based Node v4.2.3

softcontext@gmail.com

Chapter 1. Node 프로파일

Node.js 정의

노드는 자바스크립트를 활용하여 **Non-blocking I/O** 와 **단일 스레드 이벤트 루프**를 통한 높은 처리 성능을 가지고 있는 **서버 사이드 개발**에 적합한 네트워크 어플리케이션 소프트웨어 **크로스플랫폼**이다.

Node.js is an **open-source**, cross-platform runtime environment for developing server-side web applications.

Node.js applications are written in JavaScript and can be run within the Node.js runtime on a wide variety of platforms, including OS X, Microsoft Windows, Linux, FreeBSD, NonStop, IBM AIX, IBM System z and IBM i. Its work is hosted and **supported by the Node.js Foundation**, **a collaborative project at the Linux Foundation.**

Node.js provides an **event-driven architecture** and a non-blocking I/O API designed to optimize an application's throughput and scalability for real-time web applications.

It uses **Google V8 JavaScript engine** to execute code, and a large percentage of the basic modules are written in JavaScript. Node.js contains a built-in library to allow applications to act as a **stand-alone web server.**

자바스크립트는 모든 주요 브라우저가 지원하는 유일한 프로그래밍 언어다.
노드를 사용한다면 개발자 입장에서 클라이언트 쪽과 서버 쪽 언어가 자바스크립트 하나로 통일된다는 점에서 이득이 있다.

노드는 내장 HTTP 서버 라이브러리를 포함하고 있어서 웹 서버에서 단독으로 동작하는 것이 가능하다.
대부분의 자바스크립트가 웹 브라우저에서 실행되는 것과는 달리 노드는 서버 측에서 실행된다는 점이 가장 큰 차이점이다. 노드는 일부 **CommonJS** 명세를 구현하고 있으며 쌍방향 테스트를 위해 **REPL** 환경을 포함하고 있다.

- 노드는 서버 사이드 JavaScript 이다.

일반적으로 사용해왔던 JavaScript 는 브라우저안에 의해서 구동되지만 노드는 브라우저 밖, 백앤드에서 JavaScript 를 실행할 수 있게 해준다.

- **노드는 JavaScript 를 해석하고 실행한다.**

  노드는 크롬 브라우저가 사용하는 JavaScript 실행환경인 구글의 V8 가상머신을 사용한다.

- **노드는 서버사이드 JavaScript 실행환경과 라이브러리로 이루어져 있다.**

  노드는 관심사가 다른 코드들을 모듈로 분리하여 관리성의 증대를 꾀한다. 노드 모듈의 개수는 이미 20 만개를 넘었으며 가파르게 증가하고 있다.

  노드 모듈 통계 사이트 : http://www.modulecounts.com/

REPL

Read Eval Print Loop 의 약자로써 현재 사용되고들 있는 스크립팅언어들의 interactive interpreter shell 의 다른 명칭이다. REPL 은 Lisp 나 Scheme 에서 개발자가 직접 간단한 코드를 직접 입력하여 바로 결과값을 확인할 수 있도록 지원하는 툴이다.

CommonJS

노드는 CommonJS 라고 하는 스펙을 따르고 있다. CommonJS 는 특정 코드가 아니라, 웹브라우저 외 서버사이드나 독립적인 애플리케이션으로 사용하기 위해 필요한 기능을 정의한 스펙이다. module, exports, require 지시어들은 CommonJS 의 대표적인 적용 예이다.

V8 JavaScript Engine

V8 자바스크립트 엔진은 구글에서 개발된 오픈 소스로써 JIT 가상 머신형식의 자바스크립트 엔진이며 구글 크롬 브라우저와 안드로이드 브라우저에 탑재되어 있다. ECMAScript 3rd Edition 규격의 C++로 작성되었으며 독립적으로 실행이 가능하다. 또한 C++로 작성된 응용 프로그램의 일부로도 작동할 수 있다.

V8 은 자바스크립트를 바이트코드로 컴파일하거나 인터프리트하는 대신 **실행하기 전 기계어(x86, ARM, 또는 MIPS)로 컴파일**하여 성능을 향상시킨다. 추가적인 속도향상을 위해 인라인 캐싱(inline caching)과 같은 최적화 기법을 사용한다.

V8 엔진은 **컴파일된 기계어 코드를 캐시한다**. 동일한 로직을 많이 수행할 수 밖에 없는 서버사이드에서는 효율적이다. UI 를 가지면서 유저와 상호작용해야 하는 프로그램은 유저가 언제 이벤트를 발생 시킬지 모르기 때문에 이벤트드리븐 모델 사용이 일반적이다.

V8 과 그 위에서 동작하는 서버사이드 javascript 코드는 모두 event loop thread 에서 동작한다. 그리고 이 thread 가 모든 클라이언트 요청을 처리한다. Blocking I/O 작업을 해야하는 리소스에 대해서는 thread

pool 을 이용해 노드가 event loop thread 가 아닌 별도의 thread 에 그 일을 위임한다.

blocking I/O 작업을 하는 thread 의 작업이 끝나면, event loop thread 가 이를 인지하고 v8 을 통해 해당 리소스에 매핑되어 있는 javascript 콜백을 바로 호출한다. 즉, event loop thread 에서 수행되는 javascript 컨텍스트에선 시스템 리소스 접근이 non-blocking I/O 처럼 동작하는 것이다.

기본적으로 노드는 단일 event loop 를 가진 thread 로 동작하기 때문에 클라이언트 요청 증가에 대해서 thread driven 모델 보다는 좀 더 좋은 확장성(scalability)을 보인다. 그리고 별도의 thread 에서 blocking I/O 작업이 처리되는 동안 event loop thread 에선 다른 작업을 수행할 수 있으므로 동시에 발생하는 클라이언트 요청에 대해 좋은 반응성을 보여줄 수 있다.

노드는 한개의 event loop thread 로 서버로 도착하는 모든 요청을 처리하기 때문에, 특정 요청을 처리하는 javascript 콜백에서 연산등의 cpu intensive 한 작업을 오랜동안 한다면 그 시간만큼 다른 요청들은 대기하게 된다. **노드를 사용하는 좋은예는 CRUD 작업을 주로 하는 RESTful 서비스나 notification push 같은 실시간 웹서비스이다.**

History

노드는 라이언 달(Ryan Dahl)에 의해 2009 년 리눅스 기반으로 출시되었다.
노드는 구글의 V8 자바스크립트 엔진과 Joyent 사의 libuv 라이브러리를 기반으로 만들어졌다.
노드를 통해 자바스크립트 개발자도 파일시스템에 접근하고 네트워크 소켓을 열고 자식 프로세스를 만들 수 있게 되었다.

노드의 패키지 매니저인 npm 은 2011 년에 처음 소개되었다.
2011 년 6 월 마이크로소프트는 Joyent 사와 파트너십을 맺고 노드의 윈도우버전을 같은 해 7 월에 출시했다. 노드는 짧은 역사에도 불구하고 최근에 가장 폭발적으로 성장하고 있는 플랫폼 중에 하나이다.

노드는 버전 0.12.x 에서 갑자기 4.0.0 버전대로 올려서 2015 년 9 월 8 일 발표하였다. 이는 io.js 의 3.0.0 버전에서 대대적인 상향을 반영하였기 때문에 업그레이드 의미를 좀 더 부각시키기 위한 조치라고 한다.

https://nodejs.org/en/download/releases/

Pros/Cons

노드의 가장 뛰어난 장점은 많은 기능을 적은 코드로 구현하는 능력이다. 예를 몇줄의 코드만으로 간단한 웹서버를 구현한다. 빛이 있으면 어둠이 있는법이다. 새로운 도구를 얻은 목수의 입장에서 노드라는 도구를 어디에 어떻게 써야하는지 고려하기 위해서 장단점을 파악해 보자.

대부분의 기업형 애플리케이션은 주로 서버환경에서 동작한다.
서버는 Web 을 위한 HTTP 서버, 또는 소켓 통신을 위한 네트워크 서버 등이 있다.
클라이언트의 요청이 많은 경우 서버는 병목 현상이 발생하게 되는데 병목구간을 분석해 보면 대부분이 프로그램의 로직보다는 입출력(IO) 부분에서 발생한다. 웹서버는 수천에 이르는 동시 요청을 서비스하며 높은 가동 시간과 낮은 대기 시간이 요구된다. 노드는 이렇게 CPU 작업은 적고 IO 작업이 많은 업무를 처리하는데 높은 처리성능을 보여주는 기술이다.

노드는 싱글스레드 기반의 비동기처리 방식이므로 CPU 를 오랜동안 사용하는 작업이 많은 경우에는 적합하지 않다. 노드는 아직 안정성을 보장하기 힘들다는 의견도 있다. 또한 관계형 데이터베이스를 다루는데에는 아직 적합하지 않다는 의견이 지배적이다. 중첩되어 사용되는 수 많은 콜백함수는 유지보수성을 떨어뜨릴 수 있으며 이에 따라 에러 핸들링에 많은 주의가 필요하다.

노드의 성능은 기본적인 구조인 단일 스레드 기반의 비동기 IO 처리에서 온다고 볼 수 있다.
하나의 스레드가 request 를 받으면 처리를 하고 File IO 나 Network 작업 등이 있을 경우에는 IO 요청을 보내 놓고 작업을 처리하다가, IO 요청이 끝나면 이벤트를 받아서 처리하는 이벤트드리븐 방식을 사용한다. 이로 인해서 CPU 가 IO 응답을 기다리는 시간이 필요 없고, 대부분이 연산 작업에 사용되기 때문에 높은 효용성을 가지게 된다.

서버 프로그램에서 스레드간의 동기화 처리는 개발 로직을 복잡하게 만드는 주요소인데 이러한 문제 자체를 제거 함으로써 서버 프로그래밍 자체를 매우 단순하게 만들어 버린다. 노드의 리부팅시간이 몇초도 걸리지 않기 때문에 빠른 배포나 업그레이드가 빈번한 작업에서 운용하기 편리하다.

싱글스레드 모델이기 때문에 멀티 코어 머신에서는 CPU 사용을 최적화할 수 없다는 문제가 발생한다.
하나의 스레드는 하나의 물리적 코어밖에 사용하지 못하기 때문제 코어가 많은 시스템이라도 성능이 올라가지 않는다. 이를 보완하기 위해서 Cluster 모듈 등을 이용하여 하나의 서버에서 여러개의 노드 프로세스를 사용하는 모델을 가지고 가는 것이 좋다. 세션을 공유할 경우에는 세션 공유용 redis 와 같은 부가적인 인프라가 필요하다.

V8 엔진은 Garbage Collection 기반의 메모리 관리를 하기 때문에 GC 작업 시 CPU 사용률이 순간적으로 Spike 를 치면서 서버가 멈춘것처럼 생각되게 할 수 있다는 문제점을 가지고 있다. 싱글스레드 기반의 비동기 IO 를 지원해야 하기 때문에 노드 전용 모듈만을 사용해야 한다는 점은 운용의 폭을 축소시킨다.

개발관점에서는 개발이 빠르고 쉽다는 장점이 있지만 반대로 운영관점에서는 테스트, 장애 대응, 디버깅 등에 많은 주의를 필요로 한다.

노드 설치

노드는 다양한 플랫폼을 지원하며 윈도우, OS X, 리눅스 용 설치 패키지가 존재한다.
http://www.nodejs.org 페이지에서 OS 에 맞는 인스톨러를 다운로드 하고 설치한다.

설치가 되었으면, 설치된 디렉토리를 환경 PATH 에 추가한다.
윈도우 OS 에서는 **제어판₩모든 제어판 항목₩시스템 > 고급 시스템 설정 > 환경변수 > 시스템 변수** 부분에 있는 Path 항목에 노드 실행 파일인 node.exe 가 있는 위치를 추가한다.

노드가 잘 설치되었는지 확인하기 위해 콘솔 창을 띄우고 다음을 입력하여 테스트 하자.
노드 버전이 출력된다면 정상적으로 설치되었다는 뜻이다.

```
node --version
```

추가로 간단하게 "Hello World"라고 출력하는 코드를 작성해 보자. 에디터를 실행해서 helloworld.js 라는 파일을 작성한다.

/lesson/helloworld.js

```
console.log("Hello World");
```

콘솔창에서 다음을 입력하여 실행해 보자.

```
node helloworld.js
```

콘솔창에 "Hello World" 라는 문자열이 찍힌다면 노드가 정상적으로 작동하는 것이다.

콘솔에서 node 만 입력하고 엔터키를 치면 입력 문자가 '>'로 바뀌며 REPL(Read-Eval-Print-Loop)이 실행된다. REPL 로 일반적인 노드 함수와 모듈을 사용할 수 있다. 일반적으로 개발자들은 이를 **쉘커맨드**라고 부른다. 테스트로 다음 코드를 타이핑하고 엔터키를 쳐 보자.

```
> 1+2
3
> console.log('hello node')
hello node
> .help
```

break	Sometimes you get stuck, this gets you out
clear	Alias for .break
exit	Exit the repl
help	Show repl options
load	Load JS from a file into the REPL session
save	Save all evaluated commands in this REPL session to a file

Chapter 2. Node 맛보기

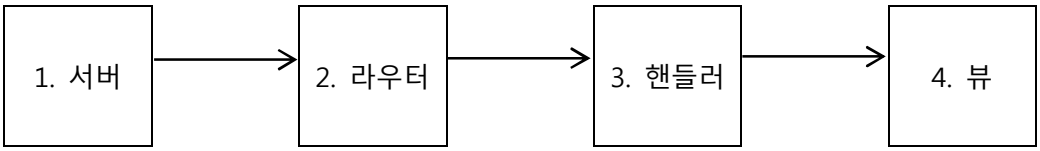
HTTP 서버

웹서비스를 제공하는 HTTP 서버도 노드를 사용하면 간단하게 만들 수 있다.

서버가 수행 해야하는 기능

1. 사용자가 접근 시 사용하는 url pattern(접근경로)을 처리해야 한다.
2. 처리로직을 갖고 있는 핸들러(함수)를 url pattern 과 연동하는 router(분기로직)가 필요하다.
3. 요청을 처리하는 핸들러(함수)를 만들어서 처리로직을 두고 그 결과를 요청자에게 전송해야 한다.
4. 필요하다고 판단되면 적합한 뷰를 선택해서 보여줄 수 있어야 한다.

- 라우터는 어떠한 방식(RESTful)의 데이터들도 처리할 수 있어야 한다.



이제부터 본격적으로 시작해 보자. 선호하는 텍스트 편집기를 열어 첫 번째 노드 웹서버를 작성해 보자. 복잡하고 무거운 IDE 는 사실 크게 필요하지 않다. 서버 모듈을 server.js 라고 하고, 다음과 같이 내용을 작성한다. 다음 코드는 노드에 기본적으로 포함된 내장모듈인 http 모듈을 읽어 들인다.

/http/server1.js

```
var http = require('http');

//http 모듈에서 제공하는 createServer 함수를 호출한다.
//createServer 함수 파라미터에 콜백함수를 넣어준다.
var server = http.createServer(function (request, response) {
        console.log("Request received.");

        //요청이 올 때마다 HTTP status 200 이라는 상태코드 값과 content-type 을
        //응답 헤더에 설정한다.
        response.writeHead(200, {
```

```
                    "Content-Type" : "text/plain"
            });
            //내용 텍스트를 HTTP 응답 바디에 추가한다.
            response.write("Hello, this is Server.");
            //응답을 마무리하고 전송한다.
            response.end();
    });

//listen 함수에 HTTP 서버에서 요청을 처리할 포트 번호를 설정하고 서버를 기동한다.
server.listen(8000, '127.0.0.1');

console.log('Server running at http://127.0.0.1:8000/');
```

콘솔에서 다음과 같이 입력하여 실행한다.

```
node server.js
```

테스트

다음으로 브라우저를 열고 주소창에 다음 주소를 입력하여 서버에 접근해 보자.

```
http://localhost:8000
```

"Hello, this is Server." 라는 문자열이 보이다면 성공이다.
우리는 노드를 사용하여 간단히 몇 줄의 코드만으로 웹서버를 만들 수 있었다.

클라이언트가 언제 서버로 접속할 지 모르므로 HTTP 요청은 비동기적으로 발생한다고 볼 수 있다.
클라이언트의 요청이 올 때 createServer 함수에 파라미터로 넘긴 콜백함수가 사용된다. 바로 동작하지 않고 이벤트가 발생할 때 사용되기 위해 파라미터로 전달되는 함수를 **콜백함수**라고 한다.

http.createServer 함수로 만든 server 객체는 server.listen(8000) 함수에 설정한 포트를 감시하면서 클라이언트의 접속을 기다린다.

브라우저가 HTTP 프로토콜 규약에 따라 서버로 전송하는 정보는 request 객체에 담겨 처리된다.
서버에서 브라우저로 보내고 싶은 정보는 response 객체에 추가되어 처리된다.
서버에서는 브라우저에게 서버가 정상적으로 응답했다는 것을 알려주기 위해서 약속된 "200" 이라는 상태코드값을 설정하며, 전송하고 싶은 정보를 바디부분에 추가하여 브라우저로 전송한다.

서버는 일반 애플리케이션과 다르게 클라이언트의 요청을 대기했다가 처리해야 하는 특성때문에 종료하지 않고 계속해서 메모리에 존재한다.

/http/server2.js

```
var http = require('http');
var fs = require('fs');

http.createServer(function(req, res) {

	if (req.url === '/favicon.ico') {
		return res.end();
	}

	console.log('Incoming request to ' + req.url);

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

	// setTimeout 함수는 비동기 호출이다.
	// 제공된 함수를 큐에 쌓아 두고 바로 다음 명령인 while 루프를 수행한다.
	setTimeout(function() {
		// __filename: 현재 수행 중인 파일의 경로를 가리킨다.
		fs.readFile(__filename, {
			encoding : 'utf8'
		}, function(error, contents) {
			if (error) {
				console.error(error);
				return res.end();
			}
			console.log('sending response for ' + req.url);
			// 현재 수행 중인 파일의 내용을 반환한다.
			res.end(contents);
		});
	}, 5000);

	var i = 2;
	while (i--) {
```

```
            console.log('Loop value: ' + i + '₩r');
        }
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');
```

테스트

다음 URL 2 개를 사용하여 거의 동시에 접속하여 테스트 한다.

http://127.0.0.1:1337/one

http://127.0.0.1:1337/two

콘솔창에 표시되는 결과

```
Server running at http://127.0.0.1:1337/
Incoming request to /one
Loop value: 1
Loop value: 0
Incoming request to /two
Loop value: 1
Loop value: 0
sending response for /one
sending response for /two
```

URL Handling

다음으로 사용자의 요청에 따라 대응하는 코드를 간단히 살펴보자.

/http/server3.js

```
var http = require('http');

http.createServer(function(req, res){
        if (req.url === '/' && req.method === 'GET') {
                res.writeHead(200, {'Content-Type':'text/html'});
                res.end('Hello <strong>home page</strong>');
        } else if(req.url === '/account' && req.method === 'GET') {
                res.writeHead(200, {'Content-Type':'text/html'});
                res.end('Hello <strong>account page</strong>');
        } else {
```

```
                res.writeHead(404, {'Content-Type':'text/html'});
                res.end();
        }
}).listen(1337);

console.log('Server is running at http://127.0.0.1:1337/');
```

Request Header

브라우저는 항상 서버에 요청할 때 사용자의 정보를 헤더에 저장하여 서버에 전송한다.

