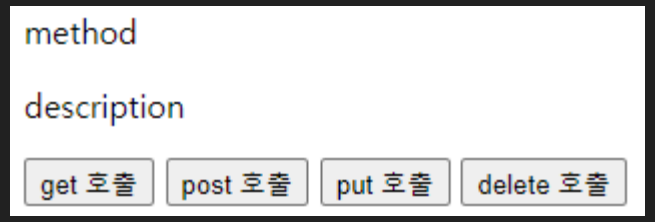
XMLHttpRequest

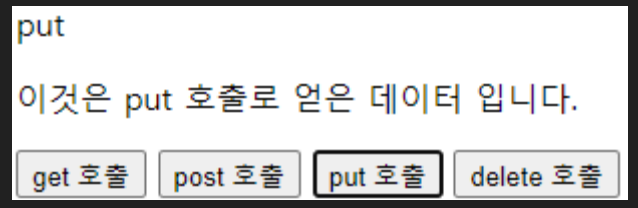
- xmlhttprequest 에서 json 타입을 받아 처리하는 예제 – ajax2.ejs

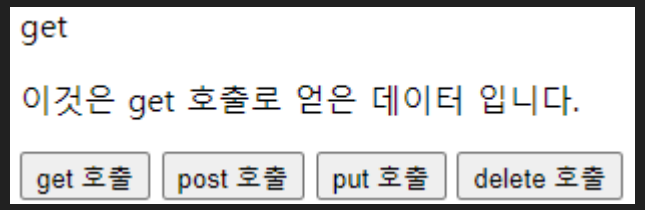
```
<body>
    <div>
        <p id="method">method</p>
        <p id="des">description</p>
    </div>
    <button type="button" onclick="loadJsonBy('get')">get 호출</button>
    <button type="button" onclick="loadJsonBy('post')">post 호출</button>
    <button type="button" onclick="loadJsonBy('put')">put 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>
```

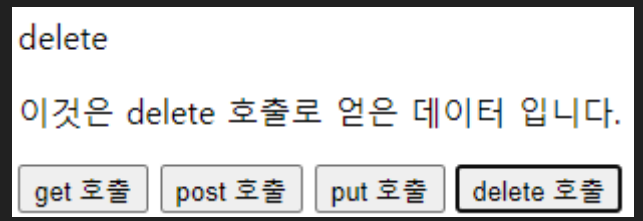
XMLHttpRequest

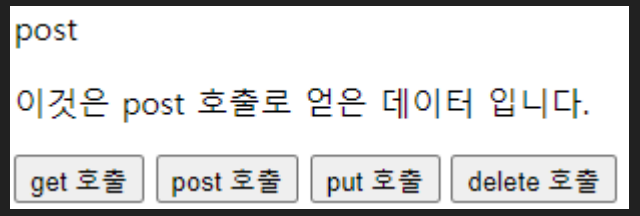
- xmlhttprequest 에서 json 타입을 받아 처리하는 예제 – router1.js












```
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);
});
```

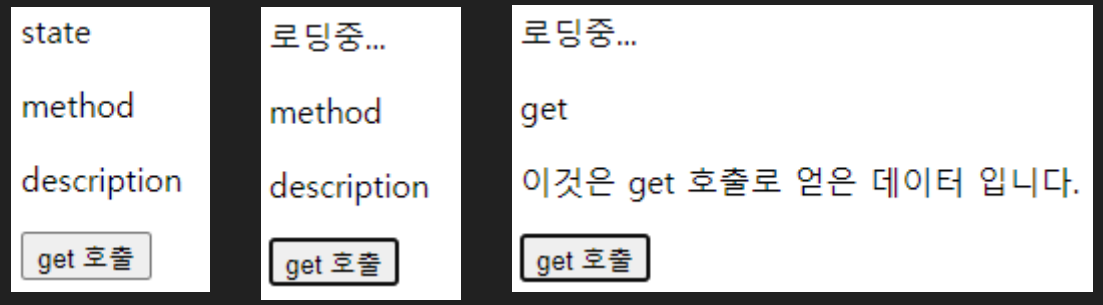
XMLHttpRequest

- 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




```
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)


```
        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

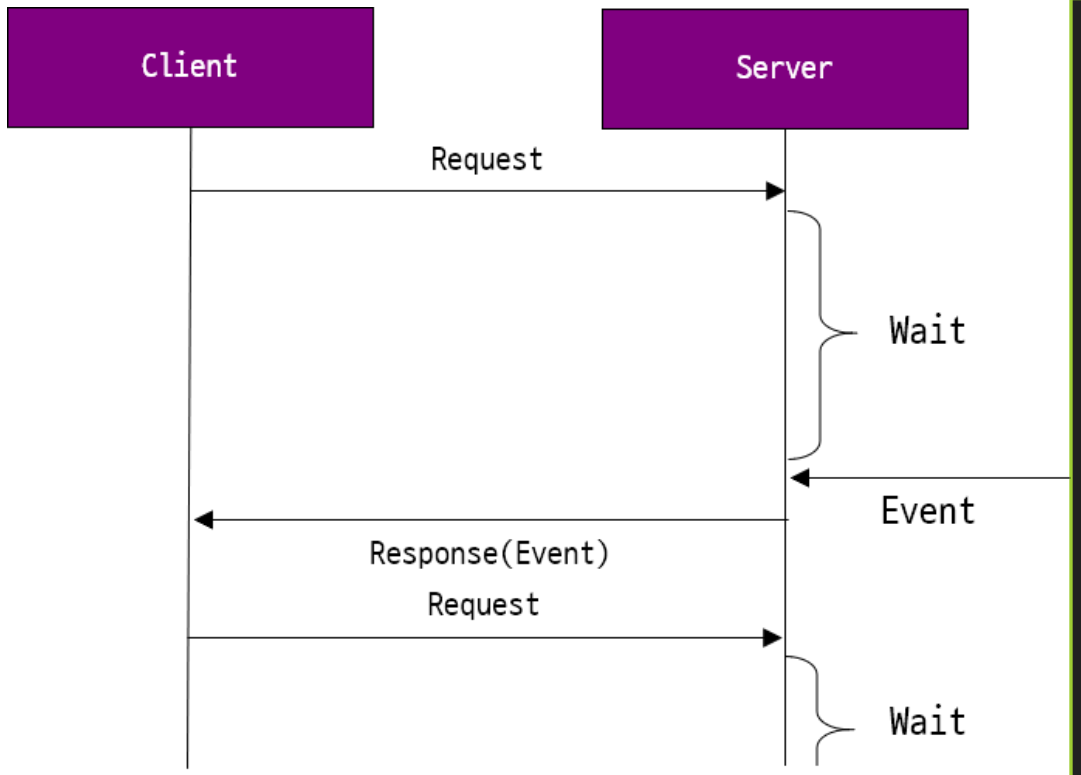
- WebSocket

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.

