Node.JS 기술 소개

Node.js 역사

2009 : joyent의 지원을 받는 Ryan Dahl이 bdfl이 되어 linux에 배포됨

( \* bdfl(Benevolent Dictator For Life ) - 오픈소스 소프트웨어 개발리더

2011 : 오픈소스 커뮤니티에 의해 작성된 라이브러리를 배포하고 공유할 수있는 npm(node package manager) 도입

2011 : Miscrosoft 와 joyent가 협력하여 Windows에서 사용 가능한 버전의 Node.js를 구현함

2012 : Ryan Dahl이 물러나고, 동료이자 npm개발자인 Isaac Schlueter가 BDFL롤 활동

2014 : TJ Fontaine 프로젝트의 새로운 리더로 활동

2014 : 노드 JS 주요 개발자를 중심으로 Node Forward라는 기술위원회가 결성되고 독자기술인 io.js 발표

( io.js는 BDFL 주도가 아닌 커뮤니티 주도 개발 회의과정을 공개, SNS로 내부 소식을 전파 )

2015 : Node.js와 io.js 커뮤니티가 Node.js로 통합 될 것이라 발표

Node.js 의 장점

* 비동기 이벤트 IO 동시처리
* javascript를 사용
* 서버와 클라이언트 모두 동일한 코드를 공유
* npm,노드패키지 모듈은 이미 거대하고, 성장중.
* github등을 통해 많은 코드가 공유되는 활발한 커뮤니티
* 큰 파일을 스트리밍 할 수 있음.

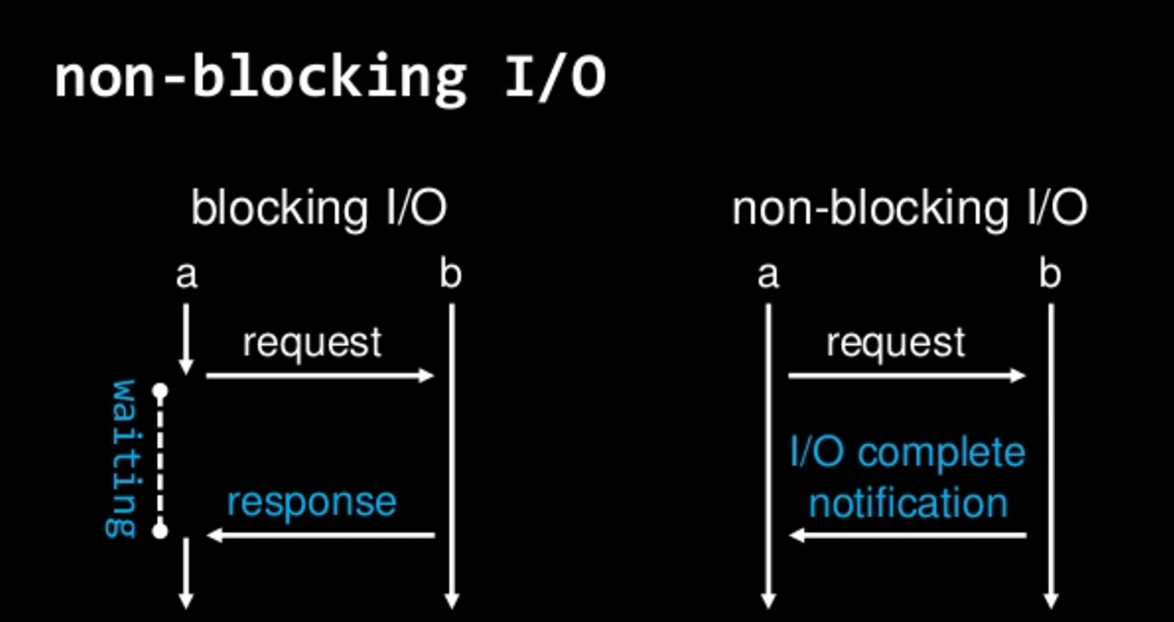
Node.js 의 단점

* 기본적으로 싱글쓰레드로 동작하므로 CPU의존성이 높은 경우 적합하지 않음.
* Node를 사용한다면 관계형 데이터베이스를 다루기 쉽지 않음.
* callback이 중첩될경우 지옥이 발생함.
* Jaascipt레벨이 낮은수준에서 노드를 사용하면 개념적 문제에 직면할 수있음.

Node.js를 사용해야하는 이유

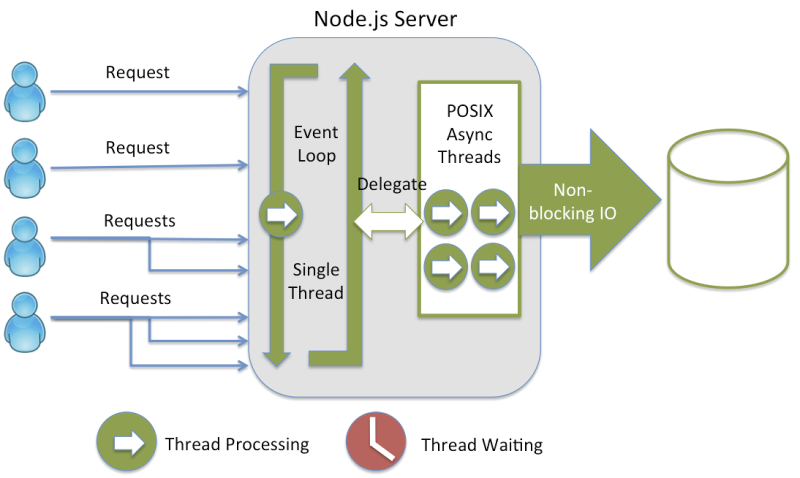
non-blocking I/O

* 작업이 완료되기 전에 다른 작업을 계속할 수 있는 I/O 처리방식
* 데이터베이스 작업 또는 통신을 시작한 후 I/O가 기다리지 않고 동작이 완료된 후에 작업을 진행
* 여전히 완료까지 대기하여야 하지만 동작에 의존성이 없는 다른 처리를 계속함



sing thread event-driven

* 마우스,메시지, 혹은 다른 프로그램이나 쓰레드의 이벤트에 의해서 프로그램의 흐름이 결정되는 구조
* 일반적으로, 메인 루프가 이벤트를 수신하고, 개별이벤트 처리기를 호출해주는 방식
* 단일 프로세스에서 최소한의 오버 헤드로 수천개의 동시 연결을 처리 할 수 있습니다.



하나의 환경

브라우저뿐만아니라 웹서버에서 JavaScript를 사용하면 양쪽에서 동일하게 작동하는 코드를 공유할수 있습니다.

빠른처리

노드는 Google크롬이 사용하는 V8 Javascirpt 런타임 엔진을 사용합니다.

V8 javScript엔진은 모두 C언어를 사용하여 작성이 되어 Ruby, Python또는 Perl보다 훨씬 빠릅니다.

Node.js가 잘 사용될 수 있는 환경

웹 소켓 서버

노드의 논 블로킹 아키텍처는 소켓 서버 어플리케이션이나 방송과 괕은 어플리케이션에 가장 적합한 솔루션입니다.

채팅 서버는 Node.js를 기반으로 효율적이고 실시간으로 사용할 수 있습니다.

빠른 파일 업로드 클라이언트

노드를 사용하면 한번에 여러 파일을 업로드 할 수 있습니다.

즉, 주어진 시점에 File1의 일부와 File2의 다른부분이 서버에 있을수 있습니다.

이러한 접근 방식은 파일 업로드 속도를 획기적으로 향상시킵니다.

데이터 스트리밍

노드가 콜백 개념을 다루기 때문에 스트리밍 데이터 흐름에 쉽게 사용할 수 있습니다.

다른 공급업체의 서로 다른 API에서 결과를 가져오는 여행업계에 유용할수 있습니다.

Node.JS 설치

NodeJS 설치

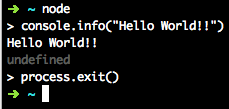
NodeJS 공식 홈페이지 : <https://nodejs.org/ko/>



**Current 버전** : 현재 변화가 일어나는 상태의 버전으로 안전성이 다소 떨어짐

**LTS 버전** : LTS버전은 30개월의 장기적인 지원을 약속받은 버전. 추후에 버전의 숫자가 오른다고 해도 호환성을 깨뜨리는 변경 사항이 추가되지 않음.

Node.JS 실행 - REPL(read Eval Print Loop)



Node.JS 실행 - Console 실행

vi node.basic.js

console.info('Hello World!!');

node node.basic.js

Hello World

Node.JS 실행 - 웹서버 실행

vi node.basic.js

// 모듈을 추출합니다.

var http = require('http');

// 웹 서버를 만들고 실행합니다.

http.createServer(function (request, response) {

response.writeHead(200, { 'Content-Type': 'text/html' });

response.end('<h1>Hello World...!</h1>');

}).listen(52273, function () {

console.log('Server running at http://127.0.0.1:52273/');

});

node node.basic.js



Node.JS 기본 객체

Node.JS 기본 객체

웹 브라우저에서 동작하는 자바스크립트의 최상위 객체는 window객체이지만, node.js는 웹브라우저에서 동작하는것이 아니므로 window 객체가 존재하지 않음.

대신 전역변수와 전역 함수를 가지고 있음.

전역 변수

|  |  |
| --- | --- |
| \_\_filename | 현재 실행 중인 코드의 파일 경로 |
| \_\_dirname | 현재 실행 중인 코드의 폴더 경로 |

console.log('filename:', \_\_filename);

console.log('dirname:', \_\_dirname);

filename: /Users/user/node/node.basic.js

dirname: /Users/user/node

전역 객체

Console 객체

|  |  |
| --- | --- |
| log() | 출력합니다. |
| time() | 시간 측정을 시작합니다. |
| timeEnd() | 시간 측정을 종료합니다. |

\*log() 에서 사용할 수 있는 특수문자는 %d 숫자, %s 문자열, %j JSON 이있음.

console.log('숫자: %d + %d = %d', 273, 52, 273 + 52);

console.log('문자열: %s', 'Hello World .. !', '특수 기호와 상관 없음');

console.log('JSON: %j', { name: 'RintIanTta' });

숫자: 273 + 52 = 325

문자열: Hello World .. ! 특수 기호와 상관 없음

JSON: {"name":"RintIanTta"}

process 객체

process 객체의 속성

|  |  |
| --- | --- |
| argv | 실행 매개변수를 나타냅니다. |
| env | 컴퓨터 환경과 관련된 정보를 나타냅니다. |
| version | Node.js 버전을 나타냅니다. |
| versions | Node.js와 종속된 프로그램 버전을 나타냅니다. |
| arch | 프로세서의 아키텍처를 나타냄 |
| platform | 플랫폼을 나타냄. |

process 객체의 메서드

|  |  |
| --- | --- |
| exit([exitCode=0]) | 프로그램을 종료합니다. |
| memoryUsage() | 메모리 사용 정보 객체를 리턴합니다. |
| uptime() | 현재 프로그램이 실행된 시간을 리턴합니다. |

\* [exitCode=0]는 생략이 가능하며, 해당값이 지닌 의미는 입력하지 않을경우 자동으로 0으로 입력이됨.