/http/server4.js

```
var http = require('http');

http.createServer(function(req, res){
        res.end('user-agent: '+req.headers['user-agent']);
        console.log(req.headers);
}).listen(1337);

console.log('Server is running at http://127.0.0.1:1337/');
```

Chapter 3. Node Basic

Node API

다음 링크로 접속하면 Node.js v4.2.3 버전의 Documentation 문서에서 함수들의 사용법을 자세히 살펴볼 수 있다.

https://nodejs.org/dist/latest-v4.x/docs/api/

추가로 http 모듈의 사용법을 API 문서에서 살펴보자.

https://nodejs.org/dist/latest-v4.x/docs/api/http.html

또는 객체 정보를 콘솔에 출력하여 확인할 수도 있다. STATUS_CODES 값을 주의깊게 살펴보자. 우리는 앞에서 서버가 클라이언트인 브라우저에게 정상적인 서버응답이라는 의미를 전달하기 위해서 "200"이라는 상태코드 값을 사용했다.

console.log(require('http')); 코드의 실행 결과

```
{
        parsers : {
                name : 'parsers',
                constructor : [Function],
                max : 1000,
                list : []
        },
        STATUS_CODES : {
                '100' : 'Continue',
                '101' : 'Switching Protocols',
                '102' : 'Processing',
                '200' : 'OK',
                '201' : 'Created',
                '202' : 'Accepted',
                '203' : 'Non-Authoritative Information',
                '204' : 'No Content',
                '205' : 'Reset Content',
                '206' : 'Partial Content',
                '207' : 'Multi-Status',
                '300' : 'Multiple Choices',
```

```
'301' : 'Moved Permanently',
'302' : 'Moved Temporarily',
'303' : 'See Other',
'304' : 'Not Modified',
'305' : 'Use Proxy',
'307' : 'Temporary Redirect',
'400' : 'Bad Request',
'401' : 'Unauthorized',
'402' : 'Payment Required',
'403' : 'Forbidden',
'404' : 'Not Found',
'405' : 'Method Not Allowed',
'406' : 'Not Acceptable',
'407' : 'Proxy Authentication Required',
'408' : 'Request Time-out',
'409' : 'Conflict',
'410' : 'Gone',
'411' : 'Length Required',
'412' : 'Precondition Failed',
'413' : 'Request Entity Too Large',
'414' : 'Request-URI Too Large',
'415' : 'Unsupported Media Type',
'416' : 'Requested Range Not Satisfiable',
'417' : 'Expectation Failed',
'418' : 'I₩'m a teapot',
'422' : 'Unprocessable Entity',
'423' : 'Locked',
'424' : 'Failed Dependency',
'425' : 'Unordered Collection',
'426' : 'Upgrade Required',
'428' : 'Precondition Required',
'429' : 'Too Many Requests',
'431' : 'Request Header Fields Too Large',
'500' : 'Internal Server Error',
'501' : 'Not Implemented',
'502' : 'Bad Gateway',
'503' : 'Service Unavailable',
'504' : 'Gateway Time-out',
'505' : 'HTTP Version Not Supported',
```

```
        '506' : 'Variant Also Negotiates',
        '507' : 'Insufficient Storage',
        '509' : 'Bandwidth Limit Exceeded',
        '510' : 'Not Extended',
        '511' : 'Network Authentication Required'
    },
    IncomingMessage : {
        [Function : IncomingMessage]
        super_ : {
            [Function : Readable]
            ReadableState : [Function : ReadableState],
            super_ : [Object],
            _fromList : [Function : fromList]
        }
    },
    OutgoingMessage : {
        [Function : OutgoingMessage]
        super_ : {
            [Function : Stream]
            super_ : [Object],
            Readable : [Object],
            Writable : [Object],
            Duplex : [Object],
            Transform : [Object],
            PassThrough : [Object],
            Stream : [Circular]
        }
    },
    ServerResponse : {
        [Function : ServerResponse]super_ : {
            [Function : OutgoingMessage]super_ : [Object]
        }
    },
    Agent : {
        [Function : Agent]
        super_ : {
            [Function : EventEmitter]listenerCount : [Function]
        },
        defaultMaxSockets : 5
```

```
        },
        globalAgent : {
                domain : null,
                _events : {
                        free : [Function]
                },
                _maxListeners : 10,
                options : {},
                requests : {},
                sockets : {},
                maxSockets : 5,
                createConnection : [Function]
        },
        ClientRequest : {
                [Function : ClientRequest]super_ : {
                        [Function : OutgoingMessage]super_ : [Object]
                }
        },
        request : [Function],
        get : [Function],
        Server : {
                [Function : Server]super_ : {
                        [Function : Server]super_ : [Object]
                }
        },
        createServer : [Function],
        _connectionListener : [Function : connectionListener],
        Client : [Function : deprecated],
        createClient : [Function : deprecated]
}
```

console.log(require('http').createServer.toString()); 코드의 실행 결과

```
function (requestListener) {
        return new Server(requestListener);
}
```

위 로직을 보면 createServer(콜백함수) 함수는 결국 new Server() 생성자 함수를 사용하여 객체를 만든 것이다. 따라서 새로 만들어지는 객체는 Sever 함수에 정의된 속성들을 물려받아 사용할 수 있다.

favicon.ico

한번의 브라우저 요청에 "Request received." 메시지가 두번 찍히는 것은 대부분의 브라우저가 http://localhost:8000/ 을 요청할 때 http://localhost:8888/favicon.ico 를 로드하려고 하기 때문이다.

최신 브라우저에서는 URL 의 왼쪽에 아이콘 하나가 표시된다.
이것이 바로 'favicon.ico'이며 일반적으로 도메인주소/favicon.ico 에서 가져온다.
서로 다른 사이트를 탐색할 때 브라우저에서는 이 파일을 자동으로 요청한다.
유효한 favicon.ico 파일이 브라우저에 전달되면 이 아이콘이 표시된다.
파일 전달이 실패하면 특수 아이콘이 표시되지 않는다.

다음 코드로 제외 시킬 수 있다.

```
if(request.url === '/favicon.ico'){
        return response.end();
}
```

Module System

노드는 프로그램을 여러 개의 파일로 나눠서 구현하는 간단한 모듈시템을 사용한다.
코드를 모듈로 만든다는 것은 별도로 .js 파일을 만들고 그 파일 안에서 정의된 기능중에서 외부에 제공하고 싶은 기능의 일부를 exports 내장객체의 속성으로 추가하여 제공하는 것을 의미한다.

자세히 살펴보기 위해서 함수를 제공할 파일을 작성하자.

/module/1/hello.js

```
exports.world = function() {
        console.log('Hello World');
}

var asia = function() {
        console.log('Hello Asia');
}

exports.korea = asia;
```

exports 내장객체에 속성 world 를 추가하고 그 속성이 함수를 가리키도록 한다.
hello.js 파일안에서는 함수를 가리키는 변수는 asia 이지만, 외부로 함수를 제공할 때 korea 로 이름을 변경할 수 있다.

다음으로 main.js 파일을 만들어서 hello.js 파일로 분리된 모듈을 사용해 보자.

/module/1/main.js

```
var hello = require('./hello');

hello.world();
hello.korea();
```

테스트

이제 main.js 를 실행하면 다음 결과를 볼 수 있다.

```
Hello World
Hello Asia
```

require()

함수로 다른 자바스크립트 파일을 임포트한다. 또한 한 번만 동작하는 코드를 수행하기 위해서도 사용할 수 있다. 모듈을 불러들일 때 파일 경로를 명시할 수 있다.

'.js' 확장자

모듈로 분리된 자바스크립트 파일명을 명기할 때 확장자는 자동으로 추가되기 때문에 확장자를 생략한다.

'./' 문자열

대상 파일이 코드를 담고 있는 main.js 와 같은 디렉토리에 있다는 것을 의미한다.
만약 대상 파일이 한 단계 위 폴더에 위치하는 경우에는 '../' 연산자를 사용한다.
world 와 korea 는 함수를 가리키고 있다. 함수를 실행하기 위해서 () 연산자를 사용한다.

노드 모듈 저장위치

-g 플래그를 사용하면 전역으로 설치되어 쉘커맨드에서 쓸 수 있게 된다.
-g 플래그를 사용하지 않으면 명령을 내린 디렉토리 위치를 기준으로 node_modules 폴더를 생성하고 로컬로 설치되어 해당 앱만 쓸 수 있게 된다.

```
npm install -g 모듈명
```

노드 모듈 설치 시 -g 옵션을 붙이면 기본적으로 다음 명령으로 알 수 있는 저장 위치에 저장된다.

```
npm root -g
```

다음 명령으로 설치된 전역모듈을 확인할 수 있다.

```
npm list -g
```

필요 없는 로컬 모듈은 다음 명령으로 삭제할 수 있다.

```
npm uninstall 모듈명
```

모듈 탐색

require 는 포함하는 파일의 상대적 경로에 대한 정보를 가지지 않는다. 그러므로 같은 디렉토리에 위치하지 않는 경우 어떻게 처리해야 하는지 알아둘 필요가 있다.

다음 같은 코드가 있는 경우라면 어떤 순서로 작동하는지 살펴 보자.

```
var http = require('http');
```

노드는 먼저 노드 내장모듈에 http 라는 모듈이 있는 지 찾는다. http 는 내장모듈이므로 바로 찾을 수 있다. 왼쪽 변수 http 로 http.js 파일 내에서 exports 객체에 추가된 속성을 접근하여 사용할 수 있게 된다.

다음으로 color 와 같이 코어(내장)모듈이 아닌 경우는 어떻게 처리되는지 살펴보자.

```
var color = require('color');
```

1. 코어 모듈인지 조사한다. color 는 내장모듈이 아니다.
2. 현재 디렉토리의 node_modules 폴더를 검사한다.
    > 부모 디렉토리로 이동해서 node_modules 폴더를 검사한다.
        > 루트 디렉토리('/')에 도달할때까지 계속 부모 디렉토리로 이동한면서
            node_modules 폴더에서 파일을 찾는다.
3. 추가적으로 require.paths 라는 변수를 사용한다면 변수가 가리키는 배열에 등록된 경로에서도 파일을 찾는다.
4. 그리고 디렉토리 대표 파일인 index.js 안에서도 찾는다.

예를들어 require('./foo')라고 호출하면 foo.js 라는 파일을 찾는다. 더불어 foo 라는 이름의 폴더 하부에 위치하는 foo/index.js 라는 파일을 찾는다.

npm link

npm link 명령으로 해당 모듈에 전역 심벌릭 링크를 설정함으로써 마치 외부 모듈이 현재 프로젝트의 지역 node_modules 폴더에 있는 것처럼 사용할 수 있다. 우선 colors 모듈을 설치해서 잘 작동하는지 살펴 보자.

```
npm install colors
```

다음 작업 대상은 정수의 덧셈, 곱셈, 팩토리얼을 수행하는 함수다.

/colors/calculator.js

```
function add(number1, number2){
	return parseInt(number1, 10) + parseInt(number2, 10);
}

function multiply(number1, number2){
	return parseInt(number1, 10) * parseInt(number2, 10);
}

function factorial(number){
	if(number === 0){
		return 1;
	}
	return number * factorial(number - 1);
}

exports.add = add;
exports.multiply = multiply;
exports.factorial = factorial;
exports.ymd= new Date();
```

다음으로 이를 사용하는 파일을 작성한다.

/colors/test.js

```
var colors = require('colors');
var m = require('./calculator');
```

```
console.log(m.add(3, 5).toString().red);
console.log(m.multiply(4, 5).toString().green);
console.log(m.factorial(4).toString().black.bgWhite);
console.log(m.ymd.toString().cyan);
```

calculator.js 를 대상으로 require 하면 자바스크립트를 실행하고 객체를 반환한다.
실행하여 잘 작동하는지 테스트한다.

모듈 캐시

모듈이 처음 호출되면 그 결과를 캐시한다. 따라서, 여러 차례 호출해도 항상 같은 인스턴스를 반환한다. 캐시는 입력받은 파일 이름을 키로 사용하여 구분한다. 같은 파일이라도 경로가 다르게 호출한다면 개별적인 객체로 취급된다.

```
var calc1 = require('./calculator');
console.log(calc1.now);

setTimeout(function(){
        var calc2 = require('./calculator');
        console.log(calc2.now);
}, 5000);
```

link 설정 및 적용

npm link 명령으로 해당 모듈에 전역 심벌릭 링크를 설정한다. 마치 외부 모듈이 현재 프로젝트의 지역 node_modules 폴더에 있는 것처럼 사용할 수 있게 해준다.

1. 새 폴더 math_module 을 만들고 calculator.js 를 이동시킨다.
2. 외부에서 이 모듈을 사용할 수 있도록 다음 코드를 실행하여 package.json 파일을 생성한다.
모듈 이름은 calc 로 지정한다.

```
npm init
```

package.json

```
{
  "name": "calc",
  "version": "1.0.0",
  "description": "basic calculator",
  "main": "calculator.js",
  "scripts": {
    "test": "echo ₩"Error: no test specified₩" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

package.json 프로퍼티 정보

Property	Description
name	모듈의 이름이다. npm install <모듈이름> 명령에 주어지는 모듈이름과 같다. 필수 항목이고 유일하게 작성함을 권장한다. node, js 는 예약어로 사용할 수 없으며 URL 에 사용할 수 없는 문자는 피한다.
version	주버전.부버전.패치 형태로 표현한다.
scripts	Package.json 파일이 위치한 디렉토리에서 실행할 수 있는 추가적인 npm 명령을 제공한다. 예를 들어 npm run populate 명령을 실행하면 npm 은 node ./bin/populate_db 를 수행할 것이다. 설정 값으로 노드 명령이나 쉘 스크립트를 적는다.
dependencies	현재 모듈이나 앱이 작동하기 위해 필요한 모듈들을 설정한다. 키는 모듈 이름이며 설정 값은 모듈 버전이다. 또한 깃허브를 직접 가리키는 방식으로도 설정할 수 있다. 세밀한 버전설정은 매뉴얼을 참조하자. https://github.com/npm/node-semver

3. math_module 폴더에서 다음을 실행한다.

```
npm link
```

이로써 math_module 디렉토리는 전역 노드 모듈의 심벌릭 링크로 등록되었다.

4. 다음으로 test.js 가 있는 디렉토리는 이동하여 다음 코드를 실행한다.

```
npm link calc
```

변경 된 test.js

```
var colors = require('colors');
var m = require('calc');

console.log(m.add(3, 5).toString().red);
console.log(m.multiply(4, 5).toString().green);
console.log(m.factorial(4).toString().blue);
console.log(m.ymd.toString().gray);
```

실행하여 이전과 같이 잘 작동하는지 테스트한다.

대상 모듈에 link 를 설정하여 사용하는 모듈과 완전히 별도의 디렉토리로 분리할 수 있었다.

Chapter 4. Web Service I

Event Listener

on 함수를 사용하여 이벤트리스너를 등록할 수 있다.

예약된 이벤트의 이름과 콜백함수를 on 함수의 파라미터로 넘긴다.

다음 코드에서 사용하는 **data** 와 **end** 는 request 객체가 미리 설정한 **예약된 이벤트의 이름**이다.

브라우저로부터 서버로 전송되는 정보가 있다면 data 예약어와 더불어 주어진 콜백함수가 기동한다.

브라우저로부터 서버로 전송되는 정보의 전송이 완료되었다면 end 예약어와 더불어 주어진 콜백함수가 기동한다.

```
var data = '';
request.on('data', function(chunk) {
        data += chunk;
}).on('end', function() {
        console.log('data: %s', data);
});
```

on 함수는 객체의 참조(request 객체 자신)를 리턴하기 때문에 함수체인 코딩방식을 사용할 수 있다.

만약 이벤트를 한 번만 처리하고 싶을 때에는 on 대신 once 함수를 사용하면 된다.

removeListener 함수를 호출하여 이벤트 리스너를 제거할 수 있다.

removeListener 함수는 이벤트 이름이 아니라 함수를 참조하는 변수를 파라미터로 사용해야 한다.

```
var onData = function(chunk) {
        console.log(chunk);
        request.removeListener(onData);
}
request.on('data', onData);
```

위 코드처럼 사용한다면 on 함수는 once 함수를 사용한 것과 같이 동일하게 작동한다.

Http Module

앞에서 살펴본 모듈 개념을 적용해 보자. 모듈로 분리하면 SoC(Seperation of Concern) 개념을 적용하는 것이며 이렇게 구성하면 관리성이 증대된다.

server.js 를 노드모듈로 만들어서 전체 앱을 통제하는 역할의 index.js 파일에서 사용하는 방식으로 변경해 보자.

/http/2/server.js

```
var http = require('http');

function start() {
	var server = http.createServer(function(request, response) {
		console.log("Request received.");

		response.writeHead(200);
		response.write("Hello, this is Server.");
		response.end();
	});

	server.listen(8000);
	console.log('Listening on port 8000...');
}

// 모듈처리를 위해 사용하는 exports 객체에 start 속성을 추가하고 함수 start 를 가리키도록 한다.
exports.start = start;
```

새로 작성한 함수를 내장객체 exports 에 속성으로 자유롭게 추가하는 방식은 자바스크립트의 first class 특징을 잘 보여준다.

First class

In computer science, a programming language is said to support first-class functions (or function literal) if it treats functions as first-class objects.

Specifically, this means that the language supports

1. constructing new functions during the execution of a program,

2. storing them in data structures,
3. passing them as arguments to other functions,
4. and returning them as the values of other functions.

- A function is an instance of the Object type
- A function can have properties and has a link back to its constructor method
- You can store the function in a variable
- You can pass the function as a parameter to another function
- You can return the function from a function

다음으로 index.js 파일을 만들고 내용을 아래와 같이 작성한다.

/http/2/index.js

```
var server = require("./server");
server.start();
```

우리가 만든 모듈도 다른 내장 모듈과 똑 같은 방법으로 사용하면 된다.
위 코드는 server.js 파일 안에서 export 한 start 함수를 호출한다.

테스트

다음처럼 실행하여 기존과 동일하게 작동하는지 확인하자.