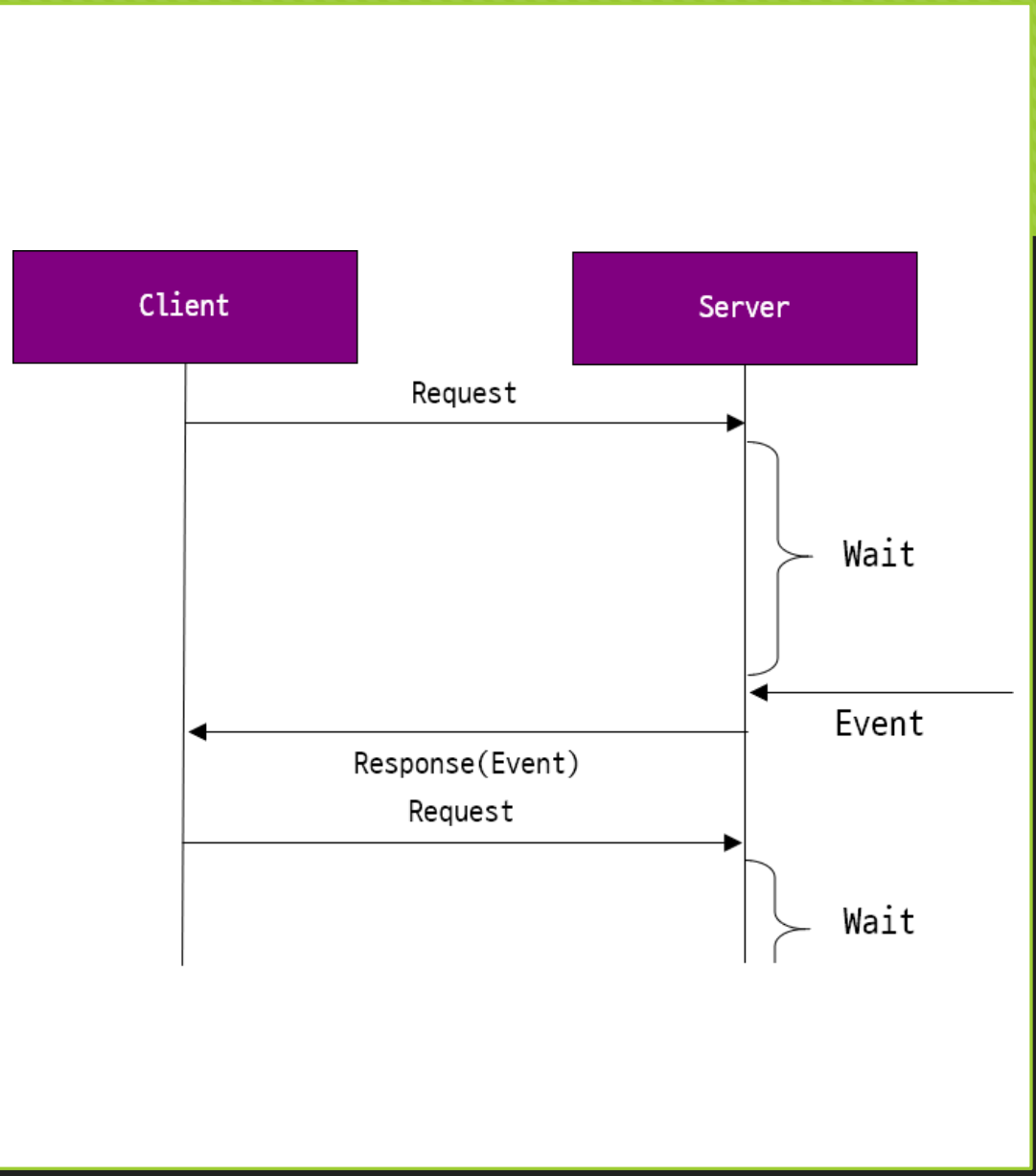
WebSocket

- 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



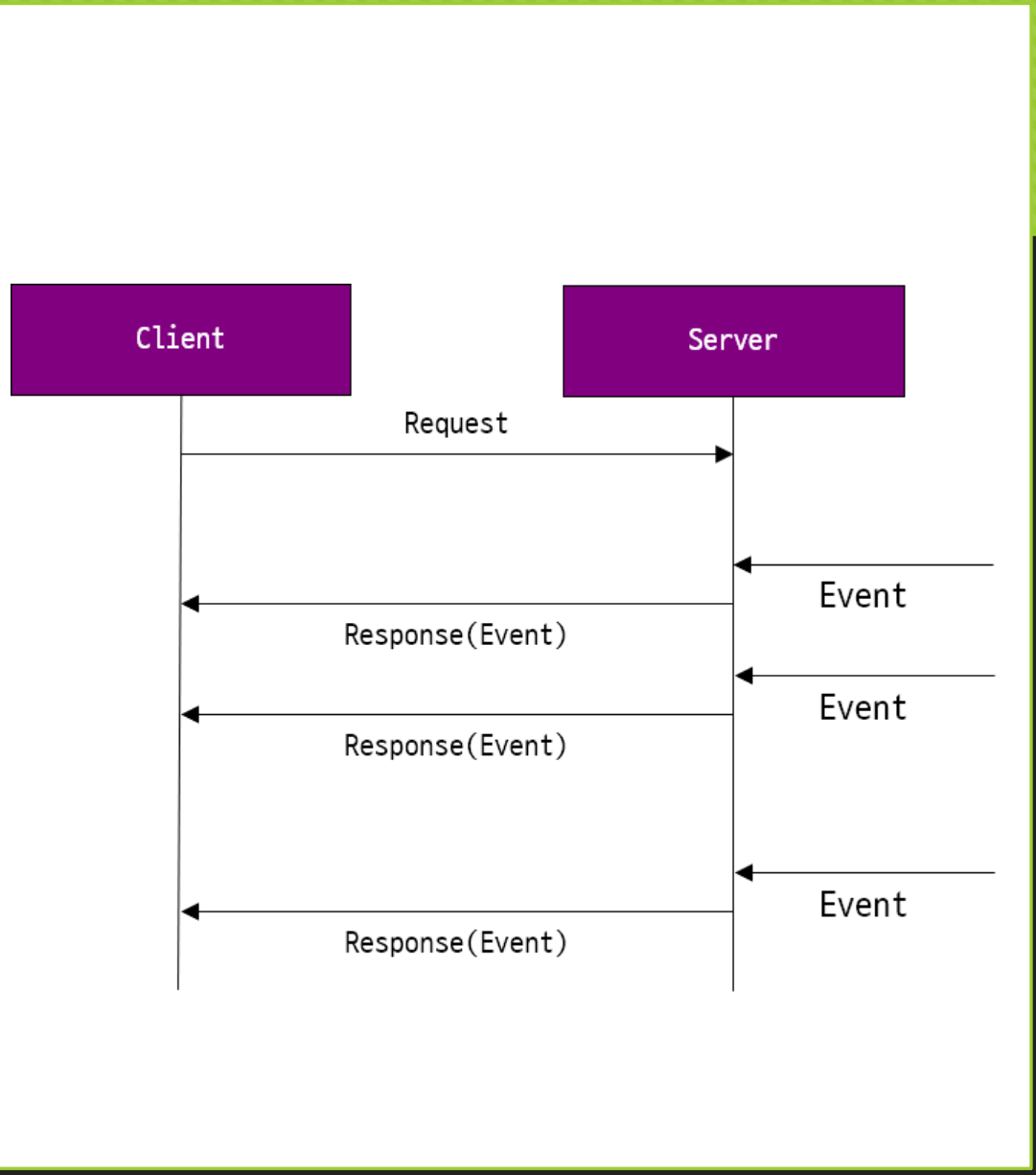
WebSocket

- 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

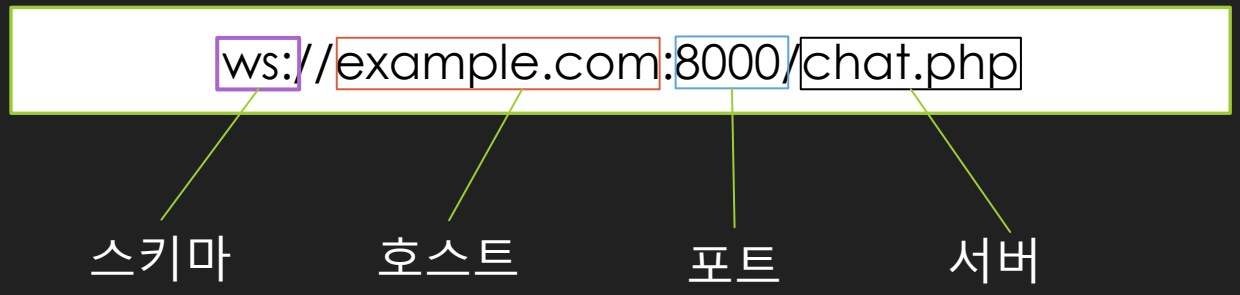
- 기존 방식의 문제점 - 포스트 백(postback)과 AJAX
  - 웹은 HTTP 요청/응답 모델을 기반으로 구축됐다. HTTP는 Stateless 프로토콜이어서 서버와 클라이언트 간의 통신은 각각의 독립적인 요청과 응답의 쌍으로 구성된다.
  - 클라이언트가 서버에 정보를 요청하면 서버는 적절한 HTML 문서로 응답하고 페이지가 새로 고쳐진다. 이러한 방식을 포스트 백 이라 한다.
  - 사용자가 무엇인가 새로운 행동을 할 때까지 둘 사이에는 아무런 일도 일어나지 않으며 새로운 페이지 로딩은 화면이 깜빡거리는 현상을 수반한다.
  - 2005년 AJAX가 등장하면서 포스트 백의 문제를 해결할 수 있게 됐다.
  - AJAX는 자바 스크립트의 XmlHttpRequest 오브젝트로 사용자 인터페이스의 나머지 부분을 방해하지 않고 자바스크립트 코드의 비동기 실행을 가능하게 해준다.
  - 전체 페이지를 다시 로딩하는 대신 AJAX로 웹 페이지의 일부분만 전송할 수 있다.
  - AJAX는 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 제공해주는 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.



WebSocket

- 브라우저 지원
  - 웹소켓 프로토콜 최신 사양은 RFC-6455이며, 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 주소이여야 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	DOMString	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	DOMString	서버에서 선택한 확장명. 이것은 현재 연결에 의해 협의된 빈 문자열 또는 확장자 목록이다.