\* process.argv 속성과 process.exit() 메서드

// process.argv

process.argv.forEach(function (item, index) {

// 출력합니다.

console.log(index + ' : ' + typeof (item) + ' : ', item);

// 실행 매개변수에 --exit가 있을 때

if (item == '--exit') {

// 다음 실행 매개변수를 얻습니다.

var exitTime = Number(process.argv[index + 1]);

// 일정 시간 후 프로그램을 종료합니다.

setTimeout(function () {

process.exit();

}, exitTime);

}

});

0 : string : /usr/local/bin/node

1 : string : /Users/narumi12/TEST/node.basic.js

2 : string : --exit

3 : string : 10000

\* process 객체의 속성

console.log('- process.env:', process.env);

console.log('- process.version:', process.version);

console.log('- process.versions:', process.versions);

console.log('- process.arch:', process.arch);

console.log('- process.platform:', process.platform);

console.log('- process.connected:', process.connected);

console.log('- process.execArgv:', process.execArgv);

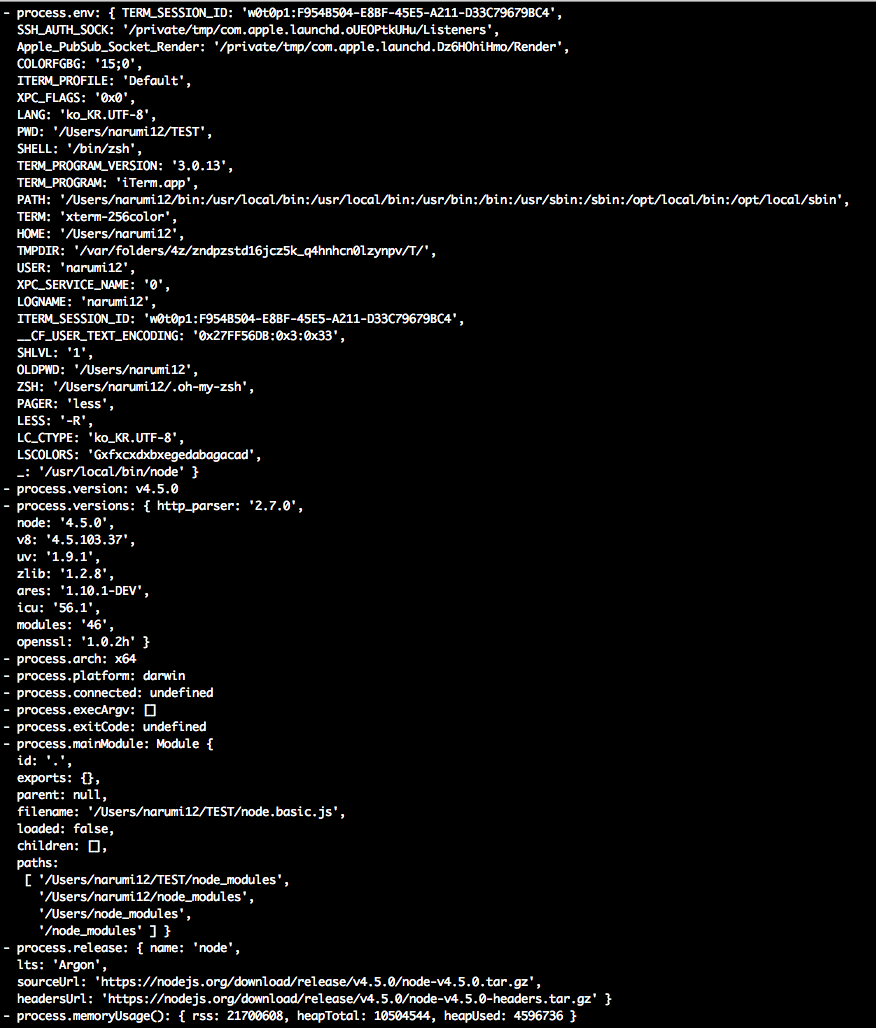
console.log('- process.exitCode:', process.exitCode);

console.log('- process.mainModule:', process.mainModule);

console.log('- process.release:', process.release);

console.log('- process.memoryUsage():', process.memoryUsage());

console.log('- process.uptime():', process.uptime());



Node.JS 기본 모듈

Node.JS 모듈

Node.js는 모듈을 사용해 기능을 확장합니다.

모듈은 기능을 쉽게 사용하고자 메서드와 속성을 미리 정의해 모아 놓은 것. ( <https://nodejs.org/dist/latest-v6.x/docs/api/> )

모듈을 생성하려면 별도의 자바스크립트 파일을 생성해야함.

모듈을 생성할 module.js 와 모듈을 사용할 main.js 으로 구성.

모듈을 생성할때에는 exports 객체를 사용하고, 생성한 모듈을 다른 자바스크립트 파일에서 추출할 때는 require() 함수를 사용.

\* module.js

// 절대값을 구하는 메서드

exports.abs = function(number){

if(0 < number){

return number;

}else{

return -number;

}

};

// 원의 넓이를 구하는 메서드

exports.circleArea = function(radius){

return radius \* radius \* Math.PI;

};

\* main.js

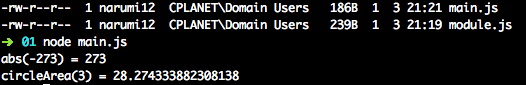
//모듈을 추출합니다.

var module = require('./module.js');

//모듈을 사용합니다.

console.log('abs(-273) = %d', module.abs(-273));

console.log('circleArea(3) = %d', module.circleArea(3));



os 모듈

os 모듈은 운영체제의 정보나 시스템의 정보를 제공하는 모듈.

<https://nodejs.org/dist/latest-v6.x/docs/api/os.html>

// 모듈을 추출합니다.

var os = require('os');

// 모듈을 사용합니다.

console.log(os.hostname());

console.log(os.type());

console.log(os.platform());

console.log(os.arch());

console.log(os.release());

console.log(os.uptime());

console.log(os.loadavg());

console.log(os.totalmem());

console.log(os.freemem());

console.log(os.cpus());

console.log(os.networkInterfaces());

url 모듈

url정보를 문자열 혹은 객체로 변환해 리턴하거나, 매개변수를 조합해 URL문자열을 생성하는 모듈.

<https://nodejs.org/dist/latest-v6.x/docs/api/url.html>

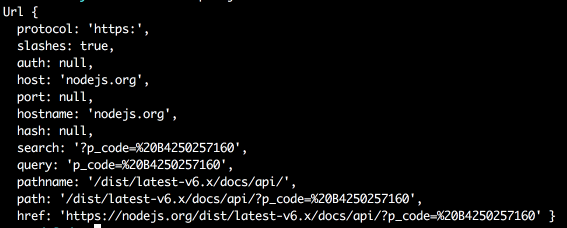
// 모듈을 추출합니다.

var url = require('url');

// 모듈을 사용합니다.

var parsedObject = url.parse('https://nodejs.org/dist/latest-v6.x/docs/api/?p\_code= B4250257160');

console.log(parsedObject);



query String 모듈

query String 모듈은 URL 객체의 쿼리와 관련된 모듈이고, URL query를 객체 혹은 문자열로 리턴하는 모듈

<https://nodejs.org/dist/latest-v6.x/docs/api/querystring.html>

// 모듈을 추출합니다.

var url = require('url');

var querystring = require('querystring');

// 모듈을 사용합니다.

var parsedObject = url.parse('http://www.hanbit.co.kr/store/books/look.php?p\_code= B4250257160');

console.log(querystring.parse(parsedObject.query));

{ p\_code: ' B4250257160' }

util 모듈

util 모듈은 Node.js의 보조적인 기능을 모아둔 모듈로 자세한내용은 API Doc에 참고.

<https://nodejs.org/dist/latest-v6.x/docs/api/util.html>

// 모듈을 추출합니다.

var util = require('util');

// 모듈을 사용합니다.

var data = util.format('%d + %d = %d', 52, 273, 52 + 273);

console.log(data);

52 + 273 = 325

crypto 모듈

crypto 모듈은 해시 생성과 암호화를 수행하는 모듈.

<https://nodejs.org/dist/latest-v6.x/docs/api/crypto.html>

// 모듈을 추출합니다.

var crypto = require('crypto');

// 변수를 선언합니다.

var key = 'SAMPLE'; // KEY입력

var input = 'PASSWORD';

// 암호화

var cipher = crypto.createCipher('aes192', key)

cipher.update(input, 'utf8', 'base64');

var cipheredOutput = cipher.final('base64');

// 암호화 해제

var decipher = crypto.createDecipher('aes192', key);

decipher.update(cipheredOutput, 'base64', 'utf8');

var decipheredOutput = decipher.final('utf8');

// 출력합니다.

console.log('원래 문자열: ' + input);

console.log('암호화: ' + cipheredOutput);

console.log('암호화 해제: ' + decipheredOutput);

원래 문자열: PASSWORD

암호화: 331RYcYkyC3JeU5Kkh8UsQ==

암호화 해제: PASSWORD

File System 모듈

file system 모둘은 파일을 처리하는 기능모듈

<https://nodejs.org/dist/latest-v6.x/docs/api/fs.html>

// 모듈을 추출합니다.

var fs = require('fs');

// 파일을 씁니다. 동기로 처리할때 예외처리 try catch

try {

var data = fs.writeFileSync('TextFileOtherWrite.txt', 'Hello World .. !', 'utf8');

console.log(data);

} catch (e) {

console.log(e);

}

// 파일을 씁니다. 비동기로 처리할때 예외처리 콜백함수의 첫번째 매개변수에서 처리

fs.writeFile('TextFileOtherWriteSync.txt', 'Hello World .. !', 'utf8', function (error) {

if (error) {

console.log(error);

} else {

console.log('FILE WRITE COMPLETE');

}

});

Node.JS 이벤트

이벤트 연결

<https://nodejs.org/dist/latest-v6.x/docs/api/events.html>

\*기존 자바스크립트에서 이벤트 연결할때 addEventListener 메소드를 사용한다

<script>

//Window 객체에 load 이벤트를 연결합니다.

window.addEventListener('load', function(){

});

</script>

\* node에서는 이벤트 연결시 on 메소드를 사용한다.

// process 객체에 exit 이벤트를 연결합니다.

process.on('exit', function (code) {

console.log('종료!!');

});

// process 객체에 uncaughtException 이벤트를 연결합니다.

process.on('uncaughtException', function (error) {

console.log('Exception 발생!!');

});

// 2초 간격으로 3번 예외를 발생시킵니다.

var count = 0;

var test = function () {

// 탈출 코드

count = count + 1;

if (count > 3) { return; }

// 예외를 강제로 발생시킵니다.

setTimeout(test, 2000);

error.error.error();

};

setTimeout(test, 2000);

process객체에서 uncaughtException(https://nodejs.org/dist/latest-v6.x/docs/api/process.html#process\_event\_uncaughtexception) 이벤트는

예외가 발생할때 실행되는 이벤트입니다.

이벤트 연결 개수 제한

\* Node.js는 한 이벤트에 10개가 넘는 이벤트 리스너를 연결할 경우 이를 개발자 실수로 간주함.

// 이벤트를 연결합니다.

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

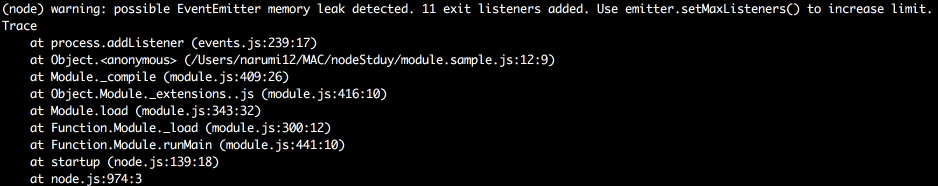
process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });



\* 이벤트 개수 제한을 높이고자 할때 setMaxListenrs() 메서드를 사용한다.

// 이벤트 연결 개수 제한을 15개까지 늘립니다.

process.setMaxListeners(15);

// 이벤트를 연결합니다.

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

process.on('exit', function () { });

이벤트 제거

\* 만약 이벤트에서 문제가 생길경우 이벤트를 제거할 필요가 있습니다. 이때 이벤트를 제거하는 removeListener 메서드를 사용하면 가능합니다.

// 변수를 선언합니다.

var onUncaughtException = function (error) {

// 출력합니다.

console.log('Exception 발생!!');

// 이벤트를 제거합니다.

process.removeListener('uncaughtException', onUncaughtException);

};

// process 객체에 uncaughtException 이벤트를 연결합니다.

process.on('uncaughtException', onUncaughtException);

// 2초 간격으로 예외를 발생시킵니다.

var test = function () {

setTimeout(test, 2000);

error.error.error();

};

setTimeout(test, 2000);

\* 이벤트를 한번만 연결할경우는 once 메서드를 사용한다.