```
node index.js
```

index.js 에서 server.js 를 사용하고 있으므로 결과는 같다.
관리 목적상 일부 기능만 모듈로써 분리한 것이다.

Routing

HTTP 요청(url pattern)을 받아 처리 로직으로 연동하는 작업을 라우팅(routing) 이라고 한다.
요청 url pattern 과 GET/POST 방식의 파라미터를 router 로 전달하고 router 에서는 어떤 로직을 실행할지 결정한다. 요청을 받았을 때 실제 일은 request handler 가 처리하도록 한다.

브라우저가 요청한 url pattern 이 무엇인지 알 수 있도록 하는 로직을 onRequest 콜백함수에 추가한다.

/http/3/server.js

```
var http = require("http");
var url = require("url");

function start() {
	function onRequest(request, response) {
		var pathname = url.parse(request.url).pathname;
		console.log("Request for " + pathname + " received.");

		response.writeHead(200, {"Content-Type": "text/plain"});
		response.write("Hello World");
		response.end();
	}

	http.createServer(onRequest).listen(8000);
	console.log("Server has started.");
}

exports.start = start;
```

/http/3/index.js

```
var server = require("./server");
server.start();
```

테스트

콘솔에서 node index.js 명령으로 서버를 재 실행한다. 브라우저 주소창에 다음처럼 입력하여 테스트 한다.

http://localhost:8000/first

http://localhost:8000/second

결과

```
Request for /first received.
Request for /second received.
```

request 객체는 callback 함수인 onRequest 함수에 첫 번째 파라미터로 넘어 온다.
request 객체의 url 을 파싱하여 세부정보를 얻어내기 위해 url 과 querystring 이라는 모듈을 사용할 수 있다. url 모듈은 request 객체의 url 을 분리해서 얻을 수 있는 함수를 제공한다. querystring 모듈을 POST 요청의 body 를 파싱하는데 사용하면 편리하다.

다음과 같이 긴 요청을 실행하여 url 모듈을 사용하는 이유를 알아보자.

http://localhost:8000/first/second?a=1

URL Pattern 처리

예를 들어 다음 URL 로 유저가 접근했다고 가정하고 살펴보자.

서버에서는 request.url 코드로 아래 URL 문자열 전부를 얻을 수 있다.

http://localhost:8000/start?foo=bar&hello=world

이를 파싱하여 세부적으로 원하는 정보를 얻기 위한 작업은 다음 표를 참고하자.

코드	결과
url.parse(request.url).pathname	/start
url.parse(request.url).query	**foo=bar&hello=world**
querystring.parse(**query**)["foo"]	bar
querystring.parse(query)["hello"]	world

URL 의 물음표(?) 기호를 기준으로 왼쪽에 path 와 오른쪽에 query 부분을 분리해서 얻을 수 있다.

querystring 함수는 파라미터들을 파싱하여 객체로 관리한다.

점(.) 연산자나 [] 연산자를 사용하여 키에 해당하는 값을 얻을 수 있다.

Routing Module

pathname 에 따라 로직을 선택하는 기능을 추가하기 위해서 router.js 라는 새 파일을 만들고 내용을 아래와 같이 작성한다.

/http/4/router.js

```
function route(pathname) {
        console.log("About to route a request for " + pathname);
}

exports.route = route;
```

HTTP server 로직에 router 를 적용해보자.

dependency injection 을 통해 server 와 router 를 느슨하게 결합하도록 만든다.

Dependency Injection

요청객체 : A, 담당객체 : B, 처리객체 : C
의존관계 주입이라는 용어는 B 가 일을 시키는 객체 C 를 스스로 선택하는 것이 아니라 업무요청을 하는 A 가 실제로 업무를 처리하는 객체 C 를 골라서 B 에게 알려주는 방식을 말한다.

다음 예에서는 server.js 가 사용할 모듈을 스스로 결정 하는 것이 아니라 server.js 를 사용할 index.js 에서 대상 모듈을 결정해서 파라미터로 server.js 에게 알려주고 있다.

server.js 에서 router.js 의 route 함수를 파라미터로 주입받아 사용하도록 다음처럼 변경한다.

/http/4/server.js

```
var http = require("http");
var url = require("url");

function start(route) {
        function onRequest(request, response) {
                var pathname = url.parse(request.url).pathname;
                console.log("Request for " + pathname + " received.");

                route(pathname);
```

```
                response.writeHead(200, {"Content-Type": "text/plain"});
                response.write("Hello World");
                response.end();
        }

        http.createServer(onRequest).listen(8000);
        console.log("Server has started.");
}

exports.start = start;
```

다음으로 index.js 를 변경한다. start 함수에 파라미터로 route 함수를 주입(Injection)한다.

/http/4/index.js

```
var server = require("./server");
var router = require("./router");

server.start(router.route);
```

테스트

index.js 를 다시 기동한다. 브라우저를 사용하여 http://localhost:8000/first 주소로 접근한다.
pathname 을 route 함수에 파라미터로 넘기고 있으므로 다음과 같은 메시지가 뜨면 성공이다.

결과

```
Request for /first received.
About to route a request for /first
```

좀 더 개선해 보자. 요청을 받아 실질적으로 처리하는 로직을 갖고 있는 함수를 request handler 라고 부른다. requestHandlers.js 를 새로 작성하여 모듈로써 이용할 수 있도록 설정하고 각각의 요청을 처리할 함수들을 추가한다.

/http/5/requestHandlers.js

```
function first() {
        console.log("Request handler 'first' was called.");
}

function second() {
        console.log("Request handler 'second' was called.");
}

exports.first = first;
exports.second = second;
```

앞에서 만든 핸들러를 서버에 알려주기 위해서 index.js 를 다음과 같이 변경한다.

/http/5/index.js

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {};
// 루트 요청도 first 로 연결되도록 한다.
handle["/"] = requestHandlers.first;
handle["/first"] = requestHandlers.first;
handle["/second"] = requestHandlers.second;

server.start(router.route, handle);
```

URL 패턴과 이에 대응하는 함수를 관리하는 객체 handle 을 만들었다.

그리고 handle 객체를 server.start 함수에 파라미터로 추가하여 전달한다.

추가된 파라미터를 처리하기 위해 server.js 를 다음과 같이 변경한다.

/http/5/server.js

```
var http = require("http");
var url = require("url");
```

```
function start(route, handle) {
	function onRequest(request, response) {
		var pathname = url.parse(request.url).pathname;
		console.log("Request for " + pathname + " received.");

		route(handle, pathname);

		response.writeHead(200, {"Content-Type": "text/plain"});
		response.write("Hello World");
		response.end();
	}

	http.createServer(onRequest).listen(8000);
	console.log("Server has started.");
}

exports.start = start;
```

추가로 router.js 파일을 다음과 같이 수정한다.
브라우저를 통해 요청받은 url pattern 에 대응하는 처리 로직이 있는지 확인하기 위해서 주어진 pathname 에 해당하는 request handler 가 있는지 체크하고, 존재한다면 그 함수가 기동하도록 조치한다.

/http/5/router.js

```
function route(handle, pathname) {
	console.log("About to route a request for " + pathname);

	if (typeof handle[pathname] === 'function') {
		handle[pathname]();
	} else {
		console.log("No request handler found for " + pathname);
	}
}

exports.route = route;
```

테스트

프로그램을 다시 실행하고 브라우저로 http://localhost:8000/first 를 요청하여 테스트 한다.

결과

```
Request for /first received.
About to route a request for /first
Request handler 'first' was called.
```

위 결과를 살펴보면 처리순서는 다음과 같다.

1. 서버가 사용자의 요청을 받았다.

Request for /first received.

2. 라우터가 pathname 에 따라 분기작업을 처리했다.

About to route a request for /first

3. 요청에 대응하는 리퀘스트핸들러가 기동했다.

Request handler 'first' was called.

Response 처리

라우팅처리는 했으므로 이제부터 request handler 가 단순히 로그를 찍는 것이 아니라 제대로 일을 수행하게 만들어 보자. 브라우저에 찍히는 "Hello World" 는 여전히 server.js 파일의 onRequest 함수에서 바로 응답한 내용이다. 이제 onRequest 함수가 하는 것처럼 request handler 가 브라우저에게 이야기할 수 있도록 변경한다.

우선 살펴 볼 방식은 **직관적이지만 노드에서는 나쁜 방식**이라 할 수 있는 방식이다.
어떻게 코딩을 해야하는지 제대로 이해하기 위해서 먼저 이 방식을 적용해 보도록 한다.
request handler 는 사용자에게 보여주려는 내용을 리턴하고, 라우터가 이를 인계하며 서버가 받아 서버의 onRequest 함수에서 user 에게 전송하는 방식이다.

requestHandlers.js 를 다음과 같이 수정한다.

/http/6/requestHandlers.js

```
function first() {
        console.log("Request handler 'first' was called.");
        return "Hello First";
}

function second() {
        console.log("Request handler 'second' was called.");
        return "Hello Second";
}

exports.first = first;
exports.second = second;
```

핸들러 함수가 리턴한 내용을 라우터가 받은 후 그대로 서버 처리로직으로 리턴하도록 router.js 를 다음과 같이 변경한다.

/http/6/router.js

```
function route(handle, pathname) {
        console.log("About to route a request for " + pathname);

        if (typeof handle[pathname] === 'function') {
```

```
            return handle[pathname]();
        } else {
            console.log("No request handler found for " + pathname);
            return "404 Not found";
        }
}

exports.route = route;
```

핸들러 > 라우터 > 서버 처리 로직으로 전해진 내용을 서버 처리 로직에서 사용하도록 server.js 를 다음과 같이 바꾼다.

/http/6/server.js

```
var http = require("http");
var url = require("url");

function start(route, handle) {
    function onRequest(request, response) {
        var pathname = url.parse(request.url).pathname;
        console.log("Request for " + pathname + " received.");

        var content = route(handle, pathname);

        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write(content);
        response.end();
    }

    http.createServer(onRequest).listen(8000);
    console.log("Server has started.");
}

exports.start = start;
```

테스트

수정한 프로그램을 다시 기동한다. 다음 주소로 접근하여 테스트 한다.

http://localhost:8000/first
http://localhost:8000/second
http://localhost:8000/
http://localhost:8000/foo

결과

```
Request for /first received.
About to route a request for /first
Request handler 'first' was called.

Request for /second received.
About to route a request for /second
Request handler 'second' was called.

Request for / received.
About to route a request for /
Request handler 'first' was called.

Request for /foo received.
About to route a request for /foo
No request handler found for /foo
```

직관적으로 로직을 구성했고 잘 작동하고 있는 것처럼 보인다.
하지만 이 방식의 코딩은 노드에서 심각한 문제를 일으킬 수 있다. request handler 함수 중 하나가 blocking 방식으로 동작을 하게되면 문제가 발생한다.

결론부터 말하자면 노드에서는 request handler 가 결과를 리턴해서 server 에서 브라우저에게 전송하는 방식을 사용하지 말자. 대신 server 에서 request handler 에게 response 객체를 전달하고 request handler 가 직접 결과를 브라우저에게 전송하도록 하자.

왜 그렇게 사용해야 하는지 이해하기 위해서 먼저 노드에서 가장 주의해야 할 개념인 blocking 과 non-blocking 방식에 대해 살펴 보자.

Chapter 5. 멈춤이 없어야 하는 처리 흐름

Blocking Operation

시간이 오래 소요되는 작업을 blocking operation 이라고 한다.

request handler 함수들 중에 하나에서 blocking operation 이 발생하면 무슨 일이 생기는지 살펴보자. 이를 위해, /first 요청을 처리하는 request handler 함수에서 "Hello First" 문자열을 리턴하기 전에 10 초 동안 기다리도록 수정한다. 이는 IO 연산에 10 초가 소요되는 경우를 가정한다고 보면 된다.

requestHandlers.js 를 아래와 같이 수정한다. first() 함수가 호출되면 노드는 10 초를 기다리고 "Hello First"를 리턴한다.

/http/7/requestHandlers.js

```
function first() {
	console.log("Request handler 'first' was called.");

	function sleep(milliSeconds) {
		var startTime = new Date().getTime();
		console.log('start time: '+startTime);
		while (new Date().getTime() < startTime + milliSeconds);
		console.log('end time: '+new Date().getTime());
	}
	sleep(10000);

	return "Hello First";
}

function second() {
	console.log("Request handler 'second' was called.");
	return "Hello Second";
}

exports.first = first;
exports.second = second;
```

오랜 시간이 걸리는 로직이 핸들러에 존재 할 때 무슨 일이 일어나는지 보기 위해 다음과 같이 테스트 한다.

1. 첫 번째 브라우저의 주소창에 http://localhost:8888/first 라고 하는 주소를 입력만 하고 대기한다.

2. 두 번째 브라우저의 주소창에 http://localhost:8888/second 라고 하는 주소를 입력만하고 대기한다.

3. 빠르게 첫 번째 브라우저와 두 번째 브라우저의 주소창으로 이동하여 연속으로 엔터키를 누른다.
즉, 두 개의 요청을 거의 동시에 서버로 날려서 그 반응을 살펴보고자 하는 것이다.

```
Request for /first received.
About to route a request for /first
Request handler 'first' was called.
start time: 1449129861307
end time: 1449129871307
Request for /second received.
About to route a request for /second
Request handler 'second' was called.
```

우리가 예상한 대로 /first URL 은 로드하는데 10 초가 걸린다.
그런데, /second 의 요청을 처리하는 request handler 함수에는 10 초 동안 대기하는 로직이 없음에도 불구하고 결과를 서버로부터 브라우저가 받아서 표시하는 데 10 초가 걸린다.

이렇게 된 이유는 first 함수가 blocking operation 동작을 포함하기 때문이다.
노드는 다수의 동시작업을 처리할 수 있지만 스레드를 사용하는 방식으로 구동되지 않는다.
노드를 사용하여 개발하는 **개발자 입장에서 보면 노드는 언제나 단일 스레드**임을 명심해야 한다.

이로써 이제 우리는 blocking 동작을 피하고 non-blocking 동작을 사용해야 하는 이유를 알았다.
노드에서 시간이 오래 걸리는 작업은 언제나 callback 함수를 사용하여 처리해야 한다.

Single 스레드 이벤트루프를 사용하는 노드는 blocking 연산이 없이 계속해서 다음 코드 라인이 실행되도록 구성해야 한다.

Non-blocking Operation

non-blocking 방식으로 동작하는 대표적인 함수 exec 를 살펴보자.

exec 함수를 사용하기 위해서는 새로운 노드 모듈인 child_process 를 불러와야 한다.

exec 함수는 **shell 커맨드를 노드 안에서 실행**시킬 수 있게 한다.

다음 예제는 현재 디렉토리에 있는 모든 파일 리스트를 구한다. 파일 리스트 내용이 많은 경우 언제든지 blocking operation 이 될 수 있는 작업이다. non-blocking 방식으로 작동하는 exec 함수를 사용해서 브라우저가 /first URL 을 요청할 때 파일 리스트를 출력하도록 해 보자.

먼저 requestHandlers.js 파일을 수정한다.

/http/8/requestHandlers.js

```
var exec = require("child_process").exec;

function first() {
        console.log("Request handler 'first' was called.");
        var content = "empty";

        //dir 이라는 shell 커맨드를 Non-blocking 방식으로 처리한다.
        exec("dir", function (error, stdout, stderr) {
                content = stdout;
        });

        return content;
}

function second() {
        console.log("Request handler 'second' was called.");
        return "Hello Second";
}

exports.first = first;
exports.second = second;
```

코드를 분석하면 우선 결과를 담을 content 라는 변수를 추가한다.

그런 다음 shell 커맨드 명령인 "dir" 을 exec 함수로 실행해서 그 결과를 파라미터 stdout 으로 받는다.

그리고 stdout 을 변수 content 에 넣어 리턴한다고 읽힌다.

애플리케이션을 구동시키고 http://localhost:8888/first 와 http://localhost:8888/second 를 거의 동시에 접근해 보면 non-blocking 방식으로 작동함을 알 수 있다.

그런데 first 로 요청 한 결과가 "empty" 문자열이다. 디렉토리 정보가 출력되지 않는다. 무엇이 잘못 된 걸까?

exec 함수를 사용하여 non-blocking 방식으로 first 함수내 로직을 처리했지만 브라우저에 그 결과를 출력하지 못했기 때문에 이 또한 바라던 결과가 아니다.

문제의 원인은 exec 함수가 non-blocking 으로 동작하기 위해서 callback 함수를 사용해야 한다는 것이다. 노드의 코드는 멈춤없이 계속해서 순차적으로 동작한다. 즉, exec 함수를 호출하고 바로 다음 줄 코드인 return 구문을 실행한다.

이 시점에 content 는 "empty" 이다. 왜냐하면 exec 함수는 비동기적으로 동작하기 때문에 전달된 callback 함수는 아직 호출되지 않았던 것이다. callback 함수는 first 함수 로직이 모두 처리된 후, callback 함수가 호출되면서 결과를 파라미터로 받고 content 변수에 담는다. 이미 content 는 "empty"라는 문자열을 리턴한 다음이다. 이러면 쓸모가 없다.

Non-blocking 방식의 응답처리

프로그램에서 기다림은 나쁘다. 절차적인 코딩방식은 기다림을 수반하기 때문에 나쁘다.
기다리지 않기 위해서 스레드로 처리하는 방식은 서버의 부하를 가중시키기 때문에 나쁘다.
대신 콜백함수를 사용하자.

server 입장에서 request handler 가 결과를 리턴 할 때까지 기다리지 말자.
리턴 작업에 사용하는 객체 response 를 request handler 에 전달하여 직접 사용하도록 만들자.
"작업의 결과를 알려달라고 하지 말고 대신 처리방법을 알려주고 작업을 시키자."

새로운 접근 방법은 다음과 같다.
결과를 server 로 보내는 대신, server 의 response 객체를 router 를 통해 request handler 에게 전달한다.
그러면 handler 는 전달 받은 response 객체가 가진 함수들을 이용해서 스스로 응답을 처리할 수 있다.
즉, handler 에서 사용하는 콜백함수에게 response 객체를 주고 처리가 끝나면 이를 이용해서 직접 브라우저에게 응답하게 만든다.

이에 맞게 애플리케이션을 하나씩 고쳐 보자.

/http/9/server.js

```
var http = require("http");
var url = require("url");

function start(route, handle) {
	function onRequest(request, response) {
		var pathname = url.parse(request.url).pathname;
		console.log("Request for " + pathname + " received.");

		route(handle, pathname, response);
	}

	http.createServer(onRequest).listen(8000);
	console.log("Server has started.");
}

exports.start = start;
```

route 함수가 결과를 return 하는 것을 기다리는 대신 세 번째 파라미터로 response 객체를 전달한다.

/http/9/router.js

```
function route(handle, pathname, response) {
	console.log("About to route a request for " + pathname);

	if (typeof handle[pathname] === 'function') {
		handle[pathname](response);
	} else {
		console.log("No request handler found for " + pathname);
		response.writeHead(404, {"Content-Type": "text/plain"});
		response.write("404 Not found");
		response.end();
	}
}

exports.route = route;
```

request handler 에게 response 객체를 전달하여 request handler 가 직접 브라우저에게 응답할 수 있게 만든다. 요청 url pattern 에 맞는 request handler 가 없다면 router 에서 직접 response 객체를 사용하여 브라우저에게 응답하면 된다.

/http/9/requestHandlers.js

```
var exec = require("child_process").exec;

function first(response) {
	console.log("Request handler 'first' was called.");

	exec("dir", function (error, stdout, stderr) {
		response.writeHead(200, {"Content-Type": "text/plain"});
		response.write(stdout);
		response.end();
	});
}

function second(response) {
	console.log("Request handler 'second' was called.");
```

```
        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("Hello Second");
        response.end();
}

exports.first = first;
exports.second = second;
```

first 함수 내부에서 사용하는 exec 함수에게 파라미터로 콜백함수를 전달한다.
쉘커맨드 dir 의 작업이 끝나면 그 결과가 콜백함수 stdout 파라미터로 전달되어 오면서 콜백함수가 기동한다.
콜백함수안에서는 response 객체를 직접 사용하여 응답한다. 그리고 second handler 는 여전히 "Hello Second"라는 문자열을 응답하지만 이번에는 직접 response 객체를 사용하고 있다. 서버를 재시작해서 결과를 확인하자.

Shell Command Encoding

노드는 언제나 child process 가 UTF-8 로 출력할거라 기대한다. 그러나, 윈도우의 콘솔모드는 CP949 를 인코딩방식으로 사용한다. 그러한 이유 때문에 앞 예제에서 브라우저에 출력된 한글이 깨지는 현상이 발생할 수 있다.

노드가 다국어 처리에는 아직 부족한 부분이 있다고 볼 수 있다. 약간의 수단을 부려 글자가 깨지지 않고 출력되도록 해 볼 수 있다. 코드에서 사용할 'testfile.txt' 파일은 미리 만들지 않는다.

/http/10/requestHandlers.js

```
var exec = require("child_process").exec;
var fs = require('fs');

function first(response) {
	console.log("Request handler 'first' was called.");

	exec("chcp 65001 | dir", function (error, stdout, stderr) {
		// 로그 용도로 파일에 저장
		fs.createWriteStream('testfile.txt', {
			flags: 'w',
			encoding: 'binary'
		}).write(stdout);

		exec('chcp 949', function(){
			console.log('₩n' + stdout);
		});

		response.writeHead(200, {"Content-Type": "text/plain; charset=UTF-8"});
		response.write(stdout);
		response.end();
	});
}

function second(response) {
	console.log("Request handler 'second' was called.");

	response.writeHead(200, {"Content-Type": "text/plain"});
	response.write("Hello Second");
```

```
        response.end();
}

exports.first = first;
exports.second = second;
```

파일들의 역할

정리하는 목적에서 지금까지 만든 파일의 역할을 검토해 보자.

파일명	역할
index.js	프로그램의 시작, 객체 생성 및 관계설정(dependency injection), 라우팅 맵핑
server.js	HTTP 서버(요청접수)
router.js	라우팅(URL Pattern 에 따른 Reqeuset Handler 연동)
requestHandler.js	요청처리, 결과 전송

Chapter 6. Web Service II

POST 요청 처리

이번에는 브라우저가 서버에 정보를 전달하는 POST 방식의 요청을 서버에서 처리하는 방법을 살펴보자. textarea 를 제공해서 사용자가 내용을 입력하고 submit 하면 POST 방식으로 서버로 전송하도록 한다. 서버에서는 받은 내용을 출력하여 연동을 확인한다.

