01. Node.js

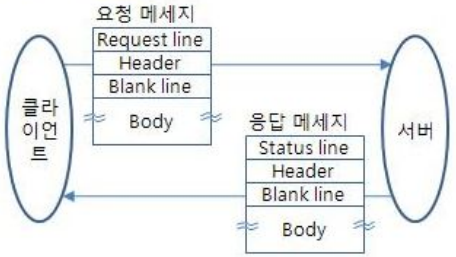
백성애 2025-05-12

1주차 커리큘럼

1일차	2일차	3일차	4일차	5일차
Node.js 개발환경 설정 노드의 동작방식 노드 내장 객체 (global, console,process,module,exports,require 등) 노드의 내장 모듈 사용하기 (os,path,url,querystring 등) 파일 시스템 접근하기	웹서버와 Node.js http 모듈로 서버 만들기 express 웹서버 만들기 express의 req/res middleware 자주 사용하는 미들웨어 (morgan,static 등) Router 객체로 라우팅 분리	Node 템플릿 엔진 (ejs) 사용한 간단한 예제 MySQL 설치 데이터베이스 및 테이블 생성 기본 SQL 문 (DDL,DML,DQL 등) 활용 Node와 MySQL 연동 RESTful 웹 서비스 http 요청 메서드 (GET, POST,PUT,DELETE 등)	React +Node +MySQL 연동 REST API 설계하기 Talend API 테스터로 테스트 CORS 이해하기 React 로 POST 게시판 UI 구성 Post 글쓰기 multer모듈을 이용한 파일 업로드 처리	Post 글목록 조회 Post 글 수정 및 삭제 처리 Post 페이징 처리 및 검색 기능 구현 회원가입 처리 bcrypt로 비밀번호 암호화 아이디 중복 체크

2주차 커리큘럼

6일차	7일차	8일차	9일차	10일차
로그인 처리 JWT 인증 인가 처리 흐름 이해하기 JWT 토큰으로 인증하기 (accessToken/refreshToken 발급) localStorage, sessionStorage 활용 인증 요청시 토큰 검증	Post 댓글 쓰기 Post 댓글 목록 조회 모달로 UI구성 웹소켓 이용한 간단한 채팅 구현하기	Sequelize 소개 ORM 개념, 장단점 소개 모델 정의-> 간단한 CRUD 구현 mysql2방식과 Sequelize 방식의 차이점 비교	Next.js 소개 Next.js 개발환경 구축 CSR vs SSR개념 이해 Pages 라우팅 동적 라우팅 및 링크 간단한 블로그 구현 스타일링	데이터 페칭 이해 getStaticProps, getStaticPaths, getServerSideProps 등 Next.js와 Node.js 연동 간단한 게시판 구현하기



웹 서비스 구조 (full stack)

Front End

클라이언트(브라우즈)

http://abc.co.kr/a.html

a.html

조회 ③

버튼 클릭

Font End : 웹브라우저를 대상으로 사용자에게 보여지는 부분(주로 JS로 작성됨)

React.js/Vue.js/Angular
Next.js

Transfer
(Http프로토콜)

①Request

② Response

Back End

웹서버

a.html

http에 의한 단순한 요청-정적인 페이지 요청 등에 대한 처리

WAS
(웹 애플리케이션 서버)

서버 프로그램

처리 로직
DB 조회

Back End : 동적인 페이지 요청 처리 로직/
데이터 처리 로직 등

JSP/Servlet/Node.js/PHP

SQL언어

데이터 베이스 서버

Node.js 란?

- CommonJS 표준과V8 Runtime을 기반으로 Ryan Dahl이 Node.js 개발
- Node.js는 자바스크립트로 서버 사이드(백엔드) 프로그래밍을 할 수 있게 해주는 실행 환경을 말한다

브라우저 밖에서도 자바스크립트를 실행할 수 있게 만든 환경이다.

Chrome 브라우저의 V8 자바스크립트 엔진 위에서 동작한다.

자바스크립트를 이용해 **웹 서버, API 서버, CLI 도구** 등을 만들 수 있다.

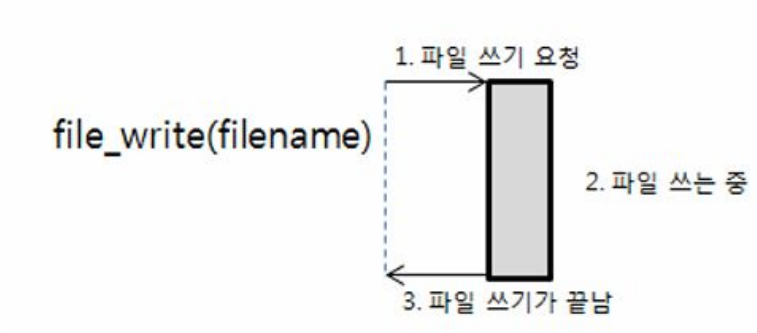
비동기 (Asynchronous) 처리와 **이벤트 기반 (Event-driven) 구조**가 특징.