// process 객체에 uncaughtException 이벤트를 연결합니다.

process.once('uncaughtException', function (error) {

console.log('Exception 발생!!');

});

// 2초 간격으로 예외를 발생시킵니다.

var test = function () {

setTimeout(test, 2000);

error.error.error();

};

setTimeout(test, 2000);

이벤트 강제 발생

\*이벤트를 강제로 발생하고자 할때에는 emit 메서드를 사용한다. emit 메서드는 이벤트를 강제로 호출시 해당 이벤트 리스너만 실행이된다.

// exit 이벤트를 연결합니다.

process.on('exit', function (code) {

console.log('종료!!');

});

// 이벤트를 강제로 발생시킵니다.

process.emit('exit');

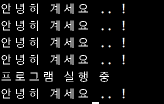
process.emit('exit');

process.emit('exit');

process.emit('exit');

// 프로그램 실행 중

console.log('프로그램 실행 중');



이벤트 생성

nodejs에서 이벤트를 연결할 수 있는 모든 객체는 EventEmitter 객체의 상속을 받습니다.

EventEmitter 객체생성은 events 모듈을 받아서 생성하거나, process 객체 안에 있는 생성자 함수로 생성할수 있다.

// events 모듈을 받아서 생성하는 방식

var EventEmitter = require('events');

var custom = new EventEmitter();

// process 객체 안에 있는 생성자 함수로 생성하는 방식

var custom = new process.EventEmitter();

\* 이벤트를 생성하여, 이벤트를 강제로 실행시키면 이벤트 실행이되는것을 확인할 수 있다.

var EventEmitter = require('events');

// EventEmitter 객체를 생성합니다.

var custom = new EventEmitter();

// 이벤트를 연결합니다.

custom.on('tick', function (code) {

console.log('이벤트 실행');

});

// 이벤트를 강제로 발생시킵니다.

custom.emit('tick');

이벤트 구조

일반적으로 이벤트를 생성하는 부분과 연결하는 부분을 모듈로 분리해서 사용합니다.

보통 app.js 파일은 프로그램 본체가 되는부분을 작성하고, rint.js 파일은 이벤트를 생성할 모듈 파일입니다.

\* rint.js 파일

// EventEmitter 객체를 생성합니다.

exports.timer = new process.EventEmitter();

// 이벤트를 강제로 발생시킵니다.

setInterval(function () {

exports.timer.emit('tick');

}, 1000);

\* app.js 파일

// 모듈을 추출합니다.

var rint = require('./rint.js');

// 이벤트를 연결합니다.

rint.timer.on('tick', function (code) {

console.log('이벤트를 실행합니다.');

});

Node.JS http 모듈

Node.js 에서 서버를 구동할 수 있게 해주는 http 모듈입니다.

Server 객체

\* 웹서버 생성과 실행, 종료에 대해 간단한 샘플을 보자.

// 서버를 생성합니다.  
**var** server = require('http').createServer();  
  
// 서버를 실행합니다.  
server.listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});  
  
// 10초 후 함수를 실행합니다.  
**var** test = **function** () {  
 // 서버를 종료합니다.  
 server.close();  
};  
setTimeout(test, 10000);

\* Server 객체는 EventEmitter 객체를 기반으로 만들어졌으므로 이벤트 방식으로 연결할 수 있다.

// 모듈을 추출합니다.  
**var** http = require('http');  
  
// server 객체를 생성합니다.  
**var** server = http.createServer();  
  
// server 객체에 이벤트를 연결합니다.  
server.on('request', **function** (code) {  
 console.log('Request On');  
});  
  
server.on('connection', **function** (code) {  
 console.log('Connection On');  
});  
  
server.on('close', **function** (code) {  
 console.log('Close On');  
});  
  
// listen() 메서드를 실행합니다.  
server.listen(52273);

response 객체

웹서버에서 응답 메시지를 작성할수 있는 response 객체.

// 메서드 방식으로 웹 서버를 생성하고 실행합니다.  
require('http').createServer(**function** (request, response) {  
 // 응답합니다.  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end('<h1>Hello Web Server with Node.js</h1>');  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

File System 모듈을 사용해서 Html 페이지 로드

// 모듈을 추출합니다.  
**var** fs = require('fs');  
**var** http = require('http');  
  
// 서버를 생성하고 실행합니다.  
http.createServer(**function** (request, response) {  
 // HTML 파일을 읽습니다.  
 fs.readFile('sample.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

// Sample.html

<!DOCTYPE html>  
<html>  
<head>  
 <title>Index</title>  
</head>  
<body>  
 <h1>Hello Node.js</h1>  
</body>  
</html>

**Redirect Page**

Redirect 페이지 처리가 필요할때 'Location' 속성을 이용해서 처리한다

// 모듈을 추출합니다.  
**var** http = require('http');  
  
// 웹 서버를 생성 및 실행합니다.  
http.createServer(**function** (request, response) {  
 response.writeHead(302, { 'Location': 'http://www.11st.co.kr' });  
 response.end();  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

request 객체

웹서버에서 request 이벤트 발생할때 처리하는 객체.

\* node.js에서 spring RequestMapping방식과 동일한 처리하는 방법을 보자

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** url = require('url');  
  
// 서버를 생성 및 실행합니다.  
http.createServer(**function** (request, response) {  
 // 변수를 선언합니다.  
 **var** pathname = url.parse(request.url).pathname;  
 // 페이지를 구분합니다.  
 **if** (pathname == '/coupang' && request.method == 'GET') {  
  
 // GET방식 매개변수를 추출합니다.  
 **var** query = url.parse(request.url, **true**).query;  
 console.log('GET Data:', query);  
  
 // coupang.html 파일을 읽습니다.  
 fs.readFile('coupanghtml', **function** (error, data) {  
 // 응답합니다.  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
 } **else if** (pathname == '/11st' && request.method == 'POST') {  
  
 // POST방식 매개변수를 추출합니다.  
 request.on('data', **function** (data) {  
 console.log('POST Data:', data);  
 });  
  
 // 11st.html 파일을 읽습니다.  
 fs.readFile('11st.html', **function** (error, data) {  
 // 응답합니다.  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
 }  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

# Node.JS 외부 모듈

Node.js는 npm을 기반으로 모듈을 공유합니다.

npm은 모듈을 공유하기 쉬운 환경입니다.

다음과 같이 npm을 이용하여 외부 모듈을 가져와봅시다.

// npm을 이용해 외부모듈 설치  
npm install 모듈명  
npm install ejs  
npm install jade  
  
// 설치한 외부모듈을 사용하기  
**var** ejs = require('ejs');  
**var** jade = require('jade');

### ejs 모듈

ejs모듈은 템플릿 엔진 모듈입니다. 특정형식의 문자열을 HTML 형식의 문자열로 변환합니다.

ejs 기본형식을 통해 어떤식으로 사용하는지 보자

// <% code %> : 자바스크립트 코드를 입력합니다  
// <%=data %> : 데이터를 출력합니다  
// sample.ejs  
<% **var** name = 'RintIanTta'; %>  
<h1><%= name %></h1>  
<p><%= 52 \* 273 %></p>  
<hr />  
<% **for**(**var** i = 0; i < 10; i++) { %>  
 <h2>The Square of <%= i %> is <%= i \* i %></h2>  
<% } %>

ejs페이지에 데이터를 전달하고 호출해보자

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** ejs = require('ejs');  
  
// 서버를 생성하고 실행합니다.  
http.createServer(**function** (request, response) {  
 // sample.ejs 파일을 읽습니다.  
 fs.readFile('sample.ejs', 'utf8', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(ejs.render(data, {  
 name: '11st',  
 description: 'Hello ejs With Node.js .. !'  
 }));  
 });  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

### jade 모듈

ejs모듈과 마찬가지로 템플릿 엔진 모듈입니다.   
현재 Node.js에서 가장 많이 사용되는 웹프레임워크 express 에서 템플릿엔진에서 ejs와 jade 모듈을 주로 사용한다.

\*\* jade 모듈은 현재 pug모듈이라는 이름이 변경되었다. 다만 Node.js 4.X에 최적화된 express 4버전은 여전히 jade모듈을 사용한다.

jade의 기본형식은 들여쓰기를 통해 각 태그의 HTML 구조를 형성합니다.

태그 안에 글자를 입력하고 싶을 때는 지정한 태그 아래에 한 단계 더 들여쓰기를 한 후 원하는 글자를 입력합니다.

태그에 속성을 입력하고 싶을때는 지정한 태그 뒤에 괄호를 사용한다. 속성을 여러 개 입력하고 싶을 때는 쉼표를 사용해 구분합니다.

div 태그는 div태그를 직접입력하지 않고 **#header** 태그를 입력하는 것만으로 id속성이 header 인 div 태그가 생성이되고,

**.classname** 를 입력하는 것만으로도 class 속성이 classname 인 div 태그를 생성할 수 있다

// jade의 기본 형태  
// //는 주석  
// style 태그와 script 태그는 사용할때 '.' 기호를 찍어줘야한다  
   
   
**doctype** html  
html  
 head  
 style.  
 \* { margin:0px; padding:0px; }  
 h1 {  
 color:Red;  
 }  
 script(src="https://code.jquery.com/jquery-1.12.4.js")  
 script.  
 $(document).ready(**function** () {  
 **var** alpha = 10;  
 });  
 title Index Page  
 body  
 // JADE String  
 #header  
 h1 Hello jade .. !  
 h2 Lorem ipsum  
 hr  
 .*article* a(href="http://11st.co.kr", data-test="multiple Attribute") Go To 11st

jada에 데이터를 전달하고 호출해보자

// sample.jada  
// -code : 자바스크립트 코드를 입력합니다  
// #{value} : 데이터를 출력합니다  
// =value : 데이터를 출력합니다.  
  
**doctype** html  
html  
 head  
 title Index Page  
 body  
 // JADE String  
 h1 #{name} .. !  
 h2= description  
 hr  
 - **for**(**var** i = 0; i < 10; i++) {  
 p  
 a(href="http://11st.co.kr") Go To 11st #{i}  
 - }

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** jade = require('jade');  
**var** fs = require('fs');  
  
// 서버를 생성하고 실행합니다.  
http.createServer(**function** (request, response) {  
 // JadePage.jade 파일을 읽습니다.  
 fs.readFile('sample2.jade', 'utf8', **function** (error, data) {  
 // jade 모듈을 사용합니다.  
 **var** fn = jade.compile(data);  
 // 출력합니다.  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(fn({  
 name: 'RintIanTta',  
 description: 'Hello jade With Node.js .. !'  
 }));  
 });  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

## **서버 실행 모듈**

supervisor 모듈과 forever 모듈은 전역 모듈로 터미널에서 곧바로 사용할 수 있는 모듈입니다.

전역모듈은 설치할때 -g 옵션을 사용해서 설치해야합니다. ( -g는 글로벌 설치옵션 )

npm install -g supervisor  
npm install -g forever

### supervisor 모듈

node.js는 파일내용을 변경해도 현재 실행 중인 스크립트에는 영향을 끼치지 못하는데,

supervisor모듈은 파일의 변경 사항을 자동으로 인식하고 실행을 종료시킨 후에 다시 실행하게 해준다.

// sample.server.js  
// 모듈을 추출합니다.  
**var** http = require('http');  
  
// 서버를 생성 및 실행합니다.  
http.createServer(**function** (request, response) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end('<h1>Test - File - 1</h1>');  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

$supervisor sample.server.js

// sample.server.js  
// 모듈을 추출합니다.  
**var** http = require('http');  
  
// 서버를 생성 및 실행합니다.  
http.createServer(**function** (request, response) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end('<h1>Test - File - 2</h1>');  
}).listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});

### forever 모듈

node.js는 예외가 발생하면 웹서비스 전체에 영향을주어 서비스가 죽어버립니다.