/http/11/index.js

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {};
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;

server.start(router.route, handle);
```

사용자가 큰 사이즈의 텍스트를 입력할 수도 있다. 이 경우 전체 데이터 블록을 하나로 처리하는 것은 blocking operation 이 발생 할 소지가 있다. 전체 프로세스를 non-blocking 으로 만들려면, POST 데이터를 작은 청크로 나누고 특정 이벤트 때마다 callback 을 호출하는 방식으로 만들어야 한다.

이벤트	처리로직

데이터의 새 청크가 도착했다.	request.addListener("**data**", function(chunk) {
	// 모든 POST 데이터 청크를 모은다.
	});

모든 청크를 다 받았다.	request.addListener("**end**", function() {
	// router 를 호출하면서 모든 데이터 청크를 전달한다.
	});

이 로직을 어디에 구현해야 할까? 지금은 server 에서만 request 객체에 접근 가능하다.
현재로선 server 에 구현하는게 적합해 보인다.

request 로부터 오는 모든 data 를 처리할 함수에게 전달하는 것은 HTTP server 의 역할이다.
POST data 를 server 에서 받은 후에 router 를 거쳐 request handler 로 보내도록 한다.

/http/11/server.js

```
var http = require("http");
var url = require("url");

function start(route, handle) {
	function onRequest(request, response) {
		var postData = "";
		var pathname = url.parse(request.url).pathname;
		console.log("Request for " + pathname + " received.");

		request.setEncoding("utf8");

		request.addListener("data", function(postDataChunk) {
			postData += postDataChunk;
			console.log("Received POST data chunk '"+ postDataChunk + "'.");
		});

		request.addListener("end", function() {
			route(handle, pathname, response, postData);
		});
	}

	http.createServer(onRequest).listen(8888);
	console.log("Server has started.");
}

exports.start = start;
```

textarea 에 큰 문자열을 입력하고 submit 하면 data callback 함수가 여러 번 호출되는 모습을 볼 수 있다.

추가로 /upload 페이지에 받은 내용을 표시하도록 해 보자. router.js 를 아래와 같이 수정한다.

/http/11/router.js

```
function route(handle, pathname, response, postData) {
```

```
        console.log("About to route a request for " + pathname);

        if (typeof handle[pathname] === 'function') {
                handle[pathname](response, postData);
        } else {
                console.log("No request handler found for " + pathname);

                response.writeHead(404, {"Content-Type": "text/plain"});
                response.write("404 Not found");
                response.end();
        }
}

exports.route = route;
```

/start 요청에서 textarea 태그가 있는 HTML 을 사용자에게 제공하도록 requestHandler.js 를 수정한다. 그리고 requestHandlers.js 의 upload request handler 에서 응답할 때 이 데이터를 표시한다.

/http/11/requestHandlers.js

```
var querystring = require("querystring");

function start(response, postData) {
        console.log("Request handler 'start' was called.");

        var body = '<html>'+
                '<head>'+
                '<meta http-equiv="Content-Type" content="text/html; '+
                'charset=UTF-8" />'+
                '</head>'+
                '<body>'+
                '<form action="/upload" method="post">'+
                '<textarea name="text" rows="10" cols="40"></textarea>'+
                '<br><input type="submit" value="Submit text" />'+
                '</form>'+
                '</body>'+
                '</html>';
```

```
        response.writeHead(200, {"Content-Type": "text/html"});
        response.write(body);
        response.end();
}

function upload(response, postData) {
        console.log("Request handler 'upload' was called.");

        response.writeHead(200, {"Content-Type": "text/plain"});
        response.write("You've sent: " + postData);
        response.write("₩nYou've sent the text: "+querystring.parse(postData).text);
        response.end();
}

exports.start = start;
exports.upload = upload;
```

/start 로 접근하여 form 을 확인한다. 사용자가 이 form 을 submit 할 때 POST 요청은 upload request handler 로 전달된다.

테스트

서버를 재시작하고 결과를 확인하자.

NPM

노드는 NPM(Node Package Manager)이라는 자체의 패키지매니저를 가지고 있다.
NPM 은 노드 외부 모듈을 쉽게 설치할 수 있게 해준다. NPM 역시 하나의 노드 모듈로써 노드를 처음 설치할 때 같이 설치되는 내장모듈이므로 바로 사용할 수 있다. 기본적으로 NPM 은 레지스트리에서 패키지를 찾아서 가져온다.

npm install 명령을 내리면 **package.json** 파일에 열거된 모든 모듈을 현재 디렉토리 안에 **node_modules** 폴더를 만들고 그 안에 설치한다. package.json 파일을 사용하지 않는 경우 "npm install 모듈명" 식으로 사용하면 된다.

작업	명령
모듈 설치	npm install 모듈명
모듈 검색	npm search 모듈명
모듈 문서	npm docs 모듈명

File Upload

파일 업로드를 구현하면서 외부 library 를 설치하는 방법과 이용하는 방법을 점검 해 보자.

우리가 사용할 외부 모듈은 formidable 이다. 이 모듈은 들어오는 파일 데이터를 파싱해서 처리하는 작업을 쉽게 할 수 있도록 추상해 놓았다. Felix 의 formidable 을 사용하기 위해 모듈 설치가 필요하다.

이제 formidable 모듈을 설치해 보자.

```
npm install formidable
```

사용하는 방법은 내부 모듈을 사용할 때 했던 것처럼 require 만 해주면 된다.

```
var formidable = require("formidable");
```

/http/12/requestHandlers.js

```
var querystring = require("querystring"),
        fs = require("fs");
```

```
function start(response, postData) {
	console.log("Request handler 'start' was called.");

	var body = '<html>'+
		'<head>'+
		'<meta http-equiv="Content-Type" '+
		'content="text/html; charset=UTF-8" />'+
		'</head>'+
		'<body>'+
		'<form action="/upload" method="post">'+
		'<textarea name="text" rows="10" cols="40"></textarea>'+
		'<br><input type="submit" value="Submit text" />'+
		'</form>'+
		'</body>'+
		'</html>';

	response.writeHead(200, {"Content-Type": "text/html"});
	response.write(body);
	response.end();
}

function upload(response, postData) {
	console.log("Request handler 'upload' was called.");
	response.writeHead(200, {"Content-Type": "text/plain"});
	response.write("You've sent the text: "+
	querystring.parse(postData).text);
	response.end();
}

function show(response, postData) {
	console.log("Request handler 'show' was called.");
	fs.readFile("c:/tmp/test.jpg", "binary", function(error, file) {
		if(error) {
			response.writeHead(500, {"Content-Type": "text/plain"});
			response.write(error + "₩n");
			response.end();
		} else {
			response.writeHead(200, {"Content-Type": "image/png"});
			response.write(file, "binary");
```

```
                response.end();
            }
        });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

파일을 읽기 위해서 fs 라는 모듈을 사용한다.
처음으로 해 볼 것은 서버에 이미 존재하는 이미지 파일을 읽고 브라우저에게 전송하는 기능이다.

request handler 를 /show 라는 URL 과 매핑하도록 index.js 를 수정한다.

/http/12/index.js

```
var server = require("./server");
var router = require("./router");
var requestHandlers = require("./requestHandlers");

var handle = {}
handle["/"] = requestHandlers.start;
handle["/start"] = requestHandlers.start;
handle["/upload"] = requestHandlers.upload;
handle["/show"] = requestHandlers.show;

server.start(router.route, handle);
```

c:/tmp/test.jpg 파일을 준비하고 브라우저에 /show 라는 URL 로 접근해서 잘 보이는지 확인한다.

/start 의 form 에 파일 업로드 element 를 추가하고, 서버로 전송된 파일을 저장하기 위해서 formidable 을 upload request handler 에 추가한다. 더불어 /upload URL 의 출력 HTML 에 업로드된 이미지가 보여지도록 처리한다.

HTML form 의 encoding type 에 multipart/form-data 를 추가하고 textarea 는 삭제한다.
파일 업로드 input 필드를 추가하고 submit 버튼의 텍스트를 "Upload file"이라고 바꾼다.

/http/13/requestHandlers.js

```
var querystring = require("querystring"),
        fs = require("fs"),
        formidable = require("formidable");

function start(response) {
        console.log("Request handler 'start' was called.");

        var body = '<html>'+
                '<head>'+
                '<meta http-equiv="Content-Type" content="text/html; '+
                'charset=UTF-8" />'+
                '</head>'+
                '<body>'+
                '<form action="/upload" enctype="multipart/form-data" '+
                'method="post">'+
                '<input type="file" name="upload" multiple="multiple">'+
                '<input type="submit" value="Upload file" />'+
                '</form>'+
                '</body>'+
                '</html>';

        response.writeHead(200, {"Content-Type": "text/html"});
        response.write(body);
        response.end();
}

function upload(response, request) {
        console.log("Request handler 'upload' was called.");
        // formidable 을 사용하기 위해서 새 IncomingForm 을 생성한다.
        //이것은 submit 된 form 의 추상화 객체다.
        //이것으로 request 객체를 파싱하여 submit 된 파일과 필드들을 얻을 수 있다.
        var form = new formidable.IncomingForm();
        console.log("about to parse");

        form.parse(request, function(error, fields, files) {
                console.log("parsing done");
```

```
            fs.renameSync(files.upload.path, "c:/tmp/test.jpg");
            response.writeHead(200, {"Content-Type": "text/html"});
            response.write("received image:<br>");
            response.write("<img src='/show' />");
            response.end();
        });
}

function show(response) {
        console.log("Request handler 'show' was called.");

        fs.readFile("c:/tmp/test.jpg", "binary", function(error, file) {
                if(error) {
                        response.writeHead(500, {"Content-Type": "text/plain"});
                        response.write(error + "₩n");
                        response.end();
                } else {
                        response.writeHead(200, {"Content-Type": "image/png"});
                        response.write(file, "binary");
                        response.end();
                }
        });
}

exports.start = start;
exports.upload = upload;
exports.show = show;
```

fs.renameSync(path1, path2) 함수를 살펴보자. 이름이 암시하듯이 이 함수는 동기적으로 작업을 처리한다. 따라서, 만약 rename 동작이 비싸고 시간이 오래 걸린다면 blocking 을 초래할 수 있다는 점을 기억하자.

파일 업로드 작업을 upload request handler 에서 처리하기 위해서 request 객체를 form.parse 함수에 넘겨준다. request 객체를 server 에서 router 로 그리고 request handler 로 전달하도록 조치한다.

postData 관련 코드를 server 와 request handler 에서 삭제한다. 노드는 어떤 data 도 버퍼링 하지 않기 때문에 한번 server 에서 request 객체의 data 이벤트를 소모해버리면 나중에 form.parse 함수에서 이벤트를 받아서 처리할 수 없게 된다.

/http/13/server.js

```
var http = require("http");
var url = require("url");

function start(route, handle) {
	function onRequest(request, response) {
		var pathname = url.parse(request.url).pathname;
		console.log("Request for " + pathname + " received.");

		route(handle, pathname, response, request);
	}

	http.createServer(onRequest).listen(8888);
	console.log("Server has started.");
}

exports.start = start;
```

postData 를 넘기는 것은 더 이상 필요 없다. 대신 request 를 넘긴다.

/http/13/router.js

```
function route(handle, pathname, response, request) {
	console.log("About to route a request for " + pathname);

	if (typeof handle[pathname] === 'function') {
		handle[pathname](response, request);
	} else {
		console.log("No request handler found for " + pathname);

		response.writeHead(404, {"Content-Type": "text/html"});
		response.write("404 Not found");
		response.end();
```

```
        }
}

exports.route = route;
```

이제 request 객체를 upload request handler 함수에서 사용할 수 있게 되었다.
formidable 이 업로드되는 파일을 c:/tmp 폴더에 파일로 저장할 것이다.

테스트

서버를 재시작 하고 테스트 한다.

Express

express 는 경량의 웹 프레임워크다. 이 프레임워크에는 라우팅 설정, 템플릿 엔진, POST 리퀘스트 파싱 등의 기능이 있다. 여기서는 간단히 사용법을 살펴보고 몽고디비를 학습한 후에 심도있게 살펴 볼 것이다.

express 를 사용하기 위한 모듈을 설치한다.

```
npm install express
npm install body-parser
```

/express/1/server.js

```
var express = require('express');
var bodyParser = require('body-parser');

var app = express();
app.use(bodyParser.urlencoded({extended: false}))

app.get('/', showMovieList);
app.get('/add', showNewMovieForm);
app.post('/add', handlePostRequest);

app.listen(3000, function () {
    console.log('Server is listening @ 3000');
});

var movieList = [{title: '쥬라기 공원', director: '스티븐 스필버그'}];

function handlePostRequest(req, res) {
    var title = req.body.title;
    var director = req.body.director;
    if (!title || !director) {
        res.sendStatus(400);
    } else {
        console.log('Movie Added > title : ' + title + ' director : ' + director);
        movieList.push({title: title, director: director});
        res.redirect('/');
    }
}
```

```
function showNewMovieForm(req, res) {
        res.writeHeader(200, {'Content-Type': 'text/html; charset=utf-8'});

        var body = '<html><body>'
        body += '<h3>새 영화 입력</h3>';
        body += '<form method="post" action="add">';
        body += '<div>';
        body += '<label>영화 제목</label>';
        body += '<input type="text" placeholder="영화제목" name="title">';
        body += '</div>';
        body += '<div>';
        body += '<label>감독</label>';
        body += '<input type="text" name="director" placeholder="감독">';
        body += '</div>';
        body += '<div><input type="submit" value="add"></div>';
        body += '</form>';
        body += '</body></html>';
        res.end(body);
}

function showMovieList(req, res) {
      res.writeHeader(200, {'Content-Type': 'text/html; charset=utf-8'});

      var body = '<html><body>';
      body += '<h3>Favorite Movie</h3><hr>';
      body += '<div><ul>';

      movieList.forEach(function (item) {
            body += '<li>' + item.title + '(' + item.director + ')</li>';
      }, this);
      body += '</ul></div>';
      body += '<div><a href="add">Add Movie</a></div>';
      body += '</body></html>';
      res.end(body);
}
```

루트('/')로 접근하면 showMovieList 함수가 처리한다.

movieList 배열에 담긴 정보를 HTML 태그와 결합하여 브라우저에게 응답한다.

TIP : 한글이 깨져 보이는 현상

.js 파일에서 한글이 제대로 보이지 않는 현상
만약 브라우저에서 한글이 깨지다면 .js 파일의 저장 인코딩 방식을 확인해 볼 필요가 있다.
메모장으로 열어서 '다름 이름으로 저장' 메뉴를 선택하면 간단히 인코딩 방식을 확인해 볼 수 있다.
인코딩 방식을 'UTF-8'로 변경하고 저장하여 한글이 제대로 출력되는 확인한다.

Express + formidable 파일 업로드

/express/2/server.js

```
var express = require('express');
var formidable = require('formidable');
var pathUtil = require('path');
var fs = require('fs');

var uploadDir = __dirname + '/files';
if (!fs.existsSync(uploadDir)) {
	console.log('files 폴더가 존재하지 않습니다.');
	process.exit();
}

var app = express();
app.post('/', handlePostReqeust);
app.get('/', showMovieList);
app.get('/post', showNewMovieForm);

app.use(express.static(uploadDir));

app.listen(3001, function () {
	console.log('Server is listening @ 3001');
});

var movieList = [];

function handlePostReqeust(req, res) {
	var form = new formidable.IncomingForm();
	form.encoding = 'utf-8';
	form.uploadDir = uploadDir;
	form.keepExtensions = true;

	form.parse(req, function (err, fields, files) {
		if (err) {
			res.sendStatus(500);
			return;
```

```
                }

                var poster;
                var file = files['poster'];
                if (file && file.path) {
                        poster = pathUtil.basename(file.path);
                }

                var title = fields['title'];
                console.log('title : ' + title);
                if (!title) {
                        res.sendStatus(400);
                        return;
                }

                var info = {
                        title : title,
                        poster : poster
                }

                movieList.push(info);
                res.redirect('/');
        });
}

function showNewMovieForm(req, res) {
        res.writeHeader(200, {
                'Content-Type' : 'text/html; charset=utf-8'
        });

        var body = '<html><body>'
        body += '<h3>새 영화 입력</h3>';
        body += '<form method="post" action="." enctype="multipart/form-data">';
        body += '<div><label>영화 제목</label>';
        body += '<input type="text" placeholder="영화제목" name="title"></div>';
        body += '<div><label>포스터</label>';
        body += '<input type="file" name="poster"></div>';
        body += '<div><input type="submit" value="upload"></div>';
        body += '</form>';
```

```
        body += '</body></html>';
        res.end(body);
}

function showMovieList(req, res) {
        res.writeHeader(200, {
                'Content-Type' : 'text/html; charset=utf-8'
        });

        var body = '<html><body>'
                body += '<h3>Favorite Movie</h3>';
        body += '<div><ul>';

        movieList.forEach(function (item) {
                body += '<li>';
                if (item.poster)
                        body += '<img src="' + item.poster + '" height="100px">';
                body += item.title + '</li>';
        }, this);
        body += '</ul></div>';
        body += '<div><a href="post">Post New Movie</a></div>';
        body += '</body></html>';
        res.end(body);
}
```

Background Running

리눅스에서 Screen 은 터미널을 닫아도 프로그램이 계속 실행되도록 해주는 프로그램이다.
sudo apt-get install screen

프로그램이 server.js 파일이라면 그 파일이 있는 디렉토리로 이동해서 다음과 같이 실행한다:
$ screen
$ node server.js

취소하기 원한다면 다음과 같이 실행한다.
ctrl + a and then ctrl + d

목록을 보고자 한다면 다음과 같이 실행한다.
screen -ls

만약 실행중인 프로그램을 다시 보고 싶으면 서버로 다시 로그인해서 다음과 같은 명령을 실행한다:
$ screen -r or screen -r pid_number

이 명령은 백그라운드에서 프로그램이 실행되고 있는 쉘과 다시 연결시켜준다.
이 방법은 node 어플리케이션이 죽으면 다시 살려 주지 않기 때문에 주로 개발 시점에서 사용한다.

윈도우에서 백그라운드로 서비스하고 싶은 경우 다음 사이트를 참고하자.
http://www.coretechnologies.com/products/AlwaysUp/Apps/RunNodeJSAsAService.html

계속해서 서비스를 제공하고 싶은 경우 사용할 만한 모듈로 forever 가 있다.
매뉴얼은 다음 사이트를 참조하자.
https://www.npmjs.com/package/forever

모듈 설치
$ npm install forever -g

forever 를 사용하여 백그라운드로 서비스를 제공할 수 있다. 콘솔을 종료해도 서비스는 살아있다.
forever start app.js

기타 기능은 도움말을 참조하자.

```
$ forever --help
```

Chapter 7. Echo Server

간단한 에코 기능의 서버를 만들어 보자.

다음 예제에서는 http 모듈 대신에 좀더 로우레벨인 net 모듈을 사용한다.

다음은 서버역할을 담당할 프로그램이다.

/echo/echo.js

```
var server = require('net').createServer(function (socket) {
        // connection 이벤트 발생시 진입
        // 여기서 socket 은 stream 이다.
        socket.setEncoding('utf8');
        socket.write("type 'quit' to exit.");

        socket.on('data', function (data) {
                console.log('data from client : ', data.toString());

                if (data.trim().toLowerCase() === 'quit') {
                        socket.write('bye');
                        return socket.end();
                }
                // echo
                socket.write(data);
        });

        socket.on('end', function () {
                console.log('client > end');
        });

        socket.on('error', function (err) {
                console.error('SOCKET ERROR:', err, '₩n');
        });
});

server.on('listening', function () {
        console.log('server > listening');
});
```

```
server.on('connection', function (socket) {
        console.log('client > connection');
});

server.on('close', function () {
        console.log('server > close₩n');
});

server.on('error', function () {
        console.log('server > error : ', err.message, '₩n');
});

server.listen(7000, '127.0.0.1');
```

다음으로 클라이언트 역할을 할 프로그램을 작성하자.