onclose	EventListener	WebSocket 인터페이스의 연결상태가 readyState 에서CLOSED 로 바뀌었을 때 호출되는 이벤트 리스너. 이 이벤트 리스너는 " close " 라는 이름의 CloseEvent를 받는다.
onerror	EventListener	"error" 라는 이름의 이벤트가 발생하면 처리할 핸들러. 이는 에러가 발생하는 상황에 호출된다.
onmessage	EventListener	"message" 이름의 MessageEvent 이벤트가 발생하면 처리할 핸들러. 이는 서버로부터 메세지가 도착했을 때 호출된다.
onopen	EventListener	WebSocket 인터페이스의 연결상태가 readyState 에서 OPEN으로 바뀌었을 때 호출되는 이벤트 리스너. 연결 상태가 OPEN으로 바뀌었다는 말은 데이터를 주고 받을 준비가 되었다는 뜻이다. 이 리스너가 처리하는 이벤트는 " open " 이라는 이벤트 하나이다.
protocol	DOMString	서버에 의해 선택된 서브 프로토콜을 가리킨다. 이 값은 객체를 생성하면서 protocols 파라미터에 전달했던 값 들 중 하나이다.
readyState	unsigned short	연결의 현재 상태. 값은 Ready state constants 중에 하나이다. 읽기 전용.
url	DOMString	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 포멧이며, 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

server

```
// 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);
});
```

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

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: 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