이런 예외상항을 대비하고자 만들어진 모듈이 forever모듈입니다.

// sample.server.js  
// 서버를 생성 및 실행합니다.  
require('http').createServer(**function** (request, response) {  
 **if** (request.url == '/') {  
 // 응답합니다.  
 response.write('<!DOCTYPE html>');  
 response.write('<html>');  
 response.write('<head>');  
 response.write(' <title>Forever</title>');  
 response.write('</head>');  
 response.write('<body>');  
 response.write(' <h1>Forever</h1>');  
 response.write('</body>');  
 response.write('</html>');  
 response.end();  
 } **else** {  
 // 오류를 발생합니다.  
 error.error.error();  
 }  
}).listen(52273, **function** () {  
 console.log('Server running at http://127.0.0.1:52273');  
});

$forever start sample.server.js

forever 모듈로 웹 서버를 실행후 http://localhost:52273으로 들어가면 정상적으로 노출이되고,

http://localhost:52273/error 으로 접속시 서버에서 예외가 발생하여 서버가 죽어버리게 되지만

다시 http://localhost:52273 으로 접속하면 다시 웹서비스가 정상적으로 동작합니다.

// forever 서버 종료

$forever stop 0 (프로세스의 번호)

1. express 모듈

웹서버를 개발할때 http모듈로 생성하여 개발한다면, 굉장히 많은 일을 직접처리해야한다.

express모듈은 http모듈에 여러기능을 추가해서 쉽게 사용할수 있게 만든 모듈입니다.

// express모듈 설치

$npm install express@4

기본 서버구동

// 모듈을 추출합니다.

**var** express = require('express');

// 서버를 생성합니다.

**var** app = express();

// request 이벤트 리스너를 설정합니다.

app.use(**function** (request, response) {

response.writeHead(200, { 'Content-Type': 'text/html' });

response.end('<h1>Hello express</h1>');

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log("Server Running at http://127.0.0.1:52273");

});

express모듈로 서버를 생성하면 request객체와 response객체에 다양한 기능이 추가됩니다.

request객체와 response객체의 내용은 API문서에서 자세히 안내하고 있습니다.

response객체 : <http://expressjs.com/ko/4x/api.html#res>

request객체 : <http://expressjs.com/ko/4x/api.html#req>

미들웨어

exprss 모듈은 http모듈과 다르게 큰 특징이 있는데, use()메서드를 활용하여 여러번 호출할수 있고,

특정한 작업을 수행하는 모듈을 분리하여 사용할수 있습니다.

// 모듈을 추출합니다.

**var** express = require('express');

// 서버를 생성합니다.

**var** app = express();

// 미들웨어 설정(1)

app.use(**function** (request, response, next) {

console.log("첫 번째 미들웨어");

next();

});

// 미들웨어 설정(2)

app.use(**function** (request, response, next) {

console.log("두 번째 미들웨어");

next();

});

// 미들웨어 설정(3)

app.use(**function** (request, response, next) {

console.log("세 번째 미들웨어");

// 응답합니다.

response.writeHead(200, { 'Content-Type': 'text/html' });

response.end('<h1>express Basic</h1>');

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log('Server running at http://127.0.0.1:52273');

});

router미들웨어

express모듈은 클라이언트 요청에 적절한 페이지를 제공한는 라우팅기능을제공합니다.(RESTful 서비스로 활용할수 있다.)

express모듈의 router미들웨어를 사용하여 페이지 라우팅을 쉽게 구현할수있습니다.

<http://expressjs.com/ko/starter/basic-routing.html>

<http://expressjs.com/ko/guide/routing.html>

<http://expressjs.com/ko/4x/api.html#router>

// 모듈을 추출합니다.

**var** express = require('express');

// 서버를 생성합니다.

**var** app = express();

// 라우터를 설정합니다.

app.get('/a', **function** (request, response) {

response.send('<a href="/b">Go to B</a>');

});

app.get('/b', **function** (request, response) {

response.send('<a href="/a">Go to A</a>');

})

// 매개변수 추출

app.get('/page/:id', **function** (request, response) {

// 변수를 선언합니다.

**var** name = request.params.id;

// 응답합니다.

response.send('<h1>' + name + ' Page</h1>');

});

// 전체 선택자를 사용

// router 메서드를 사용한 순서대로 요청을 확인하므로 전체 선택자를 가장 마지막에 위치해야힘

// 상위에 지정한 요청자를 제외한 모든 요청건을 처리합니다

app.all('\*', **function** (request, response) {

response.status(404).send('<h1>ERROR - Page Not Found</h1>');

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

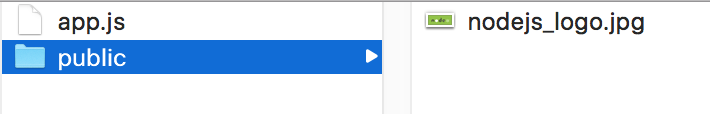
console.log('Server running at http://127.0.0.1:52273');

});

static 미들웨어

static 미들웨어는 express모듈 자체에 내장되어있으며, 웹 서버에서 손쉽게 파일을 제공하는 기능을 담당합니다.

<http://expressjs.com/en/starter/static-files.html>



// 모듈을 추출합니다.

**var** express = require('express');

// 서버를 생성합니다.

**var** app = express();

// 미들웨어를 설정합니다.

// /public폴더에 있는 내용 모두 웹서버 루트 폴더에 올림

app.use(express.static(\_\_dirname + '/public'));

app.use(**function** (request, response) {

// 응답합니다.

response.writeHead(200, { 'Content-Type': 'text/html' });

response.end('<img src="/image\_sample.jpg" width="100%" />');

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log('Server running at http://127.0.0.1:52273');

});

morgan 미들웨어

morgan 미들웨어는 웹 요청이 들어왔을 때 로그를 출력하는 미들웨어 입니다.

morgan 미들웨어는 외부 모듈이라서 따로 설치해야합니다.

<https://github.com/expressjs/morgan>

$npm install morgan

\*morgan 미들웨어의 기본형식 : <https://github.com/expressjs/morgan#predefined-formats>

// 모듈을 추출합니다.

**var** express = require('express');

**var** morgan = require('morgan');

// 서버를 생성합니다.

**var** app = express();

// 미들웨어를 설정합니다.

app.use(morgan('short'));

app.use(**function** (request, response) {

response.send('<h1>express Basic</h1>');

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log('Server running at http://127.0.0.1:52273');

});



cookie parser 미들웨어

cookie parser 미들웨어를 사용하면 request 객체와 response 객체에 cookies 속성과 cookie() 메서드가 부여됩니다.

<https://github.com/expressjs/cookie-parser?_ga=1.89646364.600263941.1488092845>

$npm install cookie-parser

// 모듈을 추출합니다.

**var** express = require('express');

**var** cookieParser = require('cookie-parser');

// 서버를 생성합니다.

**var** app = express();

// 미들웨어를 설정합니다.

app.use(cookieParser());

// 라우터를 설정합니다.

app.get('/getCookie', **function** (request, response) {

// 응답합니다.

response.send(request.cookies);

});

app.get('/setCookie', **function** (request, response) {

// 쿠키를 생성합니다.

response.cookie('string', 'cookie');

response.cookie('json', {

name: 'cookie',

property: 'delicious'

});

// 응답합니다.

response.redirect('/getCookie');

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log('Server running at http://127.0.0.1:52273');

});

body parser 미들웨어

body parser 미들웨어는 POST 요청 데이터를 추출하는 미들웨어입니다. ( application/x-www-form-urlencoded 인코딩 방식만 지원 )

body parser 미들웨어를 사용하면 request 객체에 body속성이 부여됩니다.

<https://github.com/expressjs/body-parser>

$npm install body-parser

<!--login.html-->

<!DOCTYPE html>

<html>

<head>

<title>Login Page</title>

</head>

<body>

<h1>Login Page</h1>

<hr />

<form method="post">

<table>

<tr>

<td><label>Username</label></td>

<td><input type="text" name="login" /></td>

</tr>

<tr>

<td><label>Password</label></td>

<td><input type="password" name="password" /></td>

</tr>

</table>

<input type="submit" name="" />

</form>

</body>

</html>

**var** fs = require('fs');

**var** express = require('express');

**var** cookieParser = require('cookie-parser');

**var** bodyParser = require('body-parser');

// 서버를 생성합니다.

**var** app = express();

// 미들웨어를 설정합니다.

app.use(cookieParser());

app.use(bodyParser.urlencoded({ extended: **false** })); //url인코딩이 계속 적용될지 1번만 적용할지

// 라우터를 설정합니다.

app.get('/', **function** (request, response) {

**if** (request.cookies.auth) {

response.send('<h1>Login Success</h1>');

} **else** {

response.redirect('/login');

}

});

app.get('/login', **function** (request, response) {

fs.readFile('login.html', **function** (error, data) {

response.writeHead(200, { 'Content-Type': 'text/html' });

response.end(data);

});

});

app.post('/login', **function** (request, response) {

// 쿠키를 생성합니다.

**var** login = request.body.login;

**var** password = request.body.password;

// 출력합니다.

console.log(login, password);

console.log(request.body);

// 로그인을 확인합니다.

**if** (login == 'test' && password == '1234') {

// 로그인 성공

response.cookie('auth', **true**);

response.redirect('/');

} **else** {

// 로그인 실패

response.redirect('/login');

}

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log('Server running at http://127.0.0.1:52273');

});

connect-multiparty 미들웨어

connect-multiparty 미들웨어는 multipart/form-data 인코딩방식을 제공합니다.

<https://github.com/expressjs/connect-multiparty>

$npm install connect-multiparty

<!--file\_sample.html-->

<!DOCTYPE html>

<html>

<head>

<title>Multipart Upload</title>

</head>

<body>

<h1>File Upload</h1>

<form method="post" enctype="multipart/form-data">

<table>

<tr>

<td>Comment: </td>

<td><input type="text" name="comment" /></td>

</tr>

<tr>

<td>File: </td>

<td><input type="file" name="image" /></td>

</tr>

</table>

<input type="submit" />

</form>

</body>

</html>

// 모듈을 추출합니다.

**var** fs = require('fs');

**var** express = require('express');

**var** multipart = require('connect-multiparty');

// 서버를 생성합니다.

**var** app = express();

// 미들웨어를 설정합니다.

app.use(multipart({ uploadDir: \_\_dirname + '/multipart' }));

// 라우터를 설정합니다.

app.get('/', **function** (request, response) {

fs.readFile('file\_sample.html', **function** (error, data) {

response.send(data.toString());

});

});

app.post('/', **function** (request, response) {

console.log(request.body);

console.log(request.files);

response.redirect('/');

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log('Server running at http://127.0.0.1:52273');

});

express-session 미들웨어

웹서버에서 세션처리를 하게 해주는 미들웨어 입니다.

express-session 미들웨어를 사용하면 request 객체에 session 속성을 부여합니다.

<https://github.com/expressjs/session>

$npm install express-session

// 모듈을 추출합니다.

**var** express = require('express');

**var** session = require('express-session');

// 서버를 생성합니다.

**var** app = express();

// 미들웨어를 설정합니다.

app.use(session({

name : 'session\_test', // 쿠키의 name 속성을 지정합니다

secret: 'secret key', // 비밀 키를 지정합니다

resave: **false**, //세션이 변경되지 않았어도 세션 저장소에 반영할지 설정합니다

saveUninitialized: **true**, //초기화되지 않는 세션을 세션 저장소에 저장할지 설정

cookie: { //생성할 cookie와 관련된 정보를 지정합니다.

maxAge: 60 \* 1000

}

}));

app.use(**function** (request, response) {

// 세션을 저장합니다.

request.session.now = (**new** Date()).toUTCString();

// 응답합니다.

response.send(request.session);

});

// 서버를 실행합니다.

app.listen(52273, **function** () {

console.log('Server running at http://127.0.0.1:52273');

});

2. express 프레임워크

express 프레임워크는 express 모듈기반으로 웹서비스를 개발하기 위한 여러 모듈을 제공하는  프레임워크입니다.