/echo/client.js

```
var net = require('net');
var client = new net.Socket();

client.connect(7000, '127.0.0.1', function () {
        console.log('client > connect to server');
        client.write('hi');

        setTimeout(function(){
           client.write('quit');
      },2000);
});

client.on('data', function (data) {
        console.log('data from server : ' + data);
});

client.on('close', function () {
        console.log('client > close to server₩n');
});
```

이번에는 추가로 readline 모듈을 사용하여 사용자가 콘솔로 입력한 내용을 받아 서버로 보내고 서버에서 응답(echo)이 잘 전달되어 오는지 확인하자.

/echo/client2.js

```
var net = require('net');
var client = new net.Socket();
var readline = require('readline');
var rl = readline.createInterface(process.stdin, process.stdout);
rl.setPrompt('message> ');

client.connect(7000, '127.0.0.1', function () {
	console.log('client > connect to server');
	consoleInput();
});

function consoleInput(){
	rl.on('line', function (line) {
		if (line === "quit") {
			client.write('quit');
		}
		client.write(line);
	}).on('close', function () {
		process.exit(0);
	});
}

client.on('data', function (data) {
	console.log('data from server : ' + data);
	rl.prompt();
});

client.on('close', function () {
	console.log('client > close to server₩n');
	rl.close();
});
```

Chapter 8. 비동기 프로그래밍

Event Loop

노드를 이해하는 **가장 중요한 개념은 단일 스레드 방식**이다.

이는 앱이 한 번에 하나의 작업만을 수행할 수 있음을 의미한다. 하지만 이벤트 루프를 사용해서 멀티 스레드로 동작하는 듯한 착각을 불러일으킬 수 있다.

자바스크립트 엔진은 당장 처리하지 못하는 작업을 보관하기 위해 큐를 사용한다. 이벤트, 타이머, 인터벌, 즉시 실행 큐 등이 있다.

서버는 클라이언트들의 수많은 요청을 동시에 처리해야 한다. 노드 이전에 일반적으로 동시 처리를 위해서 사용하는 방법은 멀티스레드다. 멀티스레드는 단순하게 설명하자면 CPU 자원을 시분할 형태로 나누어 가짐으로써 여러 요청을 동시에 처리하는 것처럼 보이게 만드는 것이다.

동시 접속자 수가 많아질 수록 스레드 또한 많아지게 되고 그만큼 메모리 소비도 증가한다. 서버의 자원은 제한되어 있으므로 일정 수 이상의 스레드는 발생시킬 수 없다. 처리성능을 향상시키기 위해서는 서버를 증설해야 한다. 이는 바로 비용의 증가로 이어진다.

비동기 방식이 동기 방식과 다른 점은 요청을 하고 그 결과를 기다리지 않는다는 것이다. 기다리지 않기 위해 비동기 방식에서는 함수 요청 시 콜백함수를 넘겨주고 바로 다음 라인 코드처리를 위해 계속 진행한다. 노드는 처리가 완료되었을 때 콜백함수를 호출함으로써 처리완료를 호출 측에 통보한다.

노드는 필요하다면 내부적으로 약간의 멀티스레드(child_process)를 사용하지만 개발자의 코드는 언제나 싱글스레드위에서 동작한다.

만일 처리 작업이 CPU 를 많이 소모한다든지 대용량 파일을 처리하는 작업이라면 노드는 성능을 보장할 수 없다. 반대로 단위작업이 짧고 In/Out 이 빈번한 메시징 애플리케이션의 경우에는 노드는 고성능을 보여준다.

/eventloop/test1.js

```
console.log('cycle one - 1');

setTimeout(function(){
        console.log('another cycle - 1');
}, 100);

console.log('cycle one - 2');
```

setTimeout() 함수 내부의 코드가 마지막으로 수행된다. 이런 현상이 발생하는 이유는 setTimeout() 함수가 미래 사이클에서 수행하게 될 코드를 큐에 집어넣기 때문이다.

비동기 프로그래밍의 이해

노드 애플리케이션을 개발하면서 동기방식의 코드를 사용하지 않도록 주의해야 한다.
다음 코드를 살펴보면서 비동기 방식 프로그래밍에 대해 좀 더 이해해 보자.

/async/readFile.js

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

fs.readdir('.', function (err, filenames) {
        // non-blocking code
        console.log(filenames);
});

console.log('end of code');
```

실행결과는 다음과 같다.
end of code
['readFile.js']

비동기 프로그래밍의 기본은 비동기 이벤트 발생 시점에 완료 로직을 처리하기 위해서 콜백함수를 사용하는 것이다. 코드라인의 진행은 콜백함수와 상관없이 바로 다음으로 진행이 되므로 콘솔에 바로 'end of code'가 출력된다. 콜백함수는 처리가 완료되면 호출되고 결과가 콘솔에 출력된다.

비동기 방식의 순차처리

콜백함수가 중첩되어 사용되는 상황에서 순차처리가 필요한 경우를 살펴보자.

test_file_exist.js

```
var fs = require('fs');
var fileName = 'readme2.txt';

fs.exists(fileName, function(exists){
	if(!exists){
		return console.error('file does not exist');
	}

	fs.stat(fileName, function(error, stats){
		if(error){
			return console.error(error);
		} else if(!stats.isFile()){
			return console.error('not a file');
		}

		fs.readFile(fileName, 'utf8', function(error, data){
			if(error){
				return console.error(error);
			}
			console.log(data);
		});
	});
});
```

중첩되어 사용하는 콜백함수는 가독성을 떨어뜨린다는 단점이 있다. 콜백의 지옥을 피하는 일반적인 방법은 제어 흐름 모듈을 사용하는 것이다. 한번 더 중첩된 콜백 코드를 살펴보자.

/async/doSync.js

```
var fs = require("fs");
var oldFileName = './oldFile.txt';
var newFileName = './newFile.txt';
```

```
fs.chmod(oldFileName, 777, function (err) {
        console.log('1-start');

        fs.rename(oldFileName, newFileName, function (err) {
                console.log('2-start');

                fs.lstat(newFileName, function (err, stats) {
                        console.log('3-called');
                });

                console.log('2-end');
        });

        console.log('1-end');
});
```

실행결과

```
1-start
1-end
2-start
2-end
3-called
```

비동기 방식에서 순차처리는 중첩된 콜백함수를 사용하여 처리 순서를 보장할 수 있다.
콜백함수는 부모 함수의 처리 로직이 완료된 후에 호출되므로 콜백을 내부적으로 중첩시키면 처리 순서가 보장된다.

그러나 위의 예제처럼 중첩 Callback 을 사용하면 그 코드의 Depth 가 너무 깊어진다는 단점이 있다.
중첩 콜백을 과도하게 사용하면 코드의 가독성이 떨어진다. 이와같은 문제를 해결해 주는 프레임워크가 몇가지 있다. 그중에서 'async' 라는 모듈을 사용해보자.

Async Module

async 모듈을 사용하기 위해 먼저 모듈을 설치한다.

```
npm install async
```

/async/testAsync.js

```
var fs = require('fs');
var async = require('async');

var oldFileName = './oldFile.txt';
var newFileName = './newFile.txt';

async.waterfall([
	function (cb) {
		fs.exists(oldFileName, function(exists) {
			console.log(exists ? 'already exists' : 'not exists');
			cb(null, exists);
		});
	},
	function (exists, cb) {
		if (exists) {
			cb(null);
		} else {
			fs.writeFile(oldFileName, 'just now, we have created this file', function (err) {
				if (err) {throw err;}
				console.log('a file created in same location');
				cb(null);
			});
		}
	},
	function (cb) {
		fs.chmod(oldFileName, 777, function (err){
			console.log('1-start');
			cb(null);
		});
	},
	function (cb) {
```

```
        fs.rename(oldFileName, newFileName, function (err) {
            console.log('2-start');
            cb(null,'one', 'two');
        });
    },
    function (arg1, arg2, cb) {
        fs.lstat(newFileName, function (err, stats) {
            console.log('args: '+arg1+','+arg2);
            console.log('3-start');
        });
    }
]);
```

cb(null); 코드로 다음 순서의 함수가 기동하도록 제어한다.

async 모듈의 waterfall 함수를 사용하면 Callback 의 중첩을 줄이면서 로직의 순서를 보장할 수 있다.

콜백함수의 중첩 hell 을 방지하면서 가독성을 보장하는 좋은 방법이다.

노드의 병렬처리

Event Loop 에서 처리되는 작업이 긴 시간을 요구하는 경우 전체 서버의 성능이 저하된다.

이를 보완하기 위해서 이벤트를 잘게 쪼개어서 병렬로 처리되도록 조치해야 한다.

다음 코드는 현재 디렉토리의 파일목록을 구한 후 파일사이즈의 합을 구한다.

/async/getDirSize.js

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

function calculateBytes() {
    fs.readdir('.', function (err, filenames) {
        var count = filenames.length;
        var totalBytes = 0;
        var i;
        for (i = 0; i < filenames.length; i++) {
            fs.stat('./' + filenames[i], function (err, stats) {
                totalBytes += stats.size;
                count--;
```

```
                        if (count === 0) {
                                console.log(totalBytes);
                        }
                });
        }
    });
}

calculateBytes();
```

비동기 방식에서 각 파일의 사이즈를 구하여 이를 합치는 과정은 상당히 긴 시간을 요구하는 작업이다. 따라서 이러한 작업을 작은 단위로 나누어서 병렬처리하는 것이 좋다. 즉, 파일 목록에서 각 파일마다 비동기 이벤트를 전달하는 방식이다. 이렇게 되면 단위작업이 많아지게 되므로 전체적인 관점에서 서버의 부담은 줄어든다.

여기서 stat 함수는 파일 사이즈의 총 합을 저장하기 위하여 함수 외부에 존재하는 변수 totalBytes 에 접근한다. 이는 자바스크립트가 closure 를 지원하기 때문에 가능한 것이다. **for 루프를 돌면서 사용하는 클로저는 하나의 환경을 공유**한다.

일반적으로 하나의 작업을 병렬로 분산처리 할 경우 언제 전체 작업이 완료되는지 알 수 없다.
위의 예제는 이 문제를 해결하기 위하여 callback 이 전달된 횟수만큼 count 에 저장하고 각 작업이 완료될 때 마다 count 를 하나씩 감소시킨다. 그래서 **count 가 0 이 되면 모든 작업이 완료되었다는 것**을 파악할 수 있다. 일반적으로 병렬처리시에 count 변수를 두어 제어한다.

Callback 함수의 재사용

비동기 메시지 전달시 사용되는 콜백함수는 재사용될 수 있다.

/async/reuseCallback.js

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

var path1 = './';
var path2 = '../../';
```

```
var countCallback = function (err, path, count) {
        console.log(count + ' files in ' + path);
};

function countFiles(path, callback) {
        fs.readdir(path, function (err, filenames) {
                callback(err, path, filenames.length);
        });
}

countFiles(path1, countCallback);
countFiles(path2, countCallback);
```

일반 함수가 변수에 할당되면 필요할 때마다 다른 함수의 인자로 사용할 수 있다.
countFiles 함수는 함수를 인자로 전달받아 이를 콜백함수로 전환시키는 형태로 함수 코드를 재사용할 수 있다.

Callback 함수의 호출시점

콜백함수를 연속적으로 여러개 사용하는 경우 콜백함수는 Event Loop 에 의해 처리되므로 어느 것이 먼저 실행될 지 알 수 없다.

/async/accessOuterVar.js

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

function executeCallbacks() {
        fs.readdir('.', function (err, filenames) {
                var i;
                for (i = 0; i < filenames.length; i++) {
                        fs.stat('./' + filenames[i], function (err, stats){
                                console.log(i + ':' + stats.isFile());
                        });
                }
        });
}
```

```
executeCallbacks();
```

위의 예제에서 fs.readdir 함수의 콜백과 fs.stat 함수의 콜백이 중첩되어 사용되고 있다.
내부 콜백은 병렬처리를 위해 여러번 사용된다. 내부 콜백은 외부 콜백의 변수 i 에 접근하는데 내부 콜백에서는 모두 같은 값으로 출력된다. 예를 들어 파일 개수가 5 개라면 모두 5, 5, 5,... 로 출력된다. 이는 외부 콜백과 내부 콜백이 실행시점이 달라서 그렇다.

중첩 콜백인 경우 **외부 콜백이 모두 수행된 후 내부 콜백이 수행**된다는 점을 기억하자. for 문이 끝나는 시점에 변수 i 가 갖고 있는 값을 안쪽 콜백함수가 사용하게 된다.

/async/goodAccessOuterVar.js

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

function executeCallbacks() {
        fs.readdir('.', function (err, filenames) {
                var i;
                for (i = 0; i < filenames.length; i++) {
                        (function () {
                                var j = i;
                                fs.stat('./' + filenames[i], function (err, stats) {
                                        console.log(j + ':' + stats.isFile());
                                });
                        })();
                }
        });
}

executeCallbacks();
```

내부 콜백에서 순차적인 i 값을 사용하기 위해 외부 콜백의 i 값을 내부 콜백의 j 변수에 저장한다.
함수를 정의하고 ()를 사용하면 **즉시실행 함수**로 동작한다. 이로써 원래 의도대로 작동하는 코드를 작성했다. 이제 안쪽 콜백함수는 0 부터 1 씩 증가하는 값을 사용한다. 클로저는 함수의 실행 시점에 환경을 구성한다는 점을 상기하자.

Async Operation

자바스크립트 엔진은 당장 처리하지 못하는 작업(콜백함수)을 담아두기 위해 여러 개의 큐를 사용한다. 이벤트 루프는 개별 작업을 빠르게 수행하여 마치 멀티 스레드처럼 보이게 한다.

/event/1/test_event_loop.js

```
setInterval(function () {
        console.log('Task A');
}, 500);

setInterval(function () {
        console.log('Task B');
}, 700);
```

멀티 스레드처럼 동작하는 것으로 보이는 착각을 깨뜨리는 코드를 살펴보자.

/event/2/test_event_loop.js

```
setInterval(function () {
        console.log('Task A');
}, 500);

setInterval(function () {
        while (true);
        console.log('Task B');
}, 700);
```

'Task A'가 찍힌 후 더 이상 결과가 출력되지 않는다. 무한 루프에 빠졌다. 두 작업이 별도의 스레드라면 'Task A'가 0.5 초마다 출력이 되어야 한다. 재차 강조하고 싶다. 노드 개발자는 로직이 비동기식으로 작동하도록 코딩해야 한다.

콜백함수에게 파라미터를 전달하는 방법

예를 들어 다음 setTimeout 함수에 두 번째 파라미터로 넘긴 값을 콜백함수의 파라미터로 전달하고 싶은 경우 어떻게 처리해 할까? 다음 코드의 실행결과는 'undefined' 가 콘솔에 표시된다.

```
setTimeout(function (second) {
        console.log('second: '+second);
}, 2000);
```

위 코드를 새로운 함수로 감싸고, 감싼 함수의 파라미터로 받은 지역변수 값을 사용하는 closure 방식으로 처리할 수 있다.

/event/4/ test_event_loop.js

```
var callback = function (second) {
        console.log(second / 1000 + " seconds later...");
};

function dummy(second, callback) {
        console.log('dummy() called');

        setTimeout(function () {
                callback(second);
        }, second);
}

dummy(1000, callback);
dummy(9000, callback);
dummy(1000, callback);
```

결과

```
dummy() called
dummy() called
dummy() called
1 seconds later...
1 seconds later...
9 seconds later...
```

비동기 연산의 결과를 전달하는 방법

일반적으로 동기식으로 수행되는 언어는 멀티 스레드를 지원한다. 하지만 자바스크립트는 단일 스레드 모델이다. 일부 로직이 무한루프에 빠진다면 전체 앱이 작동을 멈춘다.

비동기 연산의 결과를 노드앱에 전달하는 인기있는 방식은 다음과 같다.

1. Call-back Function
2. Event Emitter
3. Promise

Call-back Function

다음은 fs 모듈을 사용하여 비동기식 방식으로 파일 내용을 읽는다.

/event/5/test_file_read.js

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

fs.readFile('readme.txt', 'utf8', function (error, data) {
	if (error) {
		return console.error(error);
	}
	console.log(data);
});
```

콜백함수는 비동기식 함수의 인자로 전달되며 비동기식 함수의 연산이 끝나면 연산 결과를 콜백함수의 파라미터로 받으면서 호출된다. 관례적으로 함수 파라미터 정의에서 콜백함수는 마지막 인자로 지정한다. **관례적으로 콜백함수의 파라미터 정의에서 error 정보를 담고 있는 인자는 첫 번째로 사용**한다. 이는 에러처리를 실수로 누락하는 일이 없도록 하기 위함이다.

이번에는 동기식 방식으로 파일 내용을 읽는 코드를 살펴보자.

/event/5/test_file_read_sync.js

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

try {
```

```
        var data = fs.readFileSync('readme.txt', 'utf8');
        console.log(data);
} catch (error) {
        console.error(error);
}
```

동기식 코드에서는 try 문을 사용하여 에러를 처리한다. 하지만, 비동기식 코드에서는 try 문으로 처리할 수 없다. 이미 코드는 다음으로 진행되었고 결과는 나중에 리턴되기 때문이다. **비동기식 코드에서 try 문은 존재 의미가 없다.** 그러므로 노드에서 예외처리는 콜백함수에 포함시켜서 전달하는 방식을 사용한다.

Event Emitter

비동기식 코드를 구현하는 두 번째 방법은 이벤트 전송자를 사용하는 것이다.

/eventemitter/test1.js

```
var util = require('util');
var EventEmitter = require('events').EventEmitter;

function Counter(){
        var self = this;

        // EventEmitter 생성자 호출
        EventEmitter.call(this);
        var count = 0;

        this.start = function(){
                this.emit('start');

                setInterval(function(){
                        self.emit('count', count);
                        ++count;
                }, 1000);
        };
}
```

```
util.inherits(Counter, EventEmitter);

exports.Counter = Counter;
```

/eventemitter/test2.js

```
var Counter = require('./test1').Counter;

var counter = new Counter();

counter.on('start', function(){
        console.log('start event');
});

counter.on('count', function(count){
        console.log('count: '+count);
});

counter.start();
```

util.inherits(생성자, 부모생성자) 함수를 사용하면 부모생성자로부터 상속관련 프로토타입 메소드들이 Counter.super_ 라는 프로퍼티로 생성자 함수에 추가된다.
new 키워드로 새로 객체를 만들 때 만들어지는 대상 객체는 부모생성자인 에미터에게 통보된다.
새로만들어지는 객체는 프로토타입체이닝에 따라 에미터의 프로토타입 객체에 있는 함수들을 사용할 수 있다. 위 경우는 on(), start() 함수가 이 경우에 해당한다.

Event Emitter 예외처리

/eventemitter/test3.js

```
var util = require('util');
var EventEmitter = require('events').EventEmitter;

function Counter(){
        var self = this;
```

```
        // EventEmitter 생성자 호출
        EventEmitter.call(this);
        var count = 0;

        this.start = function(){
                this.emit('start');

                var refreshIntervalId = setInterval(function(){
                        self.emit('count', count);
                        ++count;

                        if (count >= 5) {
                                self.emit('error', new Error('error event'));
                                clearInterval(refreshIntervalId);
                        }
                }, 1000);
        };
}

util.inherits(Counter, EventEmitter);

exports.Counter = Counter;
```

/eventemitter/test4.js

```
var Counter = require('./test3').Counter;

var counter = new Counter();

counter.on('start', function(){
        console.log('start event');
});

counter.on('count', function(count){
        console.log('count: '+count);
});
```

```
counter.on('error', function(error){
	console.log('error: '+error.message);
});

counter.start();
```

Promise

프라미스는 아직 알려지지 않은 값을 표현하는 객체다. 프라미스는 **비동기식 함수와 관련된 약속**으로 생각할 수 있다. 비동기식 함수는 즉시 반환되지만 **미래의 특정 시점에 어떤 값이 존재할 것임을 약속**한다. 프라미스를 처음 생성하면 미결(pending) 또는 미이행(unfulfilled) 상태가 되며 관련된 비동기식 코드가 수행을 완료할 때까지 이 상태로 남는다. 비동기식 코드가 성공적으로 완료되면 프라미스는 이행(fulfilled) 상태로 변하며 비동기식 호출이 실패하면 거절(rejected) 상태로 변한다.

/promise/test1.js

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

var promise = new Promise(function(resolve, reject){
	fs.readFile('data.txt', 'utf-8', function(error, data){
		if (error) {
			return reject(error);
		}

		resolve(data);
	});
});

promise.then(function(result){
	console.log(result);
}, function(error){
	console.log(error.message);
});
```

/promise/test2.js

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

var promise = new Promise(function(resolve, reject){
	fs.readFile('none.txt', 'utf-8', function(error, data){
		if (error) {
			return reject(error);
		}

		resolve(data);
	});
});

promise.then(function(result){
	console.log(result);
}).catch(function(error){
	console.log(error.message);
}).then(function(){
	console.log('The End');
});
```

다음 모듈을 먼저 설치하고 다음을 진행한다.