Node.js 의 주요 특징

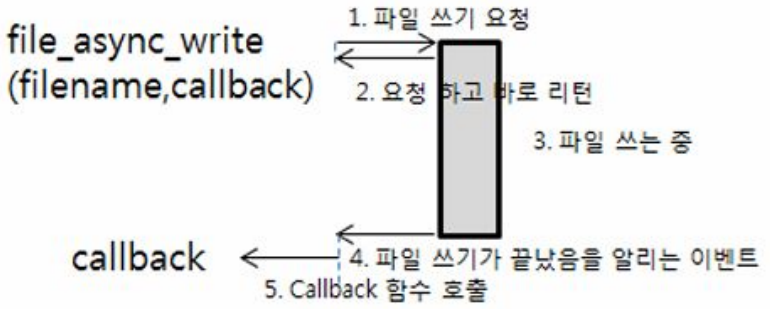
- **모듈 기반** (필요한 것들을 추가하여 사용할 수 있다)
  - NPM (Node Package Manager) 생태계. 수십만 개의 오픈 소스 패키지 사용 가능
- **Non-Blocking I/O**
  - 데이터 입출력을 기다리지 않고 처리 가능 (빠름)
- **Event-Driven 방식**
  - 요청/응답 등을 이벤트처럼 처리하여 효율적
- **크로스 플랫폼**
  - Windows, macOS, Linux 등 어디서나 실행 가능

Node.js 의 주요 특징

이벤트 기반 및 논블로킹 : Node.js는 간단한 요청(논블로킹)과 복잡한 요청(블로킹)을 모두 비동기적으로 처리한다. 즉, 각 요청이 완료될 때까지 기다리지 않고 다음 요청으로 넘어가면서 동시에 요청을 처리한다.



동기방식의 파일 입출력



node.js의 비동기 방식 파일 입출력 & 콜백함수

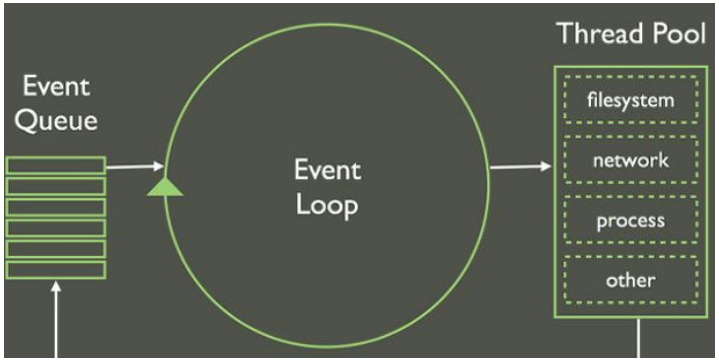
Node.js 의 주요 특징

이벤트 루프를 사용한 단일 스레드: 멀티스레드 서버와 달리 Node.js는 이벤트 루프를 사용하는 단일 스레드를 사용한다. 멀티스레드 아키텍처는 장점이 있지만, 단일 스레드는 스레드 간 컨텍스트 전환 오버헤드를 제거한다.

이벤트 루프는 이벤트 큐에서 순서대로 작업을 꺼내 실행하는 "스케줄러의 역할"을 하는 반복 루틴이다.

js코드를 실행 하다 비동기 함수(fs.readFile(), setTimeout() 등)를 만나면 그 작업은 스레드풀이 처리하고, 완료되면 콜백함수를 이벤트 큐에 등록하고 대기한다.

이벤트 루프는 이벤트 큐를 감시하다가 메인스레드가 비면(즉 지금 실행 중인 함수가 없다면) 콜백을 꺼내 실행한다



Node.js 아키텍처 구성요소

구성요소	설명
1. V8 엔진	Google Chrome에서 사용하는 자바스크립트 엔진. 자바스크립트를 **빠르게 실행**
2. 이벤트 루프 (Event Loop)	요청을 순서대로 관리하고, **블로킹 없이** 처리함
3. 이벤트 큐 (Event Queue)	이벤트(요청)가 모이는 대기열. 루프가 하나씩 처리
4. 스레드 풀 (Thread Pool)	시간이 오래 걸리는 작업(fs, DB 등)은 **libuv**가 백그라운드에서 처리
5. 모듈 시스템 (fs, http 등)	Node.js가 제공하는 다양한 기능들. 예: 파일 읽기, 서버 만들기 등

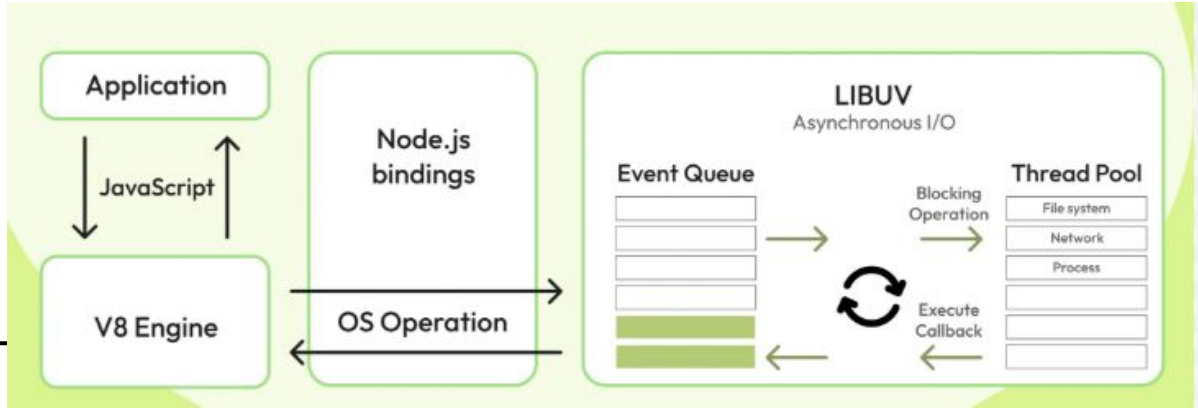


그림 출처
https://www.brilworks.com/blog/node-js-architecture-