<http://expressjs.com/>

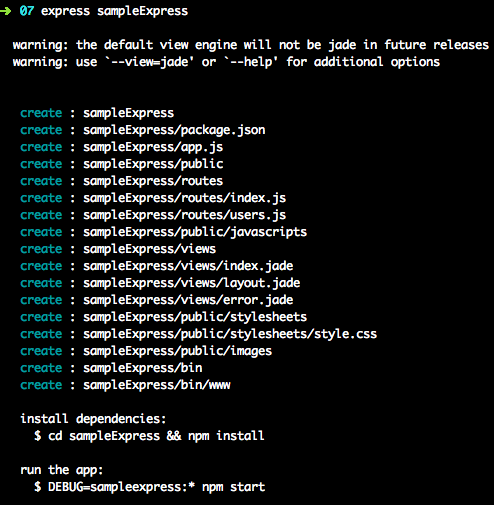
express 프레임워크 설치

express프레임워크를 설치하려면 관리자 권한이 필요합니다.

$sudo npm install -g express-generator@4

[ sudo ] password for USER: 비밀번호 입력

$express sampleExpress



express 프레임워크가 요청한대로 다음 명령을 입력하면 자동으로 express 프레임워크와 관련된 모듈이 설치가 됩니다.

$cd sampleExpress && npm install

설치가 완료된 이후 프레임워크가 안내한 실행 명령을 실행하면 웹서비스가 동작한다.

$DEBUG=sampleexpress:\* npm start

프로젝트 생성 및 설정

express 명령어를 통해서 프로젝트 생성과 설정을 지정할수있습니다. (옵션을주지않을경우 뷰템플릿엔진을 jade 을 사용한다 )

관련 명령어는 도움말을 통해 확인해봅니다.

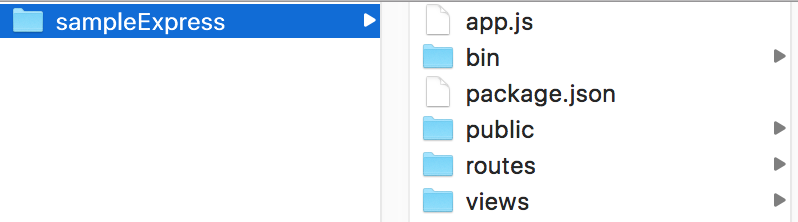
$express

* -h 또는 --help  :  도움말을 출력합니다.
* -v 또는 --vesrion : 프레임워크 버전을 출력합니다.
* -e 또는 --ejs : ejs 템플릿 엔진을 사용합니다.
* --hbs : handlebars 템플릿 엔진을 사용합니다.
* -h 또는 --hogan : hogan.js 템플릿 엔진을 사용합니다.
* -c <engine> 도는 --css <engine> : 스타일시트 엔진을 지정합니다.
* --git : .gitignore 파일을 함께 생성합니다.
* -f 또는 --force : 프로젝트가 이미 존재해도 새로 작성합니다.

$express -e --git sampleExpress

기본 프로젝트

기본 프로젝트로 express 프레임워크가 어떻게 동작하는지 살펴봅니다.



* bin 폴더는 프로그램의 실행과 관련된 파일이 들어있는 폴더입니다. 이 폴더 내부에 있는 www파일을 실행해서 프레임워크를 실행합니다.
* public 폴더는 express 모듈의 static 미들웨어를 사용해 웹서버에 올라가는 폴더입니다. 이 폴더에 자바스크립트 파일, css파일, 그림 파일등 리소스 파일을 생성합니다.
* routes 폴더는 페이지 라우트와 관련된 모듈입니다. routes 폴더에는 index.js 파일과 routes 파일이 있습니다.
* views 폴더는 ejs파일 또는 jade 파일과 같은 템플릿 파일을 저장하는 공간입니다.
* 마지막으로 app.js 파일은 프로젝트에서 중심이 되는 파일이며 package.json 파일 현재 프로젝트와 관련된 정보와 모듈을 설치하는 데 필요한 내용을 담고 있습니다.

app.js는 메인어플리케이션이 되는 파일입니다.

app.js를 통해 기본 구성을 살펴봅니다.

//외부 모듈을 추출합니다.

**var** express = require('express');

**var** path = require('path');

**var** favicon = require('serve-favicon');

**var** logger = require('morgan');

**var** cookieParser = require('cookie-parser');

**var** bodyParser = require('body-parser');

// 사용자 정의 모듈을 추출합니다.

**var** index = require('./routes/index');

**var** users = require('./routes/users');

// 서버를 생성합니다.

**var** app = express();

// 서버를 설정합니다.

app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'jade');

// 미들웨어를 설정합니다.

app.use(logger('dev'));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: **false** }));

app.use(cookieParser());

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use('/', index);

app.use('/users', users);

// 404 에러가 발생했을 때 메시지를 출력합니다

app.use(**function**(req, res, next) {

**var** err = **new** Error('Not Found');

err.status = 404;

next(err);

});

// 에러 핸들러

app.use(**function**(err, req, res, next) {

// development 환경에서의 오류를 처리합니다.

res.locals.message = err.message;

res.locals.error = req.app.get('env') === 'development' ? err : {};

// 랜더링 오류페이지

res.status(err.status || 500);

res.render('error');

});

module.exports = app;

서버 생성

app.js파일에서 가장 처음 위치하는 부분은 모듈을 추출하고, 서버 객체를 생성하는 부분입니다.

각각의 모듈을 추출하고 express()함수를 호출해서 서버 객체를 생성합니다.

//외부 모듈을 추출합니다.

**var** express = require('express');

**var** path = require('path');

**var** favicon = require('serve-favicon');

**var** logger = require('morgan');

**var** cookieParser = require('cookie-parser');

**var** bodyParser = require('body-parser');

// 사용자 정의 모듈을 추출합니다.

**var** index = require('./routes/index');

**var** users = require('./routes/users');

// 서버를 생성합니다.

**var** app = express();

미들웨어 설정

모듈을 추출하고 서버를 생성 이후에는 미들웨어를 설정하는 부분이 있습니다.

express 모듈에서 배울때 다루었던 미들웨어를 확인할수 있습니다.

// 서버를 설정합니다.

app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'jade');

// 미들웨어를 설정합니다.

app.use(logger('dev'));

app.use(bodyParser.json());

app.use(bodyParser.urlencoded({ extended: **false** }));

app.use(cookieParser());

app.use(express.static(path.join(\_\_dirname, 'public')));

app.use('/', index);

app.use('/users', users);

어플리케이션 설정

set() 메서드를 통해 express 프레임워크의 설정 옵션을 지정하는 메서드입니다.

<http://expressjs.com/ko/4x/api.html#app.set>

|  |  |
| --- | --- |
| case sensitive routing | 페이지 라우트를 할 때 대소문자를 구분할지 지정합니다. |
| env | 서버 환경을 지정합니다. |
| etag | HTTP헤더의 ETag속성을 지정합니다. |
| jsonp callback name | JSONP를 사용할 때 콜백 이름을 지정합니다. |
| json replacer | 자바스크립트 객체를 JSON 문자열로 변경할 때 사용하는 JSON.stringify의 replacer 매개변수를 지정합니다. |
| query parser | 입력된 쿼리를 파싱할 때 사용할 모듈을 지정합니다. 기본적으로는 Query String모듈입니다. |
| strict routing | 엄격 경로 확정을 사용할지 지정합니다. |
| subdomin offset | 서브 도메인을 추출하기 위해 사용할 오프셋을 지정합니다. |
| trust proxy | 신뢰할 수 있는 프록시 지정합니다. |
| views | 뷰 폴더를 지정합니다. |
| view cache | 뷰 캐시를 사용할지 지정합니다. |
| view engine | 뷰 엔진을 지정합니다. |
| x-powered-by | HTTP 헤더에 "X-Powered-By: Express"를 추가합니다. |

URL을 요청할 때 대소문자를 구분하기위해 설정옵션을 지정했고, View 폴더와 View 엔진을 지정했습니다.

// 서버를 설정합니다.

app.set('case sensitive routing', **true**); // URL요청 대소문자 구분

app.set('views', path.join(\_\_dirname, 'views'));

app.set('view engine', 'jade');

페이지 라우트

사용자가 GET방식으로 '/' URL으로 요청했을경우 routes 모듈을 제공하고, '/users' URL으로 요청했을경우 users 모듈으로 제공한다고 지정합니다.

/\* 생략 \*/

**var** index = require('./routes/index');

**var** users = require('./routes/users');

/\* 생략 \*/

app.use('/', index);

app.use('/users', users);

/\* 생략 \*/

모듈화

다른파일에서 require('./app.js') 라고 입력했을 때 서버 객체 app이 리턴이 됩니다.

module.exports = app;

페이지 레이아웃

jade 모듈 레이아웃은 express을 통해 프로젝트 생성할때 옵션을 설정하지 않을때 기본설정으로 지정이됩니다.

jade 모듈 뷰템플릿 엔진에서 기본적으로 레이아웃 페이지를 지원합니다.

ejs모듈 레이아웃

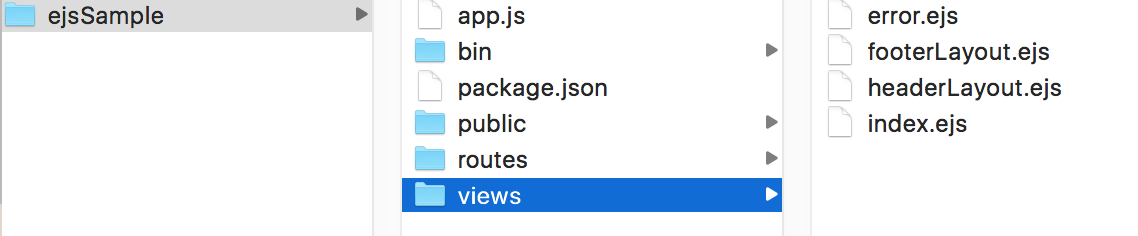
ejs모듈은 레이아웃 기능을 지원하지않습니다. 대신 include라는 방식을 사용해 레이아웃 페이지처럼 만들 수 있습니다.

$express -e ejsSample

$cd ejsSample

$npm install

**\* views 폴더구성을 아래와 같이 error.ejs, footerLayout.ejs, headerLayout.ejs, index.ejs 구성합니다.**



<!--index.ejs-->

<% **include** headerLayout.ejs %>

<h1><%= title %></h1>

<p>Welcome to <%= title %></p>

<% **include** footerLayout.ejs%>

<!--headerLayout.ejs-->

<!DOCTYPE html>

<html>

<head>

<title><%= title %></title>

<link rel='stylesheet' href='/stylesheets/style.css' />

</head>

<body>

<!--footerLayout.ejs-->

</body>

</html>

# Node.JS MySql 연동

node에서 mysql을 연동하는 방법을 해봅니다.

(\* 연동하기 전에 연동대상이 되는 mysql을 설치하고, products라는 테이블이 있다는 가정하에 작성했습니다.)

mysql를 node에서 사용하기위해서는 관련 외부 모듈이 필요합니다.

npm install mysql //모듈설치

mysql 모듈을 이용해서 mysql에 접속해서 단순 조회까지 작성해봅니다.