server

```
// 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.
```

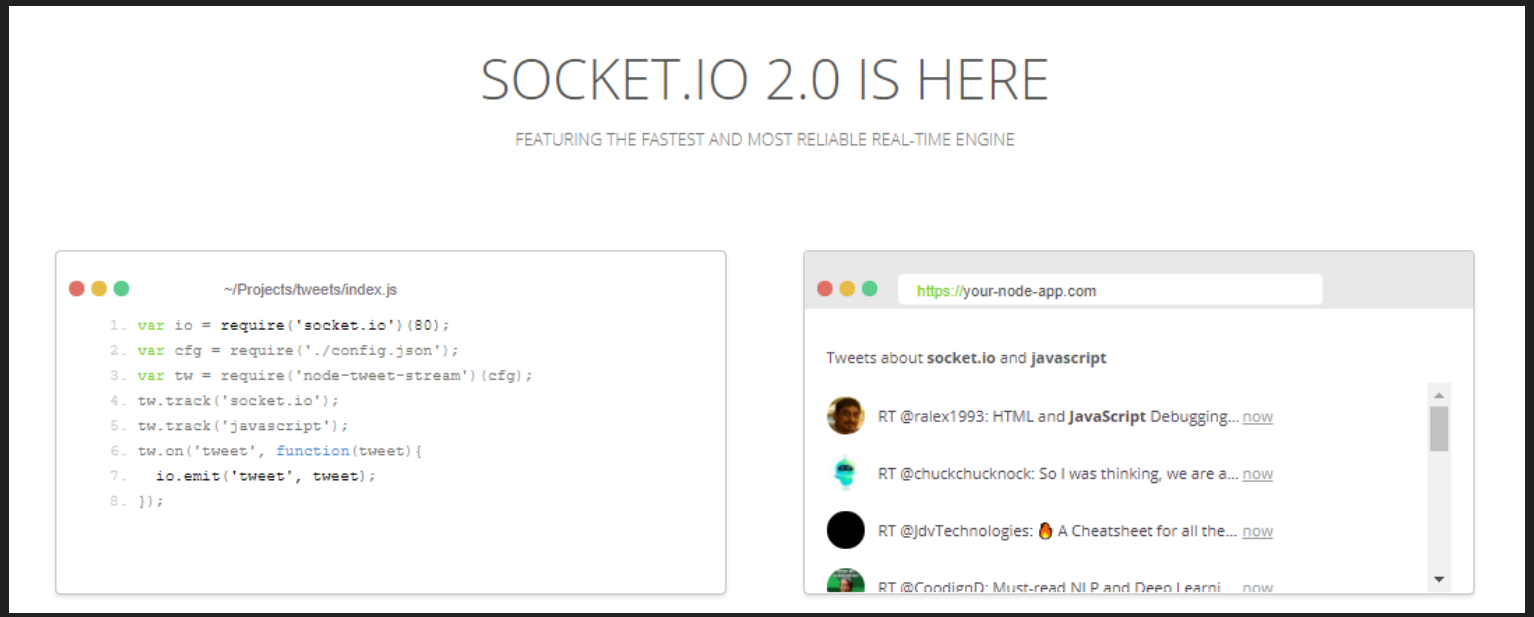
client

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

```
WebSocket Client Connected
Received: 'Hi this is WebSocket server!'
```

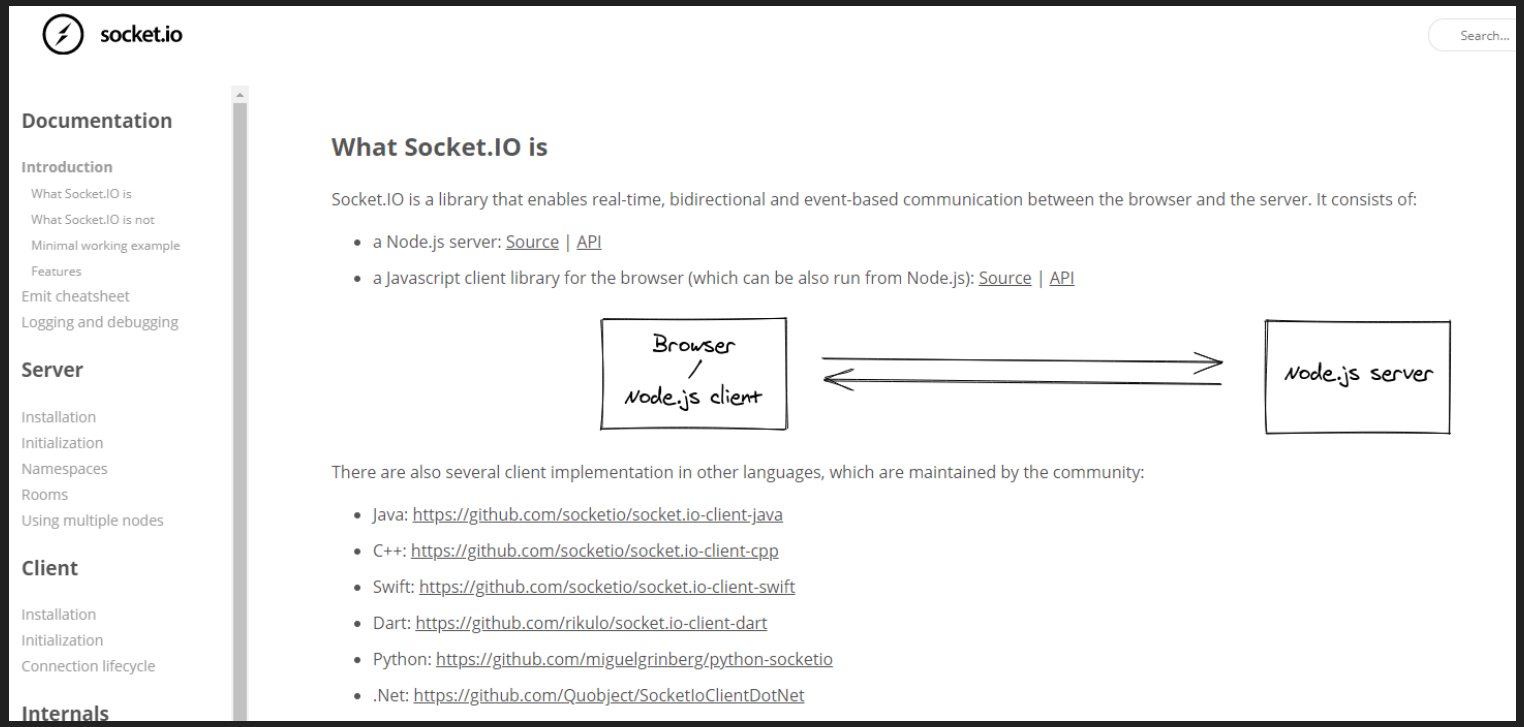
Socket.io

- 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 API 인 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/



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');
});
```

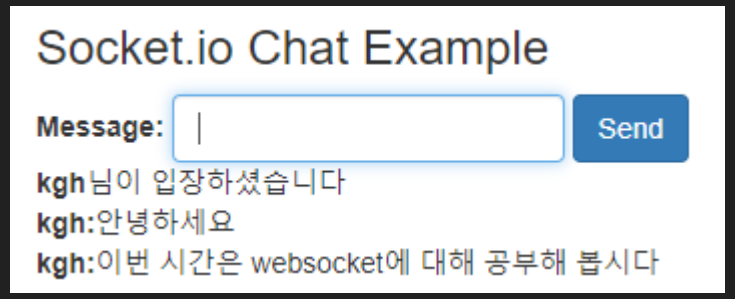
Socket.io

- 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);
        });
```

Socket.io

- socket.io 예제 - Client1_2




```
        // 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>
```