```
npm install bluebird
```

/promise/test3.js

```
var Promise = require('bluebird');
var fs = Promise.promisifyAll(require('fs'));

fs.readFileAsync('data.txt')
	.then(function(fileData){
		return fs.writeFileAsync('message.txt', fileData);
	})
	.catch(function(error){
		console.log(error);
	})
```

```
        .finally(function(){
                console.log('done');
        });
```

명령행 인자

노드 앱으로 넘어온 모든 명령행 인자는 process.argv 배열에서 얻을 수 있다.

/command/test1.js

```
process.argv.forEach(function(value, index, args){
        console.log('process.argv['+index+']='+value);
});
```

다음처럼 아규먼트를 넘겨주면서 호출해 보자.

node test1.js 10 abc

```
process.argv[0]=D:₩gangsa₩node₩tool₩node.exe
process.argv[1]=C:₩Users₩Chris₩Desktop₩sooup₩node₩download₩workspace₩chapter05₩5.1₩test1.js
process.argv[2]=10
process.argv[3]=abc
```

0: 노드의 실행파일

1: 호출된 자바스크립트 파일

파일처리

노드 앱에서는 예약어인 __filename 과 __dirname 을 사용해서 파일의 절대 위치를 파악할 수 있다.

/file/test1.js

```
console.log('__filename: '+__filename);
console.log('__dirname: '+__dirname);

console.log('process.cwd(): '+process.cwd());

try{
	process.chdir('/');
} catch(error){
	console.log(error.message);
}

console.log('process.cwd(): '+process.cwd());
```

/file/test2.js

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

fs.readFile(__filename, function(error, data){
	if (error) {
		return console.error(error.message);
	}

	// 바이트로 출력한다.
	console.log(data);
	// utf-8 로 인코딩하고 정보를 출력한다.
	console.log(data.toString());
});

fs.readFile(__filename, {encoding: 'utf-8'}, function(error, data){
	if (error) {
		return console.error(error.message);
	}
```

```
        // utf-8 로 인코딩된 정보를 출력한다.
        console.log(data);
});
```

/file/write1.js

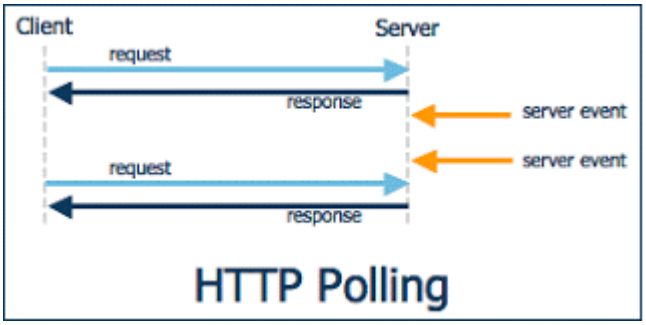
```
var fs = require('fs');
var data = 'some file data';

// 파일이 없으면 생성하고 존재하면 덮어 쓴다.
fs.writeFile(__dirname+'/foo.txt', data, function(error){
        if (error) {
                return console.error(error.message);
        }
});

//파일이 없으면 생성하고 존재하면 붙여 넣는다.
//flag:'wx'로 설정하면 파일이 이미 존재할 경우 오류를 반환한다.
fs.writeFile(__dirname+'/boo.txt', data, {flag:'a'}, function(error){
        if (error) {
                return console.error(error.message);
        }
});
```

Chapter 9. 클라이언트와 서버의 통신

클라이언트와 서버간의 연결방식

Polling

연결 상태: 연결 > 비연결 > 연결 ...



클라이언트가 처리해야 할 이벤트가 있는지를 체크하기 위해서 주기적으로 서버에 request 를 보낸다. 서버에서 응답할 것이 없다면 empty response 가 클라이언트로 전달된다.

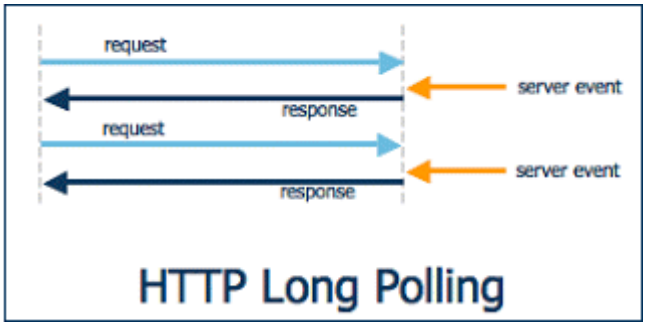
server event 가 폴링 중간에 발생한다면 다음 폴링이 이루어지기 전까지는 어떤 이벤트가 있는지를 클라이언트는 모르기 때문에 실시간성이 보장되지 않는다. 예를 들어 폴링 주기가 10 분이라고 할때 폴링 이후에 바로 이벤트가 발생해도 클라이언트가 이를 알기 위해서는 다음 폴링주기까지 기다려야 한다. 그렇다고 폴링 주기를 짧게 하면 서버의 부하가 증가하게 된다.

따라서 폴링 주기를 길게해도 무리가 없으며 실시간성이 중요하지 않은 푸시 메시지 서비스 등에 적합하다.

가장 쉬운방법이지만 클라이언트가 계속적으로 request 를 날리기때문에 클라이언가 많아지면 서버의 부담이 급증하게 된다. 연결을 맺고 끊는 빈도가 많아 클라이언트와 서버 모두 부담이 많은 방식이다.

Long Polling

연결 상태: 연결 > 재연결 > 재연결 ...



Long Polling 은 Polling 과 비슷하나 즉시성을 갖는다.

방식은 클라이언트가 HTTP request 를 보내고 바로 request 를 닫는 것이 아니라 일정 시간 동안 유지하고 있다가 서버에서 클라이언트로 보내는 메시지가 있으면 메시지를 HTTP response 로 실어 보내고 해당 Connection 을 끊는다.

만약에 일정 시간동안 보낼 메시지가 없으면 HTTP 연결을 끊는다. 응답 메시지를 받건 안받건, **끊어진 연결은 바로 다시 연결한다.**

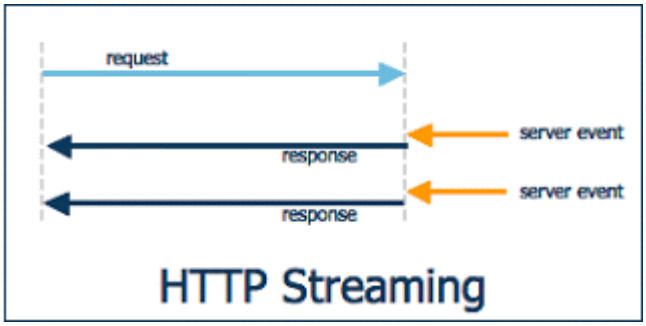
기본적으로 클라이언트가 연결해서 응답을 요청하는 Polling 형태이고 응답이 있는지 대기하는 시간이 길기 때문에 이를 Long Polling 이라고 한다.

Long Polling 의 경우 **서버와 클라이언트들이 연결되어 있는 형태이기 때문에 서버의 부하가 크다.**
서버로부터 푸시 메시지를 받으면 재 연결을 해야 하기 때문에 클라이언트로 푸시하는 빈도가 적을 경우에 유리하며 실시간 채팅과 같은 서비스에는 적합하지 않다.

일반 polling 방식보다는 서버의 부담이 줄겠지만 클라이언트로 보내는 이벤트들의 시간 간격이 좁다면 polling 과 별 차이가 없게 된다.

Streaming

연결 상태: 연결 유지 ...



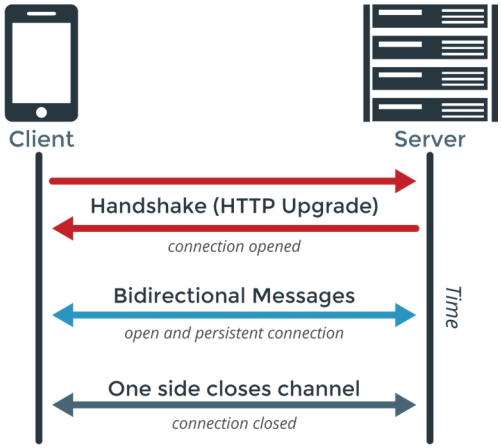
이 기법은 일반적인 TCP Connection 처럼 클라이언트가 서버로 연결을 맺은 후에 요청을 끊지 않고 메시지전송을 반복하는 방식이다.

Long Polling 이 이벤트를 받을 때마다 연결을 끊고 재 연결을 한다면 **Streaming 방식은 한번 연결되면 유지되는 연결을 통해서 이벤트 메시지를 계속해서 보내는 방식**으로 재연결에 대한 부하가 없다.

인터넷에서 동영상, 오디오 등의 파일을 실시간으로 재생해 주는 기법에 사용한다. 네트워크 속도가 중요하며 고가의 장비가 필요하다.

WebSocket

클라이언트 서버간 양방향 통신이 가능하게 하기 위해서 HTML5 표준의 일부로 webSocket 이 만들어지게 되었다.



webSocket 이 기존의 일반 TCP Socket 과 다른 점은 최초 접속이 일반 http request 를 통해 handshaking 과정을 통해 이루어 진다는 점이다. http request 를 그대로 사용하기 때문에 기존의 80, 443 포트로 접속을 하므로 추가로 방화벽을 열지 않고도 양방향 통신이 가능하고, http 규격인 CORS(Cross-Origin Resource Sharing)적용이나 인증등의 과정을 기존과 동일하게 가져갈 수 있는것이 장점이다.

그러나 다음과 같은 문제로 인해 아직 실 서비스에 적용하기에 무리가 있다.
- 오래된 버전의 웹 브라우저는 webSocket 을 지원하지 않는다.
- 웹 브라우저 이외의 클라이언트도 서비스가 제공되어야 한다.

인터넷 익스플로러 구버전 사용자들은 WebSocket 으로 작성된 웹 페이지를 볼 수가 없기 때문에 웹 서비스를 제공하는 사업자 입장에서는 치명적일 수 있다. 그래서 이를 해결하기 위해 나온 기술들이 몇가지 있는데 원리는 다음과 같다.

웹페이지가 열리는 브라우저가 webSocket 을 지원하면 일반 webSocket 방식으로 동작하고 지원하지 않는 브라우저라면 위에서 설명한 일반 http 스팩을 이용해서 실시간통신을 흉내낼수 있는 방식으로 통신을 하게 해주는 것이다.

HTML5 가 등장하면서 실시간 웹 소켓 통신인 websocket 을 내놓았지만 표준화 되기까지 아직 부족한 부분이 있다. websocket 을 다양한 브라우저가 지원을 하기 시작했고 기술적으로도 발달하였지만 아직은 테스트를 할 단계로 여겨진다.

CORS(Cross-Origin Resource Sharing)

HTTP 요청은 기본적으로 Cross-Site HTTP Requests 가 가능하다. <img> 태그로 다른 도메인의 이미지 파일을 가져오거나, <link> 태그로 다른 도메인의 CSS 를 가져오거나, <script> 태그로 다른 도메인의 JavaScript 라이브러리를 가져오는 것이 모두 가능하다.

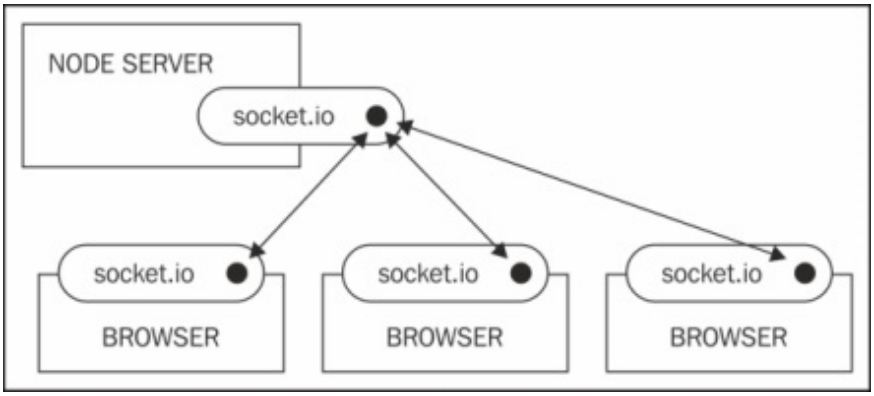
하지만 <script></script>로 둘러싸여 있는 스크립트에서 생성된 Cross-Site HTTP Requests 는 Same Origin Policy 를 적용 받기 때문에 Cross-Site HTTP Requests 가 불가능하다.

AJAX 가 널리 사용되면서 <script></script>로 둘러싸여 있는 스크립트에서 생성되는 XMLHttpRequest 에 대해서도 Cross-Site HTTP Requests 가 가능해야 한다는 요구가 늘어나자 W3C 에서 CORS 라는 이름의 권고안이 나오게 되었다.

참조: http://hanmomhanda.github.io/2015/07/21/Cross-Origin-Resource-Sharing/

Chapter 10. Socket.io

socket.io 는 자바스크립트 모듈로써 양방향 통신이 가능한 웹사이트를 구축하기 위해서 HTTP 프로토콜을 사용하여 클라이언트로 푸쉬 메시지를 보내줄 수 있는 모듈이다.



socket.io 는 node.js 에서 바로 사용할 수 있는 기술이다. 엄밀히 말하면 socket.io 가 아직 표준 기술은 아니지만 javascript 를 사용하면 node.js 에서 외부의 다른 통신요소를 거치지 않고 실시간 웹을 구현할 수 있는 기술이라서 유용하게 쓰일 수 있다.

node.js 기반으로 만들어진 기술로 자체 스팩으로 만들어진 socket.io 서버를 만들고 socket.io 클라이언트와 브라우저에 구애받지 않고 실시간 통신이 가능하다. socket.io 는 node.js 기반이기때문에 모든 코드가 javascript 로 작성되어 있다.

socket.io 는 내부적으로 다양한 방식의 클라이언트와 서버간의 연결방식을 사용한다. 그러므로 개발자는 클라이언트가 어떤 브라우저를 사용하더라도 일관되고 편리하게 푸시할 수 있는 기능을 개발할 수 있다.

socket.io 설치

~~socket.io 를 윈도우 환경에서 사용하기 위해서는 다음 3 가지가 미리 설치되어 있어야 한다.~~
~~설치 시 순서를 지켜야 한다.~~ → 노드 최신버전을 사용하면 바로 socket.io 를 설치할 수 있게 되었다.

1. Visual Studio Express 2013 for Desktop

https://www.visualstudio.com/en-us/downloads/download-visual-studio-vs.aspx

패키지매니저가 MicroSoft VisualStudio 2013 Express 버전을 사용하도록 설정한다.

```
npm config set msvs_version 2013e --global
```

2. Python 2.7.11

https://www.python.org/downloads/

패키지매니저가 파이썬 2.7 버전을 사용하도록 설정한다.

```
npm config set python python2.7
```

3.Node.js v4.2.3 LTS

https://nodejs.org/en/

socket.io 홈페이지에 접속하면 기본적인 설명과 샘플을 볼 수 있다.

http://socket.io

준비를 끝냈다면 socket.io 를 설치해 보자. 설치가 완료되면 node_modules 이라는 폴더가 생기고 그 안에 설치된 패키지가 생긴다.

```
~~npm install -g node-gyp@latest~~
~~npm install socket.io --msvs_version=2013e~~
```

노드 최신버전을 사용한다면 다음 명령으로 바로 설치가 가능하게 되었다.

```
npm install socket.io
```

서버 프로그램 작성

/socket/1/index.html

```
<html>
<body>
	<h3>welcome</h3>
	<hr>

	test socket.io
</body>
</html>
```

/socket/1/server.js

```
var app = require('http').createServer(handler),
	io = require('socket.io').listen(app),
	fs = require('fs');

app.listen(3000);

function handler(req, res) {
	fs.readFile(__dirname + '/index.html',
		function (err, data) {
			if (err) {
				res.writeHead(500);
				return res.end('Error loading index.html');
			}

			res.writeHead(200);
			res.end(data);
		}
	);
}

io.sockets.on('connection', function (socket) {
	socket.emit('news', {
		hello : 'world'
```

```
        });
        socket.on('my other event', function (data) {
                console.log(data);
        });
});
```

기본적인 웹서버를 기동하기 위해서 http 를 사용해서 서버를 생성한다.
socket.emit 함수는 접속한 대상에게 데이터를 전송한다. 데이터는 Json 포맷으로 데이터를 전송한다.
socket.on 함수는 명령 또는 데이터를 받기 위해 대기를 한다. 클라이언트로부터 'my other event' 라는 데이터를 받게 되면 콜백함수가 기동한다.

클라이언트 프로그램 작성

/socket/1/client/index.html

```
<script src="http://localhost/socket.io/socket.io.js"></script>
<script>
var socket = io.connect('http://localhost');
socket.on('news', function (data) {
        console.log(data);
        socket.emit('my other event', { my: 'data' });
});
</script>
```

테스트

위 html 파일을 작성하고 브라우저에서 기동하자.
시작되면 localhost 로 접속을 시도한다. 접속을 하면 대기하고 있던 서버에서는 'news' 데이터를 클라이언트인 웹 브라우저에 전송할 것이다. 그러면 웹 브라우저에서는 'news' 라는 데이터를 받고 그 데이터가 파라미터로 전달되면서 콜백함수를 실행한다.

콜백함수가 실행되면 socket.emit 함수를 사용하여 서버로 'my other event', { my: 'data' } 데이터를 보낸다. 서버에서는 'my other event'가 오기를 기다렸다가 데이터를 받게되면 콘솔에 출력한다.

Chapter 11. Chatting Application

1. Echo 서버

/chatting/1/server.js

```
var server = require('http').createServer();

var io = require('socket.io')(server);

io.on('connection', function (socket) {
	log(socket.id + ': a client connected');
	socket.emit('event', 'server: welcome!');

	socket.on('event', function (data) {
		log(socket.id + ': ' + data);
		socket.emit('event', 'server: '+data); // echo
	});

	socket.on('disconnect', function () {
		log(socket.id + ': a client disconnected');
	});
});

server.listen(3000);
log('server: server is running...');

function log(data){
	console.log('server: '+data);
}
```

/chatting/1/client.html

```
<html>
<head>
<script src="https://cdn.socket.io/socket.io-1.3.7.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript">
```

```
$(function(){
	var socket;
	var isConnected = false;
	var countTry = 0;

	$('#connect').click(function(){
		console.log('connect > socket: '+socket);

		if(isConnected) {

		} else {
			socket = io.connect('http://localhost:80', {'forceNew':true });
			isConnected = true;
			$('#msgError').html('');

			socket.on('connect_error', function (err) {
				$('#msgError').html(
					'server is down! fails ['+(++countTry)+']');
			});

			socket.on('event', function(data) {
				$('#msgs').append(data + '<br>');
			});
		}

	});

	$("#msgbox").keyup(function(event) {
		if(socket){
			if (event.which == 13) {
				socket.emit('event', $('#msgbox').val());
				$('#msgbox').val('');
			}
		} else {
			$('#msgError').html('connect first! click [connect] button');
		}
	});

	$('#disconnect').click(function(){
```

```
                        if(socket) {
                                socket.io.disconnect();
                                isConnected = false;
                        }
                        console.log('disconnect > socket: '+socket);

                        $('#msgError').html('disconnected to server');
                });
        });
</script>
<style>
        .warn {
                color : red;
        }
</style>
</head>
<body>
        <h3>Chat Example</h3>
        <hr>

        <button id="connect">connect</button> <button id="disconnect">disconnect</button><br>
        <label id="msgError" class="warn"></label>
        <div id="msgs"></div><br>
        <hr>

        Message: <input type="text" id="msgbox" />

</body>
</html>
```

2. 멀티유저 채팅 서버

아래 프로그램은 브라우져에 접속하면 메시지를 입력 받을 수 있는 Input Box 를 띄워주고, 여기에 메시지를 입력하면 현재 연결되어 있는 모든 클라이언트 브라우저에게 메시지를 보내주는 프로그램이다.