// 모듈을 추출합니다.  
**var** mysql = require('mysql');  
  
// 데이터베이스와 연결합니다.  
**var** client = mysql.createConnection({  
 user: 'root',  
 password: 'rkswlsla',  
 database : 'Nodedata'  
});  
  
// 데이터베이스 쿼리를 사용합니다.  
client.query('SELECT \* FROM products', **function**(error, result, fields) {  
 **if**(error) {  
 console.log('쿼리 문장에 오류가 있습니다.');  
 } **else** {  
 console.log(result);  
 }  
});

데이터 베이스에 접속할때 createConnection(option) 메서드를 사용합니다. ( user, password 속성은 필수적으로 입력해야합니다. )

createConnection메서드에서 사용할수 있는 속성은 <https://github.com/mysqljs/mysql#connection-options> 에서 추가적으로 확인할수 있습니다.

이제 mysql모듈을 사용해서 CRUD를 구현해봅니다.

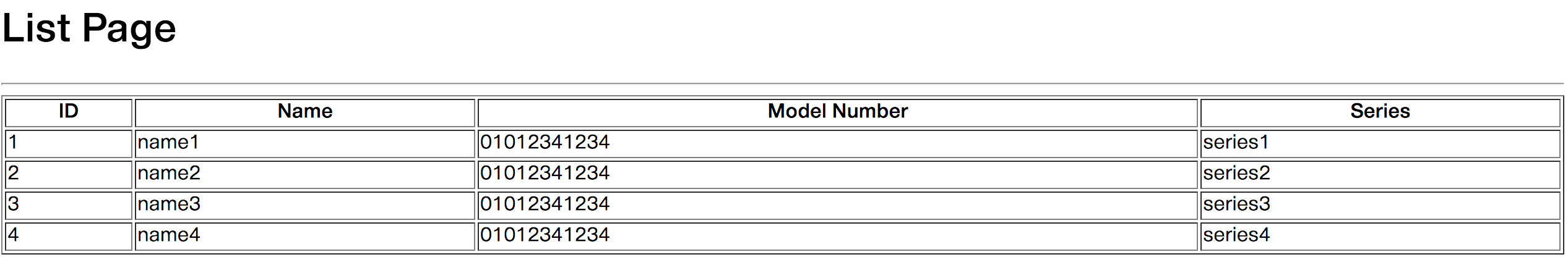
우선 DB에있는 데이터를 화면에 list하는 영역까지 작성해봅니다.

// 모듈을 추출합니다.  
**var** fs = require('fs');  
**var** ejs = require('ejs');  
**var** mysql = require('mysql');  
**var** express = require('express');  
**var** bodyParser = require('body-parser');  
  
// 데이터베이스와 연결합니다.  
**var** client = mysql.createConnection({  
 user: 'root',  
 password: 'rkswlsla',  
 database: 'Nodedata'  
});  
  
// 서버를 생성합니다.  
**var** app = express();  
app.use(bodyParser.urlencoded({  
 extended: **false**}));  
  
// 서버를 실행합니다.  
app.listen(52273, **function** () {  
 console.log('server running at http://127.0.0.1:52273');  
});  
  
// 라우트를 수행합니다.  
app.get('/', **function** (request, response) {  
 // 파일을 읽습니다.  
 fs.readFile('list.html', 'utf8', **function** (error, data) {  
 // 데이터베이스 쿼리를 실행합니다.  
 client.query('SELECT \* FROM products', **function** (error, results) {  
 // 응답합니다.  
 response.send(ejs.render(data, {  
 data: results  
 }));  
 });  
 });  
});

리스트 HTML 화면를 작성해봅니다.

<!DOCTYPE html>  
<html>  
<head>  
 <title>List Page</title>  
</head>  
<body>  
 <h1>List Page</h1>  
 <hr />  
 <table width="100%" border="1">  
 <tr>  
 <th>ID</th>  
 <th>Name</th>  
 <th>Model Number</th>  
 <th>Series</th>  
 </tr>  
 <% data.forEach(function (item, index) { %>  
 <tr>  
 <td><%= item.id %></td>  
 <td><%= item.name %></td>  
 <td><%= item.modelnumber %></td>  
 <td><%= item.series %></td>  
 </tr>  
 <% }); %>  
 </table>  
</body>  
</html>

\*실행결과



작성된 코드를 실행해보면, DB에서 가져온 데이터를 HTML화면에서 리스팅해서 보여주는 코드를 작성했습니다.

그럼 이제 데이터를 저장하는 로직을 작성해봅니다.

우선, 저장하는 페이지으로 이동하는 URL와 저장페이지에서 입력값을 받아서 DB에 insert, 정상적으로 처리가 되면 list 화면으로 이동하는 로직을 작성해봅니다.

app.get('/insert', **function** (request, response) {  
 // 파일을 읽습니다.  
 fs.readFile('insert.html', 'utf8', **function** (error, data) {  
 // 응답합니다.  
 response.send(data);  
 });  
});  
  
app.post('/insert', **function** (request, response) {  
 // 변수를 선언합니다.  
 **var** body = request.body;  
 // 데이터베이스 쿼리를 실행합니다.  
 client.query('INSERT INTO products (name, modelnumber, series) VALUES (?, ?, ?)', [  
 body.name, body.modelnumber, body.series  
 ], **function** () {  
 // 응답합니다.  
 response.redirect('/');  
 });  
});

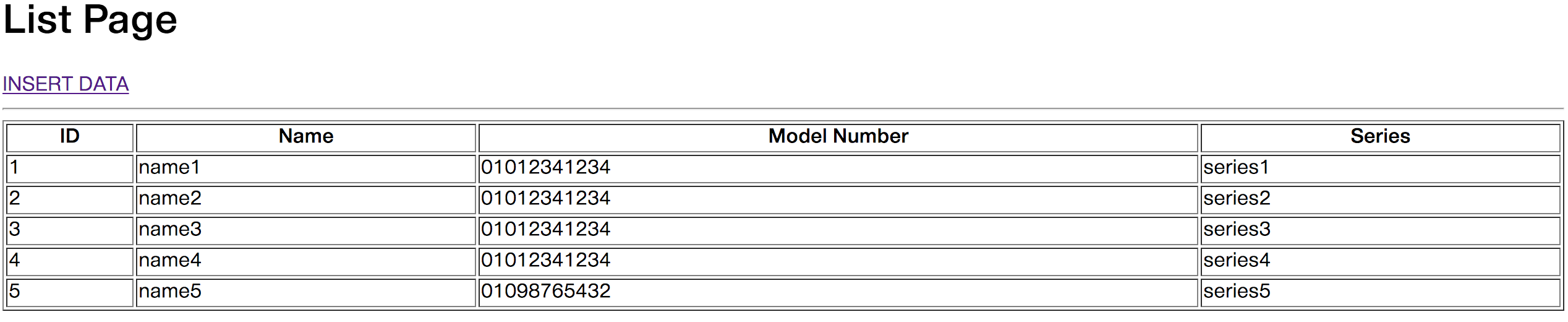
기존 list.html에서 insert 링크를 걸어줍니다.

<h1>List Page</h1>  
<a href="/insert">INSERT DATA</a>  
<hr />

insert 페이지를 작성해봅니다.

<!DOCTYPE html>  
<html>  
<head>  
 <title>Insert Page</title>  
</head>  
<body>  
 <h1>Insert Page</h1>  
 <hr />  
 <form method="post">  
 <fieldset>  
 <legend>INSERT DATA</legend>  
 <table>  
 <tr>  
 <td><label>Name</label></td>  
 <td><input type="text" name="name" /></td>  
 </tr>  
 <tr>  
 <td><label>Model Number</label></td>  
 <td><input type="text" name="modelnumber" /></td>  
 </tr>  
 <tr>  
 <td><label>Series</label></td>  
 <td><input type="text" name="series" /></td>  
 </tr>  
 </table>  
 <input type="submit" />  
 </fieldset>  
 </form>  
</body>  
</html>

\*결과



이제는 저장한 데이터를 수정하는 로직을 작성해봅니다.

수정하는 로직은 저장하는 로직과 거의 유사합니다.

app.get('/edit/:id', **function** (request, response) {  
 // 파일을 읽습니다.  
 fs.readFile('edit.html', 'utf8', **function** (error, data) {  
 // 데이터베이스 쿼리를 실행합니다.  
 client.query('SELECT \* FROM products WHERE id = ?', [  
 request.params.id  
 ], **function** (error, result) {  
 // 응답합니다.  
 response.send(ejs.render(data, {  
 data: result[0]  
 }));  
 });  
 });  
});  
  
app.post('/edit/:id', **function** (request, response) {  
 // 변수를 선언합니다.  
 **var** body = request.body;  
 // 데이터베이스 쿼리를 실행합니다.  
 client.query('UPDATE products SET name=?, modelnumber=?, series=? WHERE id=?'

        , [body.name, body.modelnumber, body.series, request.params.id], **function** () {  
 // 응답합니다.  
 response.redirect('/');  
 });  
});

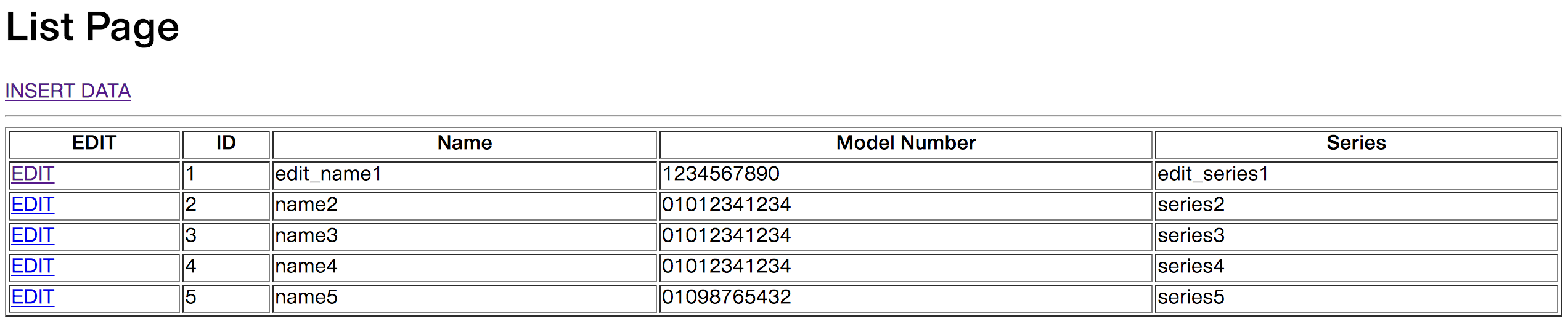
\* list html에 수정링크를 추가합니다.

<tr>  
 <td><a href="/edit/<%= item.id %>">EDIT</a></td>  
 <td><%= item.id %></td>  
 <td><%= item.name %></td>  
 <td><%= item.modelnumber %></td>  
 <td><%= item.series %></td>  
</tr>

\* edit html을 신규로 생성합니다.