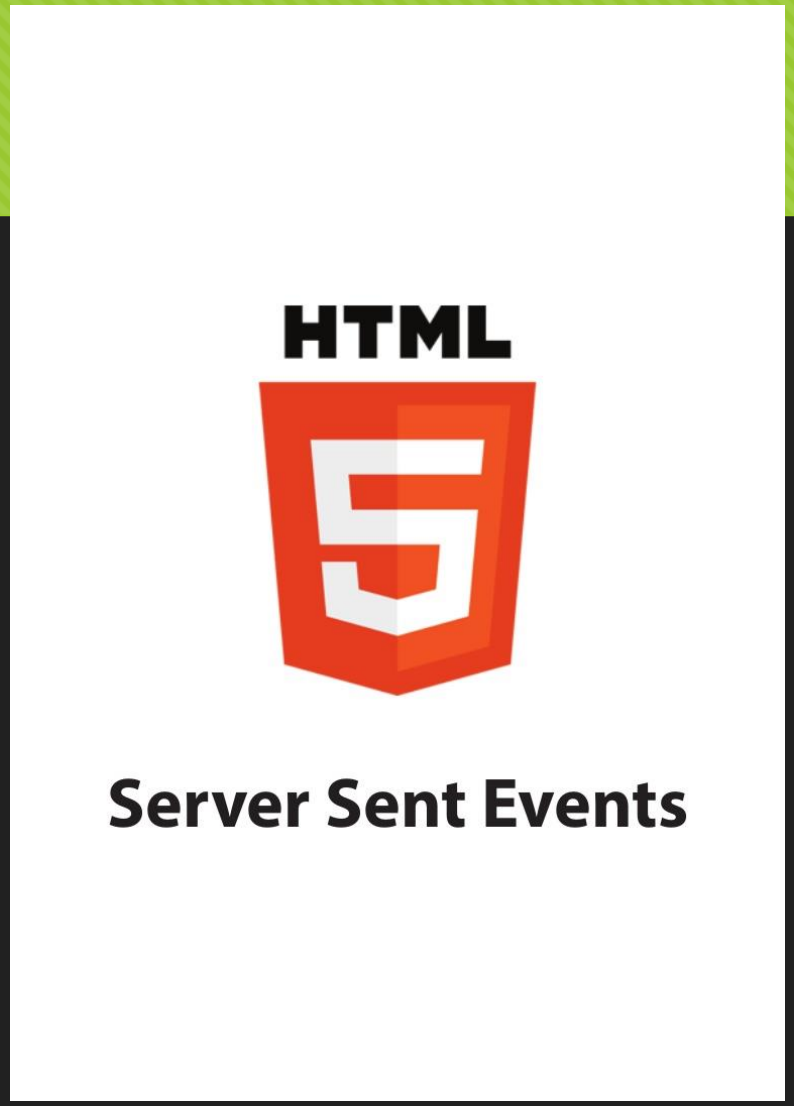
Socket.io

- socket.io 예제 - Client2_2


```
<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)

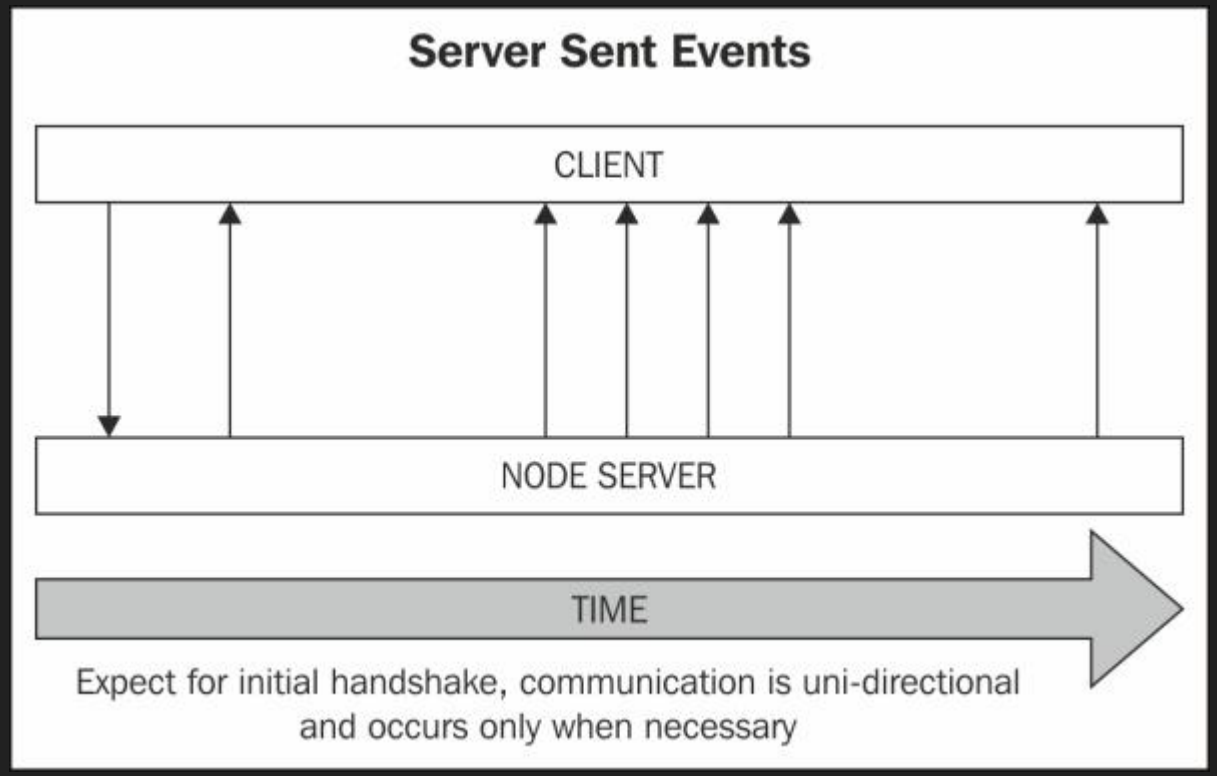


SSE(Server Sent Event)

- 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>
```

Fetch API

- Fetch 소개
  - 기존에는 XMLHttpRequest 의 사용법이 편리하지 못했다.
  - JQuery가 사람들 사이에서 준 필수로 작용하면서 자연스레 XMLHttpRequest 의 사용 빈도는 떨어져 가고 JQuery에서 사용하는 Ajax를 선호하게 되었다.
  - 이후 node.js의 등장 및 react와 같은 프레임워크의 상용화와 jquery의 성능상 문제가 대두되면서 자연스레 jquery의 사용이 줄어들었다.
  - 하지만 jquery의 사용이 줄었다 하더라도 jquery의 ajax 기능을 대체할 수 있는 기능이 XMLHttpRequest 를 제외하고는 자바 스크립트에서 존재하지 않았다.
  - 이에 ES2018 이 출시되면서 ajax 기능을 대체할 fetch 라는 기능을 추가한다.

Fetch API

- Fetch api
  - promise 형태의 비동기 통신을 지원하는 API
  - ES2017 때 나온 기술이며 기존의 ajax및 XMLHttpRequest를 대체하는 기술
  - Fetch API를 이용하면 Request나 Response와 같은 HTTP의 파이프라인을 구성하는 요소를 조작하는 것이 가능하다.
  - 또한 fetch() 메서드를 이용하는 것으로 비동기 네트워크 통신을 알기 쉽게 기술할 수 있다.

Fetch API

- Ajax 와의 차이점
  - fetch()로 부터 반환되는 Promise 객체는 HTTP error 상태를 reject하지 않는다. HTTP Statue Code가 404나 500을 반환하더라도 상관없다. 대신 ok 상태가 false인 resolve가 반환되며, 네트워크 장애나 요청이 완료되지 못한 상태에는 reject가 반환된다.
  - 보통 fetch는 쿠키를 보내거나 받지 않는다. 사이트에서 사용자 세션을 유지 관리해야 하는 경우 인증되지 않는 요청이 발생한다. 쿠키를 전송하기 위해서는 자격증명(credentials) 옵션을 반드시 설정해야 한다.