/chatting/2/public/index.html

```
<html>
<head>
	<title>chat sample</title>
</head>
<body>
	<h3>index.html</h3>
	<hr>

</body>
</html>
```

/chatting/2/server.js

```
var express = require('express');
var http = require('http');
var path = require('path');

var app = express();
app.use(express.static(path.join(__dirname, 'public')));

var httpServer = http.createServer(app).listen(3000, function (req, res) {
		log('server is running...');
	});

// upgrade http server to socket.io server
var io = require('socket.io').listen(httpServer);

io.sockets.on('connection', function (socket) {
	log(socket.id + ', a client connected');
```

```
	socket.emit('toclient', {
		from : 'server',
		msg : 'welcome!'
	});

	socket.on('fromclient', function (data) {
		log(socket.id + ', from: ' + data.from + ', msg: ' + data.msg);

		socket.broadcast.emit('toclient', data); // 전송자를 제외하고 다른 클라이언트에게 보낸다.
		socket.emit('toclient', data); // echo
	});

	socket.on('disconnect', function () {
		log(socket.id + ', a client disconnected');
	});
});

function log(data){
	console.log('server: '+data);
}
```

클라이언트가 접속(connection)하면 파라미터로 socket 객체를 받는다. 이 socket 객체를 사용하여 클라이언트와 대화할 수 있다. 클라이언트가 접속하면 바로 "toclient" 이벤트 명으로 환영메시지를 전송한다. 클라이언트로부터 전송되어 오는 메시지는 객체 상태로 서버에 전달되며 "fromclient" 라는 이벤트 명으로 받는다.

서버가 받은 메시지를 메시지를 전송한 클라이언트를 제외한 다른 접속상태인 클라이언트들에게 전송한다.

socket.broadcast.emit('toclient', data);

메시지를 보낸 당사자인 클라이언트에게 메시지를 전송한다. 이것은 echo 메시지이다.

socket.emit('toclient', data);

다음으로 클라이언트 프로그램을 작성한다. 자바스크립트에서 사용하는 라이브러리는 CDN 방식으로 선언한다.

/chatting/2/client.html

```
<html>
<head>
<script src="https://cdn.socket.io/socket.io-1.3.7.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript">
	$(function(){
		var socket;
		var isConnected = false;
		var countTry = 0;

		$('#connect').click(function(){
			console.log('connect > socket: '+socket);

			if(isConnected) {

			} else {
				socket = io.connect('http://localhost:3000', {'forceNew':true });
				isConnected = true;
				$('#msgError').html('');

				socket.on('connect_error', function (err) {
					$('#msgError').html(
						'server is down! fails ['+(++countTry)+']');
				});

				socket.on('toclient', function(data) {
					$('#msgs').append(data.from + ': ' + data.msg + '<br>');
				});
			}

		});

		$("#msgbox").keyup(function(event) {
			if(socket){
```

```
                        if (event.which == 13) {
                                socket.emit('fromclient', {
                                        from : 'client',
                                        msg : $('#msgbox').val()
                                });

                                $('#msgbox').val('');
                        }
                } else {
                        $('#msgError').html('connect first! click [connect] button');
                }
        });

        $('#disconnect').click(function(){
                if(socket) {
                        socket.io.disconnect();
                        isConnected = false;
                }
                console.log('disconnect > socket: '+socket);

                $('#msgError').html('disconnected to server');
        });
    });
</script>
<style>
    .warn {
        color : red;
    }
</style>
</head>
<body>
    <h3>Chat Example</h3>
    <hr>

    <button id="connect">connect</button> <button id="disconnect">disconnect</button><br>
    <label id="msgError" class="warn"></label>
    <div id="msgs"></div><br>
    <hr>
```

```
        Message: <input type="text" id="msgbox" />

</body>
</html>
```

자바 스크립트가 실행되면 socket.io 서버로 연결을 시도한다.

```
var socket = io.connect('http://localhost:8080');
```

클라이언트가 메시지를 작성하고 엔터키를 누르면 서버에 전송한다.

```
socket.emit('fromclient', {
        msg : $('#msgbox').val()
});
```

서버로부터 'toclient' 이벤트가 들어오면 메시지를 msgs 라는 id 를 갖는 div 태그에 추가한다.

```
socket.on('toclient', function(data) {
        $('#msgs').append(data.msg + '<br>');
});
```

실행해서 테스트를 해보자.

3. 클라이언트 닉네임 적용

/chatting/3/public/index.html

```
<html>
<head>
	<title>chat sample</title>
</head>
<body>
	<h3>index.html</h3>
	<hr>

	nickname:
	<form action="client" method="post">
		<input type="text" name="nickname" value="xxx">
		<input type="submit" value="connect">
	</form>
</body>
</html>
```

express 기본설정에 따라 클라이언트가 http://localhost/ 주소로 접속하면 public 폴더 밑에 있는 index.html 파일을 전송한다.

/chatting/3/public/client.html

```
<html>
<head>
<script src="https://cdn.socket.io/socket.io-1.3.7.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript">
	$(function(){
		var socket;
		var isConnected = false;
		var countTry = 0;

		$('#connect').click(function(){
			console.log('connect > socket: '+socket);
```

```
	if(isConnected) {
		// nothing to do
	} else {
		socket = io.connect('http://localhost:3000', {'forceNew':true });
		isConnected = true;
		$('#msgError').html('');

		socket.on('connect_error', function (err) {
			$('#msgError').html(
				'server is down! fails ['+(++countTry)+']');
		});

		socket.on('toclient', function(data) {
			$('#msgs').append(data.from + ': ' + data.msg + '<br>');
		});
	}

});

$("#msgbox").keyup(function(event) {
	if(socket){
		if (event.which == 13) {
			socket.emit('fromclient', {
				from : 'client',
				msg : $('#msgbox').val()
			});

			$('#msgbox').val('');
		}
	} else {
		$('#msgError').html('connect first! click [connect] button');
	}
});

$('#disconnect').click(function(){
	if(socket) {
		socket.io.disconnect();
		isConnected = false;
```

```
                        }
                        console.log('disconnect > socket: '+socket);

                        $('#msgError').html('disconnected to server');
                });
        });
</script>
<style>
        .warn {
                color : red;
        }
</style>
</head>
<body>
        <h3>Chat Example</h3>
        <hr>

        <button id="connect">connect</button> <button id="disconnect">disconnect</button><br>
        <label id="msgError" class="warn"></label>
        <div id="msgs"></div><br>
        <hr>

        Message: <input type="text" id="msgbox" />

</body>
</html>
```

/chatting/3/server.js

```
var express = require('express');
var http = require('http');
var path = require('path');
var querystring = require('querystring');

var app = express();
app.use(express.static(path.join(__dirname, 'public')));

app.post('/client', handlePostReqeust);
```

```
function handlePostReqeust(req, res) {
	var postData = '';
	req.setEncoding('utf-8');

	req.addListener('data', function(postDataChunk) {
		postData += postDataChunk;
		log('POST data chunk: '+ postDataChunk);
	});

	req.addListener('end', function() {
		log('POST data: '+ postData);
		var nickname = querystring.parse(postData)['nickname'];
		log('nickname: '+nickname);
	});

	res.sendFile(path.join(__dirname+'/public/client.html'));
}

var httpServer = http.createServer(app).listen(3000, function (req, res) {
		log('server is running...');
	});

// upgrade http server to socket.io server
var io = require('socket.io').listen(httpServer);

io.sockets.on('connection', function (socket) {
	log(socket.id + ', a client connected');

	socket.emit('toclient', {
		from : 'server',
		msg : 'welcome!'
	});

	socket.on('fromclient', function (data) {
		log(socket.id + ', from: ' + data.from + ', msg: ' + data.msg);

		socket.broadcast.emit('toclient', data); // 전송자를 제외하고 다른 클라이언트에게 보낸다.
		socket.emit('toclient', data); // echo
```

```
        })

        socket.on('disconnect', function () {
                log(socket.id + ', a client disconnected');
        });
});

function log(data){
        console.log('server: '+data);
}
```

index.html 에서 사용자가 선택한 닉네임을 서버로 전송한다. 이 닉네임을 클라이언트와 매핑시키기 위해서 다음 예제에서 쿠키를 익혀보자.

Cookie

쿠키를 적용하기 전에 간단히 사용법을 살펴보자.

/cookie/public/index.html

```
<html>
<head>
	<title>cookie example</title>
</head>
<body>
	<h3>cookie example</h3>
	<hr>

	<a href="/check">check</a><br>
	<a href="/save">save</a><br>
	<a href="/delete">delete</a><br>
</body>
</html>
```

/cookie/server.js

```
var express = require('express');
var path = require('path');
var cookieParser = require('cookie-parser');

var app = express();
app.use(express.static(path.join(__dirname, 'public')));

app.use(cookieParser());

app.get('/check', function (req, res) {
	res.status(200).send(JSON.stringify(req.cookies));
});

app.get('/save', function (req, res) {
	res.cookie('cookie_name', 'cookie_value', {maxAge : 5000});
	res.status(200).send('cookie is saved');
});
```

```
app.get('/delete', function (req, res) {
        res.clearCookie('cookie_name');
        res.status(200).send('cookie is deleted');
});

app.listen(3000);
console.log('server is running...');
```

4. 닉네임 적용(쿠키로 처리) 멀티유저 채팅 서버

/chatting/4/public/index.html 이전과 동일

```
<html>
<head>
	<title>chat sample</title>
</head>
<body>
	<h3>index.html</h3>
	<hr>

	nickname:
	<form action="client" method="post">
		<input type="text" name="nickname" value="xxx">
		<input type="submit" value="connect">
	</form>
</body>
</html>
```

/chatting/4/public/client.html

```
<html>
<head>
<script src="https://cdn.socket.io/socket.io-1.3.7.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript">
	$(function(){

		var nickname = readCookie('nickname');

		var socket;
		var isConnected = false;
		var countTry = 0;

		$('#connect').click(function(){
			console.log('connect > socket: '+socket);
```

```
		if(isConnected) {
			// nothing to do
		} else {
			socket = io.connect('http://localhost:3000', {'forceNew':true });
			if(socket){
				isConnected = true;
				if (nickname) {
					socket.emit('changenick', {
						alias : nickname
					});
				}
			}
			$('#msgError').html('');

			socket.on('connect_error', function (err) {
				$('#msgError').html(
					'server is down! fails ['+(++countTry)+']');
			});

			socket.on('toclient', function(data) {
				$('#msgs').append(data.from + ': ' + data.msg + '<br>');
			});
		}

	});

	$("#msgbox").keyup(function(event) {
		if(socket){
			if (event.which == 13) {
				socket.emit('fromclient', {
					from : 'client',
					msg : $('#msgbox').val()
				});

				$('#msgbox').val('');
			}
		} else {
			$('#msgError').html('connect first! click [connect] button');
		}
```

```
	});

	$('#disconnect').click(function(){
		if(socket) {
			socket.io.disconnect();
			isConnected = false;
		}
		console.log('disconnect > socket: '+socket);

		$('#msgError').html('disconnected to server');
	});
});

// Cookies
function createCookie(name, value, days) {
	if (days) {
		var date = new Date();
		date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
		var expires = "; expires=" + date.toGMTString();
	}
	else var expires = "";

	document.cookie = name + "=" + value + expires + "; path=/";
}

function readCookie(name) {
	var nameEQ = name + "=";
	var ca = document.cookie.split(';');
	for (var i = 0; i < ca.length; i++) {
		var c = ca[i];
		while (c.charAt(0) == ' ') c = c.substring(1, c.length);
		if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length, c.length);
	}
	return null;
}

function eraseCookie(name) {
	createCookie(name, "", -1);
}
```

```
</script>
<style>
	.warn {
		color : red;
	}
</style>
</head>
<body>
	<h3>Chat Example</h3>
	<hr>

	<button id="connect">connect</button> <button id="disconnect">disconnect</button><br>
	<label id="msgError" class="warn"></label>
	<div id="msgs"></div><br>
	<hr>

	Message: <input type="text" id="msgbox" />

</body>
</html>
```

/chatting/4/server.js

```
var express = require('express');
var http = require('http');
var path = require('path');
var querystring = require('querystring');
var cookieParser = require('cookie-parser');

var app = express();

app.use(cookieParser());

app.get('/', function (req, res) {
	res.sendFile(path.join(__dirname+'/public/index.html'));
});

app.post('/client', handlePostReqeust);
```

```
function handlePostReqeust(req, res) {
	var postData = '';
	req.setEncoding('utf-8');

	req.addListener('data', function(postDataChunk) {
		postData += postDataChunk;
	});

	req.addListener('end', function() {
		log('POST data: '+ postData);
		var nickname = querystring.parse(postData)['nickname'];

		res.cookie('nickname', nickname, {maxAge : 30*60*1000}).status(200);

		res.sendFile(path.join(__dirname+'/public/client.html'));
	});
}

var httpServer = http.createServer(app).listen(3000, '127.0.0.1', function (req, res) {
		var host = httpServer.address().address;
		var port = httpServer.address().port;
		log('server is running at http://'+host+':'+port);
	});

// upgrade http server to socket.io server
var io = require('socket.io').listen(httpServer);

io.sockets.on('connection', function (socket) {
	log(socket.id + ', a client connected');

	socket.emit('toclient', {
		from : 'server',
		msg : 'welcome!'
	});

	socket.on('fromclient', function (data) {
		var sender = socket.nickname || data.from;
		log(socket.id + ', from: ' + sender + ', msg: ' + data.msg);
```

```
            socket.broadcast.emit('toclient', {
                from : sender,
                msg : data.msg
            });

            socket.emit('toclient', {
                from : sender,
                msg : data.msg
            }); // echo
        })

        socket.on('disconnect', function () {
            log(socket.id + ', a client disconnected');

            socket.broadcast.emit('toclient', {
                from : 'server',
                msg : socket.nickname+' disconnected'
            });
        });

        socket.on('changenick', function (data) {
            //var oldNickname = socket.nickname;

            socket.nickname = data.alias;
            log('socket.nickname: '+socket.nickname);

            socket.broadcast.emit('toclient', {
                from : 'server',
                msg : socket.nickname+' connected'
            });
        });
});

function log(data){
        console.log('server: '+data);
}
```

5. 멀티룸 채팅

/chatting/5/public/index.html

```
<html>
<head>
	<title>chat sample</title>
</head>
<body>
	<h3>index.html</h3>
	<hr>

	nickname:
	<form action="rooms" method="post">
		<input type="text" name="nickname" value="xxx">
		<input type="submit" value="connect">
	</form>
</body>
</html>
```

/chatting/5/public/rooms.html

```
<html>
<head>
<title>chat sample</title>
<script src="https://cdn.socket.io/socket.io-1.3.7.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript">
	$(function(){

		$('#layoutRooms').show();
		$('#exit').show();
		$('#layoutMessage').hide();
		$('#leave').hide();

		var nickname = readCookie('nickname');

		var socket;
```

```
var isConnected = false;
var countTry = 0;

if(isConnected) {
	// nothing to do
} else {
	// 클라이언트가 F5 키를 누르면 기존 접속정보가 있는데도 다시 접속하게 된다.
	// 적절한 대응방안을 적용하자.
	socket = io.connect('http://localhost:3000', {'forceNew':true });
	if(socket){
		isConnected = true;

		if (nickname) {
			socket.emit('changenick', {
				alias : nickname
			});
		}
	}
	$('#msgError').html('');

	socket.on('connect_error', function (err) {
		$('#msgError').html(
			'server is down! fails ['+(++countTry)+']');
	});

	socket.on('toclient', function(data) {
		$('#msgs').append(data.from + ': ' + data.msg + '<br>');
	});

	socket.on('rooms', function(data) {
		/*
			{
				roomList : {
					roomName : { count : 0 }
				}
			}
		*/

		var idx = 1;
```

```
                for(var room in data.roomList) {
                        if(data.roomList.hasOwnProperty(room)) {
                                console.log(room+'/'+data.roomList[room].count);

                                var btn = '<button class="talkroom">'+room+'</button>';
                                $('#rooms').append(idx+' : '+btn+'<br>');
                                idx++;
                        }
                }

        });
}

$("#msgbox").keyup(function(event) {
        if(socket){
                if (event.which == 13) {
                        socket.emit('fromclient', {
                                from : 'client',
                                msg : $('#msgbox').val()
                        });

                        $('#msgbox').val('');
                }
        } else {
                //$('#msgError').html('connect first! click [connect] button');
        }
});

$('#exit').click(function(){
        window.location.href = 'http://localhost:3000';
});

$('#leave').click(function(){
        if(socket) {
                $('#layoutRooms').show();
                $('#exit').show();
                $('#layoutMessage').hide();
                $('#leave').hide();
```

```
                        socket.emit('leaveroom', {});
                }
        });

        $(document).on('click', '.talkroom', function(){
                var room = $(this).text();
                console.log('room selected: '+room);

                $('#layoutRooms').hide();
                $('#exit').hide();
                $('#layoutMessage').show();
                $('#leave').show();

                socket.emit('joinroom', {
                        name : room
                });
        });

});

// Cookies
function createCookie(name, value, days) {
        if (days) {
                var date = new Date();
                date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
                var expires = "; expires=" + date.toGMTString();
        }
        else var expires = "";

        document.cookie = name + "=" + value + expires + "; path=/";
}

function readCookie(name) {
        var nameEQ = name + "=";
        var ca = document.cookie.split(';');
        for (var i = 0; i < ca.length; i++) {
                var c = ca[i];
                while (c.charAt(0) == ' ') c = c.substring(1, c.length);
                if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length, c.length);
```

```
                }
                return null;
        }

        function eraseCookie(name) {
                createCookie(name, "", -1);
        }
</script>
<style>
        .warn {
                color : red;
        }
</style>
</head>
<body>
        <h3>room.html</h3>
        <hr>

        <button id="exit">exit</button><button id="leave">leave</button><br>
        <label id="msgError" class="warn"></label>
        <hr>

        <div id="layoutRooms">
                Rooms: <div id="rooms"></div>
        </div>

        <div id="layoutMessage">
                <div id="msgs"></div><br>
                <hr>

                Message: <input type="text" id="msgbox" />
        </div>

</body>
</html>
```