<!DOCTYPE html>  
<html>  
<head>  
 <title>Edit Page</title>  
</head>  
<body>  
 <h1>Edit Page</h1>  
 <hr />  
 <form method="post">  
 <fieldset>  
 <legend>Edit Data</legend>  
 <table>  
 <tr>  
 <td><label>Id</label></td>  
 <td><input type="text" name="id" value="<%= data.id %>" disabled /></td>  
 </tr>  
 <tr>  
 <td><label>Name</label></td>  
 <td><input type="text" name="name" value="<%= data.name %>" /></td>  
 </tr>  
 <tr>  
 <td><label>Model Number</label></td>  
 <td>  
 <input type="text" name="modelnumber"  
 value="<%= data.modelnumber %>" />  
 </td>  
 </tr>  
 <tr>  
 <td><label>Series</label></td>  
 <td><input type="text" name="series" value="<%= data.series %>" /></td>  
 </tr>  
 </table>  
 <input type="submit" />  
 </fieldset>  
 </form>  
</body>  
</html>

\*결과

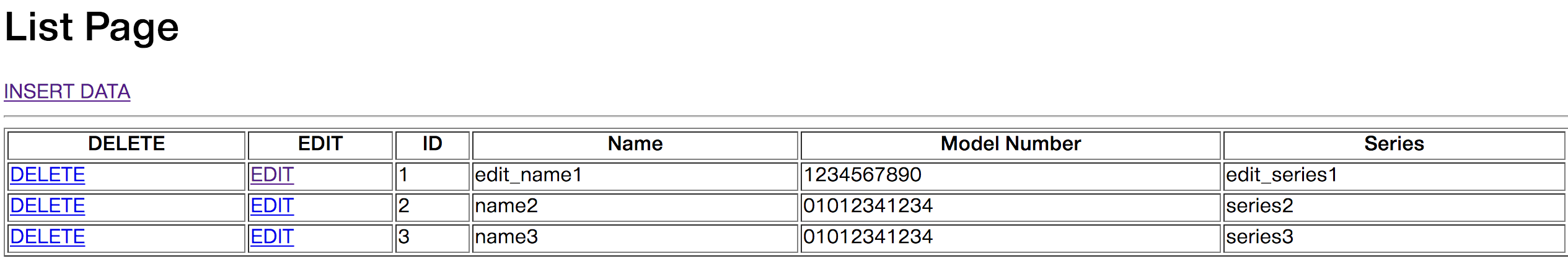


마지막으로 삭제하는 로직을 추가해봅니다.

app.get('/delete/:id', **function** (request, response) {  
 // 데이터베이스 쿼리를 실행합니다.  
 client.query('DELETE FROM products WHERE id=?', [request.params.id], **function** () {  
 // 응답합니다.  
 response.redirect('/');  
 });  
});

<td><a href="/delete/<%= item.id %>">DELETE</a></td>

\* 결과



# Node.JS Socket.io 모듈

Socket.io 모듈은 웹 소켓서버를 쉽게 구현할 수 있게 도와주는 모듈입니다.

원래 웹소켓은 html5부터 지원하는데 인터넷 익스플로러 8이하에서는 웹소켓을 사용할수 없다.

하지만 Socket.io모듈은 자체적으로 웹소켓이 가능하면 웹소켓으로 불가능한 경우 롱 폴링을 이용해서 통신을 사용할수 있게 해줍니다.

즉 Socket.io 모듈은 모든 푸시 메시지를 환경과 무관하게 전달하도록 자체적으로 구현되어있다.

<https://github.com/socketio/socket.io>

### socket.io모듈 기본

$npm install socket.io

**\* 웹소켓 서버 실행**

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** socketio = require('socket.io');  
  
// 웹 서버를 생성합니다.  
**var** server = http.createServer(**function** (request, response) {  
 // sample\_html.html 파일을 읽습니다.  
 fs.readFile('sample\_html.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
}).listen(52273, **function** () {  
 console.log('Server running at http://127.0.0.1:52273');  
});  
  
// 소켓 서버를 생성 및 실행합니다.  
**var** io = socketio.listen(server);

<!-- sample\_html -->  
<!DOCTYPE html>  
<html>  
<head>  
 <script src="/socket.io/socket.io.js"></script>  
 <script>  
 window.onload = **function** () {  
 //소켓을 연결합니다.  
 **var** socket = io.connect();  
 };  
 </script>  
</head>  
<body>  
</body>  
</html>

서버를 실행해보면 소켓연결에대해 아무런 정보가 노출이 안되는데, socket.io 모듈에서 디버그 모드를 활성화하면 웹소켓과 관련한 내용이 출력이됩니다.

$DEBUG=socket.io\* node sample

### 웹 소켓 이벤트

서버와 클라이언트 사이에 데이터를 교환할 때 이벤트를 사용합니다.

socket.io 모듈의 이벤트는 아래와 같이 사용합니다만, 대게는 사용자 정의 이벤트를 굉장히 많이 사용합니다.

**connection : 클라이언트가 연결할 때 발생합니다.**

**disconnect : 클라이언트가 연결을 해체할 때 발생합니다.**

**socket.io 모듈은 on() 메서드로 이벤트를 생성하고, emit() 메서드로 이벤트를 발생시킵니다.**

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** socketio = require('socket.io');  
  
// 웹 서버를 생성합니다.  
**var** server = http.createServer(**function** (request, response) {  
 // sample\_html.html 파일을 읽습니다.  
 fs.readFile('sample\_html.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
}).listen(52273, **function** () {  
 console.log('Server running at http://127.0.0.1:52273');  
});  
  
// 소켓 서버를 생성 및 실행합니다.  
**var** io = socketio.listen(server);  
io.sockets.on('connection', **function** (socket) {  
 // ev1 이벤트  
 socket.on('ev1', **function** (data) {  
 // 클라이언트가 전송한 데이터를 출력합니다.  
 console.log('Client Send Data:', data);  
  
 // 클라이언트에 smart 이벤트를 발생시킵니다.  
 socket.emit('ev2', data);  
 });  
});

<!DOCTYPE html>  
<html>  
<head>  
 <script src="/socket.io/socket.io.js"></script>  
 <script>  
 window.onload = **function** () {  
 // 소켓을 생성합니다.  
 **var** socket = io.connect();  
  
 // 소켓 이벤트를 연결합니다.  
 socket.on('ev2', **function** (data) {  
 alert(data);  
 });  
  
 // 문서 객체 이벤트를 연결합니다.  
 document.getElementById('button').onclick = **function** () {  
 // 변수를 선언합니다.  
 **var** text = document.getElementById('text').value;  
  
 // 데이터를 전송합니다.  
 socket.emit('ev1', text);  
 };  
 };  
 </script>  
</head>  
<body>  
<input type="text" id="text" />  
<input type="button" id="button" value="echo" />  
</body>  
</html>

### 소켓 통신 종류

socket.io 모듈이 생성할 수 있는 소켓통신방법은 3가지로 나눌수있습니다.

public : 자신을 포함한 모든 클라이언트에 데이터를 전달합니다.

broadcast : 자신을 제외한 모든 클라이언트에 데이터를 전달합니다.

private : 특정 클라이언트에 데이터를 전달합니다.

##### **public 통신**

public 통신은 클라이언트가 이벤트를 발생시키면 서버에서 모든 클라이언트에게 이벤트를 발생시키는 방식입니다.

io.sockets 객체의 emit() 메서드를 사용하면 현재 접속한 모든 사용자에게 메시지를 보낼수 있습니다.

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** socketio = require('socket.io');  
  
// 웹 서버를 생성합니다.  
**var** server = http.createServer(**function** (request, response) {  
 // sample\_html.html 파일을 읽습니다.  
 fs.readFile('sample\_html.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
}).listen(52273, **function** () {  
 console.log('Server running at http://127.0.0.1:52273');  
});  
  
// 소켓 서버를 생성 및 실행합니다.  
**var** io = socketio.listen(server);  
io.sockets.on('connection', **function** (socket) {  
 socket.on('ev1', **function** (data) {  
 // public 통신  
 io.sockets.emit('ev2', data);  
 });  
});

##### **broadcast 통신**

broadcast 통신은 public통신과 비슷하지만 소켓이벤트를 발생한 당사자에게는 이벤트를 전달하지않습니다.

socket객체의 broadcast 속성을 사용합니다. broadcast객체의 emit() 메서드를 사용하면 자신을 제외한 모든 사용자에게 이벤트를 전달합니다.

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** socketio = require('socket.io');  
  
// 웹 서버를 생성합니다.  
**var** server = http.createServer(**function** (request, response) {  
 // sample\_html.html 파일을 읽습니다.  
 fs.readFile('sample\_html.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
}).listen(52273, **function** () {  
 console.log('Server running at http://127.0.0.1:52273');  
});  
  
// 소켓 서버를 생성 및 실행합니다.  
**var** io = socketio.listen(server);  
io.sockets.on('connection', **function** (socket) {  
 socket.on('ev1', **function** (data) {  
 // public 통신  
 socket.broadcast.emit('ev2', data);  
 });  
});

##### **private 통신**

private통신은 소켓 이벤트를 발생한 당사자가 선택된 특정 대상에게만 데이터를 전달하는 방법을 뜻합니다. ( 채팅에서 귓속말을 하는경우가 이에 해당합니다. )

sockets 객체의 in(),to() 메서드에 대상 유니크값을 넣고 emit() 메서드를 사용하면 유니크값에 해당하는 유저에게 메시지를 발송합니다.

가장최근에 접속한 클라이언트 한명에게 데이터를 전달하는 예제입니다.

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** socketio = require('socket.io');  
  
// 웹 서버를 생성합니다.  
**var** server = http.createServer(**function** (request, response) {  
 // sample\_html.html 파일을 읽습니다.  
 fs.readFile('sample\_html.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
}).listen(52273, **function** () {  
 console.log('Server running at http://127.0.0.1:52273');  
});  
  
// 소켓 서버를 생성 및 실행합니다.  
**var** id = 0;  
**var** io = socketio.listen(server);  
io.sockets.on('connection', **function** (socket) {  
  
 console.info("this.id : " +id);  
 console.info("socket.id : " +socket.id);  
 // id를 설정합니다.  
 id = socket.id;  
  
 // rint 이벤트  
 socket.on('ev1', **function** (data) {  
 // private 통신  
 io.sockets.to(id).emit('ev2', data);  
 });  
});

### 방 생성

socket.io 모듈은 방(grop) 을 생성하는 기능을 포함하고 있습니다.

방을 생성할 때는 아래와 같은 메서드를 사용합니다.

socket.join() : 클라이언트를 방에 집에 넣습니다.

io.sockets.in() / io.sockets.to() : 특정 방에 있는 클라이언트를 추출합니다. ( in(), to() 메서드는 완전히 같은 메서드입니다. )

// 모듈을 추출합니다.  
**var** fs = require('fs');  
  
// 서버를 생성합니다.  
**var** server = require('http').createServer();  
**var** io = require('socket.io').listen(server);  
  
// 서버를 실행합니다.  
server.listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});  
  
// 웹 서버 이벤트를 연결합니다.  
server.on('request', **function** (request, response) {  
 // sample\_html.html 파일을 읽습니다.  
 fs.readFile('sample\_html.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
});  
  
// 방 이름을 저장할 변수  
**var** roomName = 1;  
// 소켓 서버 이벤트를 연결합니다.  
io.sockets.on('connection', **function** (socket) {  
 // if(io.nsps['/'].adapter.rooms["room-"+roomName] && io.nsps['/'].adapter.rooms["room-"+roomName].length > 1){  
 // roomName++;  
 // }  
 socket.join("room-"+roomName);  
  
 //특정룸에 메시지전달  
 socket.on('test', **function** (data) {  
 io.sockets.in("room-"+roomName).emit('message', "You are in room no. "+roomName);  
 });  
});

<!--sample\_html-->  
<!DOCTYPE html>  
<html>  
<head>  
 <meta charset="utf8" />  
 <script src="https://code.jquery.com/jquery-1.12.4.js"></script>  
 <script src="/socket.io/socket.io.js"></script>  
 <script>  
 window.onload = **function** () {  
 // 변수를 선언합니다.  
 **var** socket = io.connect();  
 socket.on('message', **function** (data) {  
 $('<p>' + data + '</p>').appendTo('body');  
 });  
 // 문서 객체 이벤트를 연결합니다.  
 document.getElementById('button').onclick = **function** () {  
 socket.emit('test', '');  
 };  
  
 };  
 </script>  
</head>  
<body>  
<button id="button">EMIT</button>  
</body>  
</html>

join 이벤트는 클라이언트가 특정한 방에 접속하게 만드는 이벤트입니다. 이때 방 이름을 함께 저장합니다.

message 이벤트는 같은 방에 속한 클라이언트에 메시지를 전달하는 이벤트입니다.

웹 채팅 프로그램

socket.io 모듈을 활용해 웹채팅 프로그램을 만들어봅니다.