Fetch API

- 주요 인터페이스
  - Body : Request / Response 의 본문의 유형이 무엇인지와 처리하는 방식을 정할 때 사용한다.
  - Headers : Request / Response 의 Header Instance 를 설정할 수 있다.
  - Request : 요청에 사용되는 객체, 직접적으로 사용되는 일은 없고 다른 API 작업의 결과로 반환되는 경우가 많다.
  - Response : 응답에 사용되는 객체

Fetch API

- 사용방법
  - 사용방법은 promise와 비슷하며 fetch 내에는 option이 들어간다.
  - option 의 내용은 request의 속성을 설정하여 사용할 수 있다.

```
fetch( url , [option]).then(function(response) {
  // Code ...
}).catch(function(error) {
  // Error ...
});
```

Fetch API

- 기본적인 사용방법

fetch1.ejs

```
<body>
    <div id="demo">
        <button type="button" onclick="loadXMLDoc()">Change Content</button>
    </div>
    <script>
        function loadXMLDoc() {
            fetch("fetch1_result1")
            .then(response => {
                console.log(response);
                return response.text();
            }).then(
                text => document.getElementById("demo").innerHTML = text
            );
        }
    </script>
</body>
```

router2.js

```
app.get("/fetch1", function(req,res){
    res.render("fetch1.ejs");
});

app.get("/fetch1_result1", function(req,res){
    res.send("이것은 xmlhttprequest 를 활용한 비동기 통신입니다.");
});
```

이것은 xmlhttprequest 를 활용한 비동기 통신입니다.

Fetch API

- Request 설정
  - Request는 fetch url 파라미터 뒤에 선언해주며 객체 형태로 옵션들을 전달한다.
  - Request에 쓰이는 옵션들은 다음과 같다.

속성	설명
method	사용할 메소드를 선택 (GET, POST, PUT, DELETE etc)
headers	헤더에 전달할 값
body	request 바디에 전달할 값
mode	cors 등의 값을 설정 (cors, no-cors, same-origin)
cache	캐쉬 사용 여부 (no-cache, reload, force-cache, only-if-cached)
credentials	자격 증명을 위한 옵션 설정(include, same-origin, omit) (Default. same-origin)

Fetch API

- Response
  - Response는 Fetch를 실행할 때 결과로 리턴 되는 객체이다.
  - Response 메서드는 상당히 많은 메서드를 가지고 있지만 모든 것을 기억할 필요는 없다.
  - 주요 속성/메서드는 다음과 같다.

메서드	설명
response.status	HTTP Status의 정수치, 기본값 200
response.statusText	HTTP Status 코드의 메서드와 일치하는 문자열, 기본값은 "OK"
response.ok	상술한 프로퍼티에서 사용한 HTTP Status 코드가 200에서 299중 하나임을 체크하는 값, Boolean를 반환
body	response 바디에 전달하는 값
header	response 헤더에 전달하는 값

Fetch API

- header
  - Request, Response에 둘 다 가지고 있는 객체
  - Request와 Response의 성질을 정의하는 객체
  - 보통 Request의 성질을 정의하고자 할 때 사용하며 header를 직접 선언하여 사용하는 경우가 존재하며, 혹은 request 에 보낼 option 에 headers 라는 속성에서 설정도 가능하다.

```
const header = new header({...})

or

fetch( url , {headers : ... })...
```

Fetch API

- Body
  - Request, Response에 둘 다 존재하는 객체
  - 본문의 내용을 실어 보내거나 응답받은 내용을 처리할 때 쓰는 객체
  - 해당 객체에서는 내용을 어떤식으로 받을지 제공하는 메서드가 존재한다.

속성	설명
body.text()	데이터를 문자열로 리턴
body.json()	데이터를 json 타입으로 리턴
body.blob()	데이터를 바이트 단위 데이터스트림으로 리턴
body.arraybuffer()	데이터를 arraybuffer 형태로 받기
body.formdata()	데이터를 formdata 형태로 받기

Fetch API

- Request 설정 예제 1

fetch2.ejs

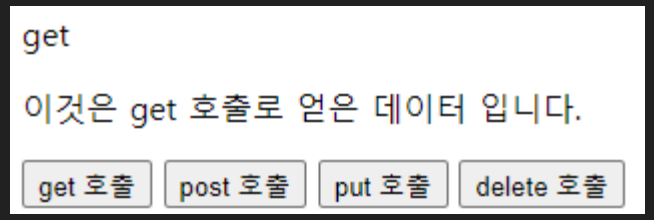
```
<div>
    <h1>데이터 삽입 후 확인</h1>
    <label>이름 : <input type="text" id="name" name="name"></label><br>
    <label>제목 : <input type="text" id="title" name="title"></label><br>
    <label>내용 : <input type="text" id="context" name="context"></label><br>
    <button id="btn">전송</button>
</div>
<div>
    <p id="result"></p>
</div>
<script>
    var myHeaders = new Headers({
        'Content-Type': 'application/json'
    });

    document.getElementById("btn").addEventListener("click", () => {
        fetch("fetch3_result1", {
            method: 'post',
            //headers: { 'Content-Type': 'application/json' },
            headers: myHeaders,
            body : JSON.stringify({
                name: document.getElementById('name').value,
                title: document.getElementById('title').value,
                context: document.getElementById('context').value
            })
        }).then(response => response.json())
        .then(
            json => document.getElementById('result').innerHTML = json.context
        );
    });
</script>
```

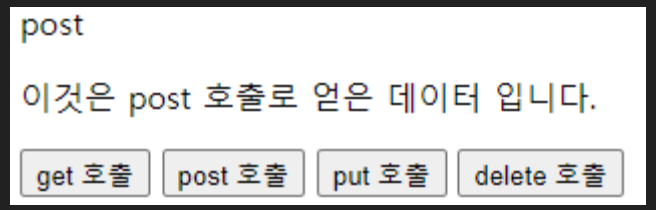
router2.js

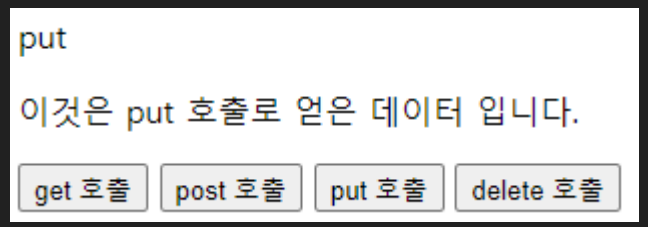
```
app.get("/fetch2", function(req,res){
    res.render("fetch2.ejs");
});

app.get("/fetch2_result1", function(req,res){
    var data = {
        method : "get",
        des : "이것은 get 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});

app.post("/fetch2_result1", function(req,res){
    var data = {
        method : "post",
        des : "이것은 post 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});

app.put("/fetch2_result1", function(req,res){
    var data = {
        method : "put",
        des : "이것은 put 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});

app.delete("/fetch2_result1", function(req,res){
    var data = {
        method : "delete",
        des : "이것은 delete 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});
```