/chatting/5/server.js

```
var express = require('express');
var http = require('http');
var path = require('path');
var querystring = require('querystring');
var cookieParser = require('cookie-parser');

var app = express();

app.use(cookieParser());

app.get('/', function (req, res) {
	res.sendFile(path.join(__dirname+'/public/index.html'));
});

app.post('/rooms', function (req, res) {
	req.setEncoding('utf-8');
	var postData = '';

	req.addListener('data', function(chunk) {
		postData += chunk;
	});

	req.addListener('end', function() {
		var nickname = querystring.parse(postData)['nickname'];
		log('response cookie added, nickname='+ nickname);

		res.cookie('nickname', nickname, {maxAge : 60*60}).status(200);
		res.sendFile(path.join(__dirname+'/public/rooms.html'));
	});

});

var httpServer = http.createServer(app).listen(3000,
	'127.0.0.1', function (req, res) {
		var host = httpServer.address().address;
		var port = httpServer.address().port;
		log('=========================================');
```

```
        log('server is running at http://'+host+':'+port);
        log('=========================================');
    });

// upgrade http server to socket.io server
var io = require('socket.io').listen(httpServer);

// 기본적으로 존재하는 room 설정
/*
    roomList 의 구조
    room 이름 : {클라이언트 수}
*/
var roomList = {
    'room 1' : { count : 0 },
    'room 2' : { count : 0 }
};

io.sockets.on('connection', function (socket) {
    log(socket.id + ', a new client connected, total clients: '+io.engine.clientsCount);

    // 클라이언트가 접속하면 room 의 목록정보를 알려준다.
    socket.emit('rooms', {
        roomList : roomList
    });

    socket.on('fromclient', function (data) {
        var sender = socket.nickname || data.from || 'who';
        log(socket.id + '(' + sender + ') message sent: ' + data.msg);

        // 클라이언트로부터 메시지가 서버로 오면
        // 클라이언트가 참여하고 있는 방안에 있는 다른 클라이언트들에게 메시지를 전달한다.
        socket.broadcast.to(socket.room).emit('toclient', {
            from : sender,
            msg : data.msg
        });

        // echo
        socket.emit('toclient', {
            from : sender,
```

```
                msg : data.msg
        });
})

socket.on('disconnect', function () {
        log(socket.id + ', a client disconnected, total clients: '+io.engine.clientsCount);

        // 클라이언트 socket 이 room 과 바인딩된 것을 해제한다.
        socket.leave(socket.room);

        // 클라이언트의 접속종료를 방안에 있는 다른 클라이언트들에게 통보한다.
        socket.broadcast.to(socket.room).emit('toclient', {
                from : 'server',
                msg : socket.nickname+' disconnected.'
        });
});

socket.on('changenick', function (data) {
        log('socket.nickname changed: '+socket.nickname+' --> '+data.alias);
        // 클라이언트로부터 nickname 의 변경요청이 오면 업데이트한다.
        socket.nickname = data.alias;
});

socket.on('leaveroom', function () {
        log(socket.nickname+' leaved from room: '+socket.room);

        // 클라이언트 socket 이 room 과 바인딩된 것을 해제한다.
        socket.leave(socket.room);

        // 클라이언트의 접속종료를 방안에 있는 다른 클라이언트들에게 통보한다.
        socket.broadcast.to(socket.room).emit('toclient', {
                from : 'server',
                msg : socket.nickname+' leaved from this room.'
        });
});

socket.on('joinroom', function (data) {
        /*
        // param data's info
```

```
                {
                        name : 'room_name'
                }
        */
        var room = data.name;
        log(socket.nickname+' joined in room: '+room);

        // room 과 클라이언트 socket 을 바인딩한다.
        socket.join(room);
        // socket 이 참가하고 있는 room 의 이름을 저장한다.
        socket.room = room;

        roomList[socket.room].count++;

        // 방에 입장하면 환영메시지를 보낸다.
        socket.emit('toclient', {
                from : 'server',
                msg : 'welcome! you are joined in '+socket.room+'.'
        });

        // 방에 입장하면 방에 이미 있는 다른 클라이언트에게 새로운 클라이언트의 입장을 통보
        socket.broadcast.to(socket.room).emit('toclient', {
                from : 'server',
                msg : socket.nickname+' joined in this room.'
        });

        serverStatistics(socket);
    });
});

function log(data) {
    console.log('server: '+data);
}

function serverStatistics(socket) {

    // io.sockets.sockets is an array of connected sockets
    var nicknames = io.sockets.sockets.map(function(soc) {
        return soc.nickname;
```

```
        });

        // default room that each socket is automatically joined to
        var roomAry = [];
        var rooms = io.sockets.adapter.rooms; // or socket.rooms

        log('rooms: '+JSON.stringify(rooms));
        /*
        방이 2 개 만들어진 후 rooms 의 상태이다.
        따라서 io.sockets.adapter.rooms['room_name'] 코드로 해당 room 의 객체를 얻을 수 있다.
        server: rooms: {
                "3MC4OVj6NDzCZBnkAAAA":{"3MC4OVj6NDzCZBnkAAAA":true},
                "room 1":{"3MC4OVj6NDzCZBnkAAAA":true},
                "NQuMCdCTh4AA2s54AAAB":{"NQuMCdCTh4AA2s54AAAB":true},
                "room 2":{"NQuMCdCTh4AA2s54AAAB":true}}
        */
        for (var room in rooms) {
                if (rooms.hasOwnProperty(room)) {
                        roomAry.push(room);
                }
        }

        // list all rooms this socket is in
        var myRooms = io.sockets.connected[socket.id].rooms;
        log('my rooms: '+JSON.stringify(myRooms));
        /*
                my rooms: ["pLyc8ga6xL-55mAQAAAA"]
        */

        log('----------------------------------------');
        log('connected client₩'s count: '+io.engine.clientsCount);
        log('connected client₩'s nicknames: '+nicknames);
        log('server active rooms: '+roomAry);
        log('this socket active rooms: '+myRooms);
        log('----------------------------------------');
}
```

6. 채팅 애플리케이션 완성

/chatting/6/public/index.html 이전과 동일

```
<html>
<head>
	<title>chat sample</title>
</head>
<body>
	<h3>index.html</h3>
	<hr>

	nickname:
	<form action="rooms" method="post">
		<input type="text" name="nickname" value="xxx">
		<input type="submit" value="connect">
	</form>
</body>
</html>
```

/chatting/6/rooms.html

```
<html>
<head>
<title>chat sample</title>
<script src="https://cdn.socket.io/socket.io-1.3.7.js"></script>
<script src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.4/jquery.min.js"></script>
<script type="text/javascript">
	$(function(){

		$('#layoutRooms').show();
		$('#exit').show();
		$('#layoutMessage').hide();
		$('#leave').hide();

		var nickname = readCookie('nickname');

		var socket;
```

```
var isConnected = false;
var countTry = 0;

if(isConnected) {
	// nothing to do
} else {
	// 클라이언트가 F5 키를 누르면 기존 접속정보가 있는데도 다시 접속하게 된다.
	// 적절한 대응방안을 적용하자.
	socket = io.connect('http://localhost:3000', {'forceNew':true });
	if(socket){
		isConnected = true;

		if (nickname) {
			socket.emit('changenick', {
				alias : nickname
			});
			$('.alias').text(nickname);
		}
	}
	$('#msgError').html('');

	socket.on('connect_error', function (err) {
		$('#msgError').html(
			'server is down! fails ['+(++countTry)+']');
	});

	socket.on('toclient', function(data) {
		if(data.from === 'server') {
			if(data.category && data.category === 'duplicated') {
				window.location.href = 'http://localhost:3000';
				alert('nickname is duplicated, try again!');
			} else {
				$('#msgs').append('<label class="blue">'+
					data.from + ': ' + data.msg + '</label><br>');
			}
		} else {
			if(data.category) {
				$('#msgs').append('<label class="red">'+
					data.from + ': ' + data.msg + '</label><br>');
```

```
                        } else {
                                $('#msgs').append(data.from + ': ' + data.msg + '<br>');
                        }
                }
        });

        socket.on('rooms', function(data) {
                /*
                        {
                                roomList : {
                                        roomName : { count : 0 }
                                }
                        }
                */

                var idx = 1;
                for(var room in data.roomList) {
                        if(data.roomList.hasOwnProperty(room)) {
                                console.log(room+'/'+data.roomList[room].count);

                                var btn = '<button class="talkroom">'+room+'</button>';
                                $('#rooms').append(idx+' : '+btn+'<br>');
                                idx++;
                        }
                }

        });
}

$("#msgbox").keyup(function(event) {
        if(socket){
                if (event.which == 13) {
                        var to = '';
                        var message = $('#msgbox').val();
                        if (message.indexOf('/') !== -1 && message.indexOf(' ') !== -1) {
                                var msgAry = message.split(' ');
                                message = '';
                                for(var i=0; i < msgAry.length; i++) {
                                        var chunk = msgAry[i];
```

```
                                        console.log('chunk: '+chunk);
                                        if(chunk.indexOf('/') !== -1) {
                                                to = chunk.replace('/', '');
                                        } else {
                                                message += chunk+' ';
                                        }
                                }
                        }

                        /*
                        console.log({
                                to : to,
                                from : 'client',
                                msg : message.trim()
                        });
                        */
                        socket.emit('fromclient', {
                                to : to,
                                from : 'client',
                                msg : message
                        });

                        $('#msgbox').val('');
                }
        } else {
                //$('#msgError').html('connect first! click [connect] button');
        }
});

$('#exit').click(function(){
        window.location.href = 'http://localhost:3000';
});

$('#leave').click(function(){
        if(socket) {
                $('#layoutRooms').show();
                $('#exit').show();
                $('#layoutMessage').hide();
                $('#leave').hide();
```

```
                socket.emit('leaveroom', {});
            }
        });

        $(document).on('click', '.talkroom', function(){
            var room = $(this).text();
            console.log('room selected: '+room);

            $('#layoutRooms').hide();
            $('#exit').hide();
            $('#layoutMessage').show();
            $('#leave').show();

            socket.emit('joinroom', {
                name : room
            });
        });

});

// Cookies
function createCookie(name, value, days) {
    if (days) {
        var date = new Date();
        date.setTime(date.getTime() + (days * 24 * 60 * 60 * 1000));
        var expires = "; expires=" + date.toGMTString();
    }
    else var expires = "";

    document.cookie = name + "=" + value + expires + "; path=/";
}

function readCookie(name) {
    var nameEQ = name + "=";
    var ca = document.cookie.split(';');
    for (var i = 0; i < ca.length; i++) {
        var c = ca[i];
        while (c.charAt(0) == ' ') c = c.substring(1, c.length);
```

```
                        if (c.indexOf(nameEQ) == 0) return c.substring(nameEQ.length, c.length);
                }
                return null;
        }

        function eraseCookie(name) {
                createCookie(name, "", -1);
        }
</script>
<style>
        .warn {
                color : red;
        }
        label.blue {
                color : blue;
        }
        label.red {
                color : red;
        }
</style>
</head>
<body>
        <h3>room.html</h3>
        <hr>

        <button id="exit">exit</button><button id="leave">leave</button>
        your nickname is <label class="alias">Client</label><br>
        <label id="msgError" class="warn"></label>
        <hr>

        <div id="layoutRooms">
                Rooms: <div id="rooms"></div>
        </div>

        <div id="layoutMessage">
                <div id="msgs"></div><br>
                <hr>

                <label class="alias">Message</label>: <input type="text" id="msgbox" />
```

```
        </div>

</body>
</html>
```

/chatting/6/server.js

```
var express = require('express');
var http = require('http');
var path = require('path');
var querystring = require('querystring');
var cookieParser = require('cookie-parser');

var app = express();

app.use(cookieParser());

app.get('/', function (req, res) {
        res.sendFile(path.join(__dirname+'/public/index.html'));
});

app.post('/rooms', function (req, res) {
        req.setEncoding('utf-8');
        var postData = '';

        req.addListener('data', function(chunk) {
                postData += chunk;
        });

        req.addListener('end', function() {
                var nickname = querystring.parse(postData)['nickname'];
                log('response cookie added, nickname='+ nickname);

                res.cookie('nickname', nickname, {maxAge : 60*60}).status(200);
                res.sendFile(path.join(__dirname+'/public/rooms.html'));
        });

});
```

```
var httpServer = http.createServer(app).listen(3000,
        '127.0.0.1', function (req, res) {
                var host = httpServer.address().address;
                var port = httpServer.address().port;
                log('=========================================');
                log('server is running at http://'+host+':'+port);
                log('=========================================');
        });

// upgrade http server to socket.io server
var io = require('socket.io').listen(httpServer);

// 기본적으로 존재하는 room 설정
// TODO: 클라이언트가 방 추가
/*
        room 이름 : {클라이언트 수}}
*/
var roomList = {
        'room 1' : { count : 0 },
        'room 2' : { count : 0 }
};

/*
        nickname : { id : socket.id }
*/
var clientList = {
        'server' : { id : '' }
}

io.sockets.on('connection', function (socket) {
        log(socket.id + ', a new client connected, total clients: '+io.engine.clientsCount);

        // 클라이언트가 접속하면 room 의 목록정보를 알려준다.
        socket.emit('rooms', {
                roomList : roomList
        });

        socket.on('fromclient', function (data) {
```

```
var sender = socket.nickname || data.from || 'who';
log(socket.id + '(' + sender + ') message sent: ' + data.msg);

// TODO: 귓속말
if(data.to !== '') {
	var target = clientList[data.to];
	if(target) {
		if (io.sockets.connected[target.id]) {
			io.sockets.connected[target.id].emit('toclient', {
				category : 'private',
				from : sender,
				msg : data.msg
			});
		}
	} else {
		socket.emit('toclient', {
			from : 'server',
			msg : '/'+data.to+' target does not exist!'
		});
		return false;
	}

	// echo
	socket.emit('toclient', {
		category : 'private',
		from : sender,
		msg : data.msg
	});
} else {
	// 클라이언트로부터 메시지가 서버로 오면
	// 클라이언트가 참여하고 있는 방안에 있는 다른 클라이언트들에게
	// 메시지를 전달한다.
	socket.broadcast.to(socket.room).emit('toclient', {
		from : sender,
		msg : data.msg
	});

	// echo
	socket.emit('toclient', {
```

```
                from : sender,
                msg : data.msg
            });
        }

})

socket.on('disconnect', function () {
        log(socket.id + ', a client disconnected, total clients: '+io.engine.clientsCount);

        // 클라이언트 socket 이 room 과 바인딩된 것을 해제한다.
        socket.leave(socket.room);

        // 클라이언트의 접속종료를 방안에 있는 다른 클라이언트들에게 통보한다.
        socket.broadcast.to(socket.room).emit('toclient', {
            from : 'server',
            msg : socket.nickname+' disconnected.'
        });

        delete clientList[socket.nickname];
});

socket.on('changenick', function (data) {
        log('socket.nickname change asked: '+socket.nickname+' --> '+data.alias);

        if(isDuplicated(data.alias)) {
            log('socket.nickname change failed: it is duplicated');

            socket.emit('toclient', {
                category : 'duplicated',
                from : 'server',
                msg : 'nickname is duplicated. try again!'
            });
        } else {
            // 클라이언트로부터 nickname 의 변경요청이 오면 업데이트한다.
            socket.nickname = data.alias;

            // TODO: 중복체크, 덮어쓰기
            clientList[socket.nickname] = { id : socket.id };
```

```
        }

});

function isDuplicated(alias) {
        var nicknames = Object.keys(clientList).map(function (key) {
                return key.toString();
        });

        for(var i=0; i < nicknames.length; i++) {
                if(nicknames[i] === alias) {
                        return true;
                }
        }
        return false;
}

socket.on('leaveroom', function () {
        log(socket.nickname+' leaved from room: '+socket.room);

        // 클라이언트 socket 이 room 과 바인딩된 것을 해제한다.
        socket.leave(socket.room);

        // 클라이언트의 접속종료를 방안에 있는 다른 클라이언트들에게 통보한다.
        socket.broadcast.to(socket.room).emit('toclient', {
                from : 'server',
                msg : socket.nickname+' leaved from this room.'
        });
});

socket.on('joinroom', function (data) {
        /*
        // param data's info
                {
                        name : 'room_name'
                }
        */
        var room = data.name;
        log(socket.nickname+' joined in room: '+room);
```

```
		// room 과 클라이언트 socket 을 바인딩한다.
		socket.join(room);
		// socket 이 참가하고 있는 room 의 이름을 저장한다.
		socket.room = room;

		roomList[socket.room].count++;

		// 방에 입장하면 환영메시지를 보낸다.
		socket.emit('toclient', {
			from : 'server',
			msg : 'welcome! you are joined in '+socket.room+'.'
		});

		// 방에 입장하면 방에 이미 있는 다른 클라이언트에게 새로운 클라이언트의 입장을 통보
		socket.broadcast.to(socket.room).emit('toclient', {
			from : 'server',
			msg : socket.nickname+' joined in this room.'
		});

		serverStatistics(socket);
	});
});

function log(data) {
	console.log('server: '+data);
}

function serverStatistics(socket) {

	// io.sockets.sockets is an array of connected sockets
	var nicknames = io.sockets.sockets.map(function(soc) {
		return soc.nickname;
	});

	// default room that each socket is automatically joined to
	var roomAry = [];
	var rooms = io.sockets.adapter.rooms; // or socket.rooms
```

```
        log('rooms: '+JSON.stringify(rooms));
        /*
        방이 2 개 만들어진 후 rooms 의 상태이다.
        따라서 io.sockets.adapter.rooms['room_name'] 코드로 해당 room 의 객체를 얻을 수 있다.
        server: rooms: {
                "3MC4OVj6NDzCZBnkAAAA":{"3MC4OVj6NDzCZBnkAAAA":true},
                "room 1":{"3MC4OVj6NDzCZBnkAAAA":true},
                "NQuMCdCTh4AA2s54AAAB":{"NQuMCdCTh4AA2s54AAAB":true},
                "room 2":{"NQuMCdCTh4AA2s54AAAB":true}}
        */
        for (var room in rooms) {
                if (rooms.hasOwnProperty(room)) {
                        roomAry.push(room);
                }
        }

        // list all rooms this socket is in
        var myRooms = io.sockets.connected[socket.id].rooms;
        log('my rooms: '+JSON.stringify(myRooms));
        /*
                my rooms: ["pLyc8ga6xL-55mAQAAAA"]
        */

        log('----------------------------------------');
        log('connected client₩'s count: '+io.engine.clientsCount);
        log('connected client₩'s nicknames: '+nicknames);
        log('server active rooms: '+roomAry);
        log('this socket active rooms: '+myRooms);
        log('----------------------------------------');
}
```

확장포인트

- 룸을 사용자가 생성하는 기능제공
- 다른 룸에 있는 사용자에게 귓속말 기능 추가
- 사용자 닉네임을 쿠키 대신 세션으로 처리

Socket.IO 서비스의 확장 : 클러스터링

여러개의 프로세스를 이용하면서 노드를 운영하기 위해서 내부적으로 redis store 를 사용할 수 있다. redis 에는 publish/subscribe 라는 기능이 있다. 하나의 node 프로세스에서 메시지를 보내면 다른 프로세스로 redis 를 통해서 메시지를 전달한다. 이때 메시지를 보내는 프로세스는 redis 에 메시지를 publish 하고 나머지 프로세스들은 subscribe 를 이용하여 메시지를 읽는다. 이때, 메시지를 전달하는 채널은 dispatch 라는 이름의 채널을 이용한다.

그리고, cluster 모듈을 이용하거나 앞단에 nginx 로드밸런서를 이용하여 여러개의 노드 프로세스에 대한 end point 를 하나로 묶으면 대규모 분산 서비스를 할 수 있는 socket.io 클러스터를 구성할 수 있다.

https://github.com/socketio/socket.io-redis

Chapter 12. Miscellaneous

nodemon

매번 코드를 수정하고 다시 서버를 기동하는 일은 어찌보면 성가신 일이다.
코드변경을 감지해 자동으로 프로세스를 다시 시작하는 모듈을 써보자.

```
npm install -g nodemon
```

/monitor/test1.js

```
var http = require("http");

http.createServer(function(request, response) {
        response.writeHead(200, {
                "Content-Type" : "text/plain"
        });
        response.write("hello world");
        response.end();
}).listen(3000);
```

실행 시 node 명령어 대신 nodemon 을 사용한다.

```
nodemon server.js
```

서버를 시작하고 브라우저로 접속하여 메시지를 확인한다.
코드를 변경하고 다시 브라우저로 접속하여 메시지가 변경되었는지 확인한다.

콘솔 로그

```
[nodemon] 1.9.1
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node test1.js`
[nodemon] restarting due to changes...
[nodemon] starting `node test1.js`
```