웹 채팅 서버가 사용하는 socket.io 모듈은 2가지 이벤트로 구현합니다.

첫번째 - 클라이언트에서 메시지를 입력했을경우 서버로 message 이벤트와 함께 데이터를 전달합니다.

두번째 - 서버는 message 이벤트가 실행이되면 클라이언트에 받은 데이터를 현재 접속해 있는 모든 클라이언트에 보냅니다.

// 모듈을 추출합니다.  
**var** http = require('http');  
**var** fs = require('fs');  
**var** socketio = require('socket.io');  
  
// 웹 서버를 생성합니다.  
**var** server = http.createServer(**function** (request, response) {  
 // sample\_html.html 파일을 읽습니다.  
 fs.readFile('sample\_html.html', **function** (error, data) {  
 response.writeHead(200, { 'Content-Type': 'text/html' });  
 response.end(data);  
 });  
}).listen(52273, **function** () {  
 console.log('Server running at http://127.0.0.1:52273');  
});  
  
// 소켓 서버를 생성 및 실행합니다.  
**var** io = socketio.listen(server);  
io.sockets.on('connection', **function** (socket) {  
 // message 이벤트  
 socket.on('message', **function** (data) {  
 io.sockets.emit('message', data);  
 });  
});

<!--sample\_html-->  
<!DOCTYPE html>  
<html>  
<head>  
 <title>Web chat</title>  
 <script src="https://code.jquery.com/jquery-1.12.4.js"></script>  
 <script src="/socket.io/socket.io.js"></script>  
 <script>  
 //HTML 문서가 모두 준비되면  
 $(document).ready(**function** () {  
 //변수를 선언합니다.  
 **var** socket = io.connect();  
 // 이벤트를 연결합니다  
 socket.on('message', **function** (data){  
 //추가할 문자열을 만듭니다.  
 **var** output = '';  
 output += '<li>';  
 output += ' <h3>' + data.name + '</h3>';  
 output += ' <p>' + data.message + '</p>';  
 output += ' <p>' + data.date + '</p>';  
 output += '</li>';  
 //문서 객체를 추가합니다.  
 $(output).prependTo('#content');  
 });  
 //버튼을 클릭할때  
 $('button').click(**function** () {  
 socket.emit('message',{  
 name : $('#name').val(),  
 message : $('#message').val(),  
 date : **new** Date().toUTCString()  
 });  
 });  
 });  
 </script>  
</head>  
<body>  
 <h1>Socket.io Chat</h1>  
 <p>Chat With Node.js</p>  
 <hr>  
 <input id="name"/>  
 <input id="message"/>  
 <button>Button</button>  
 <ul id="content">  
  
 </ul>  
</body>  
</html>

Node.JS 영화예매 프로그램

프로젝트 개요

실시간 영화 예매 어플리케이션을 만들 것입니다. Ajax를 사용해 화면에 현재 좌석을 출력하고 socket.io 모듈로 좌석 예약 상황을 실시간으로 전달하는 기능입니다.

서버코드 작성

서버구현을 시작합니다. 웹 서버와 소켓 서버를 생성하고, 변수 seats는 배열형식으로 공연장 좌석 정보를 저장합니다.

변수 seats에서 숫자 0은 빈 공간, 숫자 1은 예약가능한 좌석, 숫자2는 예약이 완료된 좌석으로 구분할 것입니다.

웹서버에서는 "/" 경로로 유입이 될경우 sample.html 파일로 연결하고, "/seats" 경로로 유입이될경우 Ajax를 사용해 변수 seats를 화면에 출력할 JSON데이터를 제공합

// 모듈을 추출합니다.  
**var** socketio = require('socket.io');  
**var** express = require('express');  
**var** http = require('http');  
**var** fs = require('fs');  
  
// 변수를 선언합니다.  
**var** seats = [  
 [1, 1, 0, 1, 1, 0, 0, 0, 0, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
 [1, 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1],  
];  
  
// 웹 서버를 생성합니다.  
**var** app = express();  
**var** server = http.createServer(app);  
  
// 라우트를 수행합니다.  
app.get('/', **function** (request, response, next) {  
 fs.readFile('sample.html', **function** (error, data) {  
 response.send(data.toString());  
 });  
});  
app.get('/seats', **function** (request, response, next) {  
 response.send(seats);  
});  
  
// 웹 서버를 실행합니다.  
server.listen(52273, **function** () {  
 console.log('Server Running at http://127.0.0.1:52273');  
});  
  
// 소켓 서버를 생성 및 실행합니다.  
**var** io = socketio.listen(server);  
io.sockets.on('connection', **function** (socket) {  
 socket.on('reserve', **function** (data) {  
 seats[data.y][data.x] = 2;  
 io.sockets.emit('reserve', data);  
 });  
});

클라이언트 페이지 코드 작성

우선 첫 번째 css와 scripte 태그를 작성하고, socket 객체를 생성합니다.

<!DOCTYPE html>  
<html>  
<head>  
 <title>Node.js Programming for Modern Web</title>  
 <style>  
 .line {  
 overflow: hidden;  
 }  
  
 .seat {  
 margin: 2px;  
 float: left;  
 width: 30px;  
 height: 30px;  
 border-radius: 3px;  
 }  
  
 .enable {  
 background: gray;  
 }  
  
 .enable:hover {  
 background: black;  
 }  
  
 .disable {  
 background: red;  
 }  
 </style>  
 <script src="http://code.jquery.com/jquery-1.12.1.js"></script>  
 <script src="/socket.io/socket.io.js"></script>  
 <!-- 소켓 생성 -->  
 <script>  
 // 소켓 이벤트를 수행합니다.  
 **var** socket = io.connect();  
 </script>  
 <!-- 초기 좌석 생성 -->  
   
 </script>  
</head>  
<body>  
  
</body>  
</html>

초기좌석 생성을 작성하도록합니다.

좌석을 생성할때는 Ajax로 "/seats"에 위치하는 변수 seats의 JSON 파일 가져와 구성합니다.

dummy 객체를 서버에 전달한 이유는 인터넷 익스플로러 때문입니다. 인터넷 익스플로러는 같은 URL에 연속적으로 요청이 발생하면 이전에 불려온 데이터를 제공합니다.

따라서 현재 시간을 기반으로 URL을 매번 다른 형태로 전달하도록 하

// Ajax를 수행합니다.  
$.getJSON('/seats', { dummy: **new** Date().getTime() }, **function** (data) {  
   
});

'/seats' 에서 가져온 data는 2차원 배열이므로 반복문을 2번 사용해서 X위치와 Y위치에 해당하는 요소를 선택할수 있습니다.

// Ajax를 수행합니다.  
$.getJSON('/seats', { dummy: **new** Date().getTime() }, **function** (data) {  
 // 좌석을 생성합니다.  
 $.each(data, **function** (indexY, line) {  
 // 문서 객체를 생성합니다.  
 **var** $line = $('<div></div>').addClass('line');  
 $.each(line, **function** (indexX, seat) {  
 // 문서 객체를 생성하고 변수 $line에 추가합니다.  
 **var** $output = $('<div></div>', {  
 'class': 'seat',  
 'data-x': indexX,  
 'data-y': indexY  
 }).appendTo($line);  
 **if** (seat == 1) {  
 // 좌석이 비어 있으면 enable 클래스와 click 리스너를 추가합니다.  
 $output.addClass('enable').on('click', onClickSeat);  
 } **else if** (seat == 2) {  
 // 좌석이 사용 불가능하면 disable 클래스를 추가합니다.  
 $output.addClass('disable');  
 }  
 });  
 // 문서 객체를 추가합니다.  
 $line.appendTo('body');  
 });  
});

지금까지 좌석 구성을 완료했습니다.

이제 좌석을 클릭하면 이벤트를 발생하게 해서, 좌석의 X,Y 좌표를 추출해서 서버에 전송합니다.

$(document).ready(**function** () {  
 // 변수를 선언합니다.  
 **var** onClickSeat = **function** () {  
 **var** x = $(**this**).attr('data-x');  
 **var** y = $(**this**).attr('data-y');  
 **if** (confirm('좌석을 예약하시겠습니까?')) {  
 $(**this**).off('click');  
 socket.emit('reserve', {  
 x: x,  
 y: y  
 });  
 } **else** {  
 alert('취소되었습니다.');  
 }  
 };  
 // Ajax를 수행합니다.  
 $.getJSON('/seats', { dummy: **new** Date().getTime() }, **function** (data) {  
 // 생략  
   
 });  
});

서버로 좌석정보를 전송하면, 변수 seats를 변경하고 모든 클라이언트에 예약된 좌석과 관련된 정보를 전달합니다.

그리고 좌석의 클래스를 변경하고 이벤트 리스너를 제거합니다. ( 'reserve'는 서버js에서 설정한 이벤트입니다. )

<!-- 소켓 생성 -->  
<script>  
 // 소켓 이벤트를 수행합니다.  
 **var** socket = io.connect();  
  
 // 이벤트를 연결합니다.  
 socket.on('reserve', **function** (data) {  
 **var** $target = $('div[data-x = ' + data.x + '][data-y = ' + data.y + ']');  
 $target.removeClass('enable');  
 $target.addClass('disable');  
 });  
</script>

전체코드

<!DOCTYPE html>  
<html>  
<head>  
 <title>Node.js Programming for Modern Web</title>  
 <style>  
 .line {  
 overflow: hidden;  
 }  
  
 .seat {  
 margin: 2px;  
 float: left;  
 width: 30px;  
 height: 30px;  
 border-radius: 3px;  
 }  
  
 .enable {  
 background: gray;  
 }  
  
 .enable:hover {  
 background: black;  
 }  
  
 .disable {  
 background: red;  
 }  
 </style>  
 <script src="http://code.jquery.com/jquery-1.12.1.js"></script>  
 <script src="/socket.io/socket.io.js"></script>  
 <!-- 소켓 생성 -->  
 <script>  
 // 소켓 이벤트를 수행합니다.  
 **var** socket = io.connect();  
  
 // 이벤트를 연결합니다.  
 socket.on('reserve', **function** (data) {  
 **var** $target = $('div[data-x = ' + data.x + '][data-y = ' + data.y + ']');  
 $target.removeClass('enable');  
 $target.addClass('disable');  
 });  
 </script>  
 <!-- 초기 좌석 생성 -->  
 <script>  
 $(document).ready(**function** () {  
 // 변수를 선언합니다.  
 **var** onClickSeat = **function** () {  
 **var** x = $(**this**).attr('data-x');  
 **var** y = $(**this**).attr('data-y');  
 **if** (confirm('좌석을 예약하시겠습니까?')) {  
 $(**this**).off('click');  
 socket.emit('reserve', {  
 x: x,  
 y: y  
 });  
 } **else** {  
 alert('취소되었습니다.');  
 }  
 };  
 // Ajax를 수행합니다.  
 $.getJSON('/seats', { dummy: **new** Date().getTime() }, **function** (data) {  
 // 좌석을 생성합니다.  
 $.each(data, **function** (indexY, line) {  
 // 문서 객체를 생성합니다.  
 **var** $line = $('<div></div>').addClass('line');  
 $.each(line, **function** (indexX, seat) {  
 // 문서 객체를 생성하고 변수 $line에 추가합니다.  
 **var** $output = $('<div></div>', {  
 'class': 'seat',  
 'data-x': indexX,  
 'data-y': indexY  
 }).appendTo($line);  
 **if** (seat == 1) {  
 // 좌석이 비어 있으면 enable 클래스와 click 리스너를 추가합니다.  
 $output.addClass('enable').on('click', onClickSeat);  
 } **else if** (seat == 2) {  
 // 좌석이 사용 불가능하면 disable 클래스를 추가합니다.  
 $output.addClass('disable');  
 }  
 });  
 // 문서 객체를 추가합니다.  
 $line.appendTo('body');  
 });  
 });  
 });  
 </script>  
</head>  
<body>  
  
</body>  
</html>