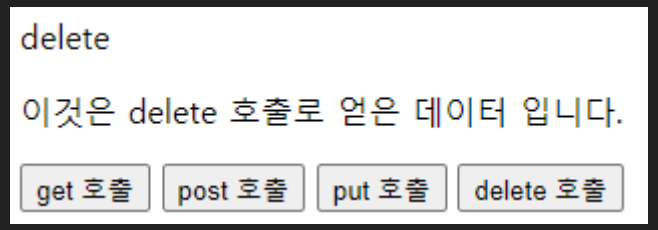
Fetch API

- Request 설정 예제 1







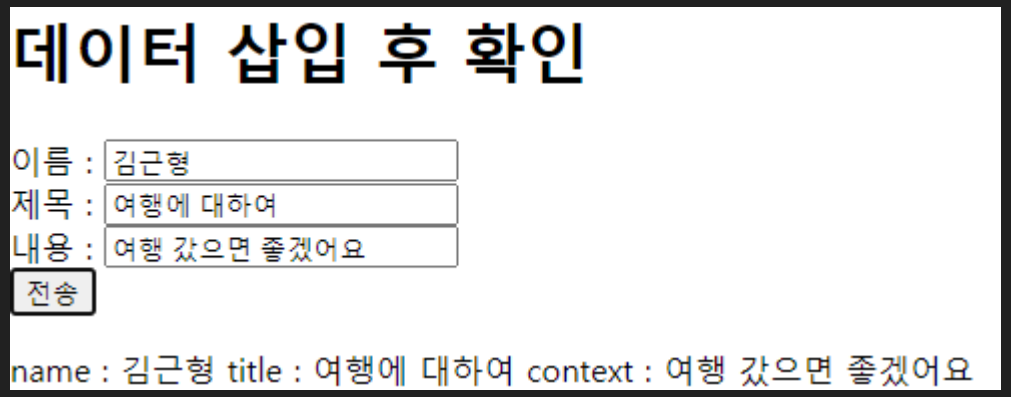


Fetch API

- 데이터를 받아 보내는 예제

router2.js

```
app.get("/fetch3", function(req,res){
    res.render("fetch3.ejs");
});

app.post("/fetch3_result1", function(req,res){
    var result = {
        context : `name : ${req.body.name}    title : ${req.body.title}
context : ${req.body.context}`
    }
    res.json(result);
});
```



fetch3.ejs

```
<div>
    <h1>데이터 삽입 후 확인</h1>
    <label>이름 : <input type="text" id="name" name="name"></label><br>
    <label>제목 : <input type="text" id="title" name="title"></label><br>
    <label>내용 : <input type="text" id="context" name="context"></label><br>
    <button id="btn">전송</button>
</div>
<div>
    <p id="result"></p>
</div>
<script>
    var myHeaders = new Headers({
        'Content-Type': 'application/json'
    });

    document.getElementById("btn").addEventListener("click", () => {
        fetch("fetch3_result1", {
            method: 'post',
            //headers: { 'Content-Type': 'application/json' },
            headers: myHeaders,
            body : JSON.stringify({
                name: document.getElementById('name').value,
                title: document.getElementById('title').value,
                context: document.getElementById('context').value
            })
        }).then(response => {
            if(response.ok){
                return response.json();
            }
            throw new Error('내부적인 에러 발생');
        }).then(
            json => document.getElementById('result').innerHTML = json.context
        ).catch( err => console.log("에러발생 : ",err.message));
    });
</script>
```

Axios

- Axios
  - Axios는 브라우저, Node.js를 위한 Promise API를 활용하는 HTTP 비동기 통신 라이브러리

Axios

- Axios 특징
  - 브라우저 환경에서는 XMLHttpRequests 요청을 생성하고 http 요청을 생성
  - Promise API 기능 지원
  - 요청/응답 차단(Intercept), 데이터 변환, 요청 취소 가능
  - JSON 데이터 자동 변환
  - 크로스 브라우징에 대응 가능
  - 사이트 간 요청 위조(XSRF) 보호를 위한 클라이언트 사이드 지원

Axios

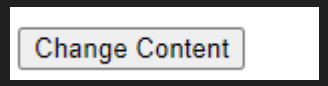
- Axios 특징

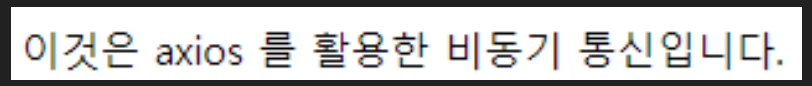
axios1.ejs

```
<div id="demo">
    <button type="button" onclick="loadXMLDoc()">Change Content</button>
</div>
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    function loadXMLDoc() {
        axios.get("axios1_result1")
        .then(response => {
            console.log(response);
            document.getElementById("demo").innerHTML = response.data;
        });
    }
</script>
```

router3.js

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

    app.get("/axios1", function(req,res){
        res.render("axios1.ejs");
    });

    app.get("/axios1_result1", function(req,res){
        res.send("이것은 axios 를 활용한 비동기 통신입니다.");
    });
```





Axios

- Axios 기능
  - 요청하고자 하는 method 방식을 메소드 명으로 가지고 있다.
    - get : axios.get(url[, config])
    - post : axios.post(url[, data[, config]])
    - put : axios.put(url[, data[, config]])
    - patch : axios.patch(url[, data[, config]])
    - delete : axios.delete(url[, config])

Axios

- Axios 기능
  - config에 선언할 수 있는 기능은 다음과 같다.

속성	설명
url	요청에 사용될 서버 URL
method	요청을 할 때 사용될 메소드 이름
baseURL	`url` 속성 값이 절대 URL이 아니라면, `url` 앞에 `baseURL`이 붙는다
transformRequest	서버에 보내기 전에 요청 데이터를 변경한다. 요청 메소드 'PUT', 'POST' 및 'PATCH' 에만 적용 가능하다
transformResponse	응답할 데이터에 대한 변경을 전달해 then/catch에 전달하도록 허용한다.
headers	서버에 전송 될 사용자 정의 헤더
params	요청과 함께 전송 될 URL 매개 변수
paramsSerializer	`params`를 직렬화(serializing) 하는 옵션 함수
data	요청 본문(request body)으로 전송할 데이터, 'PUT', 'POST' 및 'PATCH' 요청 메소드에만 적용 가능하다.' transformRequest' 가 설정되지 않은 경우 [string, plain object, ArrayBuffer, ArrayBufferView, URLSearchParams] 중 하나가 되어야 한다.

Axios

- Axios 기능
  - config에 선언할 수 있는 기능은 다음과 같다.

속성	설명
timeout	요청이 타임 아웃되는 밀리 초(ms)를 설정, 요청이`timeout` 설정 시간보다 지연될 경우, 요청은 중단된다 기본값은 0 (타임아웃 없음)
withCredentials	자격 증명(credentials)을 사용하여 크로스 사이트 접근 제어(cross-site Access-Control) 요청이 필요한 경우 설정한다. 기본값은 false
adapter	테스트를 보다 쉽게 해주는 커스텀 핸들링 요청을 허용한다. 유효한 응답(Promise)을 반환해야 한다.
auth	HTTP 기본 인증(auth)이 사용되며, 자격 증명(credentials)을 제공함을 나타낸다. 기존의 `Authorization` 커스텀 헤더를 덮어쓰는 `Authorization` 헤더(header)를 설정한다.
responseType	서버에서 응답할 데이터 타입을 설정한다. ['arraybuffer', 'blob', 'document', 'json', 'text', 'stream']
responseEncoding	응답 디코딩에 사용할 인코딩을 설정한다. 클라이언트 사이드 요청 또는 `responseType`이 'stream'인 경우는 무시한다
xsrfCookieName	xsrf(Cross-site request forgery) 토큰(token)에 대한 값으로 사용할 쿠키 이름
xsrfHeaderName	xsrf(Cross-site request forgery) 토큰 값을 운반하는 HTTP 헤더 이름

Axios

- Axios 기능
  - config에 선언할 수 있는 기능은 다음과 같다.

속성	설명
onUploadProgress	업로드 프로그래스 이벤트를 처리한다.
onDownloadProgress	다운로드 프로그래스 이벤트를 처리한다.
validateStatus	주어진 HTTP 응답 상태 코드에 대한 약속을 해결할지 거절 할지를 정의한다. `validateStatus`가`true`를 반환하면 (또는`null`,`undefined`) promise를 resolve 한다. 그렇지 않으면 promise가 reject 된다. return status >= 200 && status < 300; // 기본 값
maxRedirects	Node.js에서 리디렉션 가능한 최대 갯수를 정의한다.
socketPath	Node.js에서 사용될 UNIX 소켓을 정의한다. 기본 값은 null
httpAgent/httpsAgent	Node.js에서 http와 https 요청을 수행 할 때 사용할 커스텀 에이전트를 정의한다. 이것은 기본적으로 활성화 되지 않은 `keepAlive`와 같은 옵션을 추가 할 수 있게 한다.
proxy	프록시 서버의 호스트 이름과 포트를 정의한다. 기존의 `http_proxy` 및 `https_proxy` 환경 변수를 사용하여 프록시를 정의 할 수도 있다.
cancelToken	요청을 취소하는 데 사용할 수 있는 취소 토큰을 지정한다.

Axios

- Axios 응답 스키마
  - 요청에 따른 응답 결과는 다음 정보가 들어 있다

속성	설명
data	서버가 제공한 응답(데이터)
status	서버 응답의 HTTP 상태 코드
statusText	서버 응답으로부터의 HTTP 상태 메시지
headers	서버가 응답 한 헤더는 모든 헤더 이름이 소문자로 제공됨
config	요청에 대해 `axios`에 설정된 구성(config)정보
request	응답을 생성한 요청

Axios

- Axios 예제 1

axios2.ejs

```
<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 src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    function loadJsonBy(md) {
        axios({
            method: md,
            url: "axios2_result1"
        }).then(response => {
                document.getElementById('method').innerHTML = response.data.method;
                document.getElementById('des').innerHTML = response.data.des;
        });
    }
</script>
```

router3.js

```
app.get("/axios2", function(req,res){
    res.render("axios2.ejs");
});

app.get("/axios2_result1", function(req,res){
    var data = {
        method : "get",
        des : "이것은 get 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});

app.post("/axios2_result1", function(req,res){
    var data = {
        method : "post",
        des : "이것은 post 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});

app.put("/axios2_result1", function(req,res){
    var data = {
        method : "put",
        des : "이것은 put 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});

app.delete("/axios2_result1", function(req,res){
    var data = {
        method : "delete",
        des : "이것은 delete 호출로 얻은 데이터 입니다."
    }
    res.json(data);
});
```

Axios

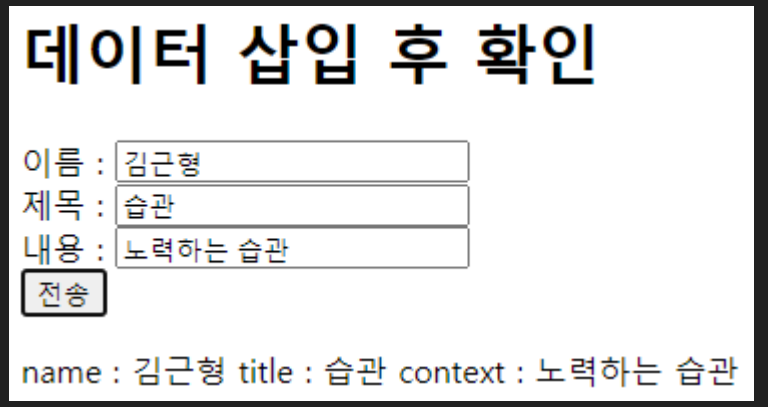
- Axios 예제 2-1

axios3.ejs

```
<div>
    <h1>데이터 삽입 후 확인</h1>
    <label>이름 : <input type="text" id="name" name="name"></label><br>
    <label>제목 : <input type="text" id="title" name="title"></label><br>
    <label>내용 : <input type="text" id="context" name="context"></label><br>
    <button id="btn">전송</button>
</div>
<div>
    <p id="result"></p>
</div>
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    document.getElementById("btn").addEventListener("click", () => {
        axios.post("axios3_result1",{
            name: document.getElementById('name').value,
            title: document.getElementById('title').value,
            context: document.getElementById('context').value
        }).then(response => {
            if(response.status >= 200 && response.status < 300){
                document.getElementById('result').innerHTML = response.data.context;
            }else{
                throw new Error('내부적인 에러 발생');
            }
        }).catch( err => console.log(err));
    });
</script>
```

router3.js

```
    app.get("/axios3", function(req,res){
        res.render("axios3.ejs");
    });

    app.post("/axios3_result1", function(req,res){
        var result = {
            context : `name : ${req.body.name}    title : ${req.body.title}
context : ${req.body.context}`
        }
        res.json(result);
    });
```



Axios

- Axios 예제 2-2

axios4.ejs

```
<div>
    <h1>데이터 삽입 후 확인</h1>
    <label>이름 : <input type="text" id="name" name="name"></label><br>
    <label>제목 : <input type="text" id="title" name="title"></label><br>
    <label>내용 : <input type="text" id="context" name="context"></label><br>
    <button id="btn">전송</button>
</div>
<div>
    <p id="result"></p>
</div>
<script src="https://unpkg.com/axios/dist/axios.min.js"></script>
<script>
    document.getElementById("btn").addEventListener("click", () => {
        axios.get("axios4_result1",{
            params : {
                name: document.getElementById('name').value,
                title: document.getElementById('title').value,
                context: document.getElementById('context').value
            }
        }).then(response => {
            if(response.status >= 200 && response.status < 300){
                document.getElementById('result').innerHTML = response.data.context;
            }else{
                throw new Error('내부적인 에러 발생');
            }
        }).catch( err => console.log(err));
    });
</script>
```

router3.js

```
    app.get("/axios4", function(req,res){
        res.render("axios3.ejs");
    });

    app.get("/axios4_result1", function(req,res){
        var result = {
            context : `name : ${req.params.name}    title : ${req.params.title}
context : ${req.params.context}`
        }
        res.json(result);
    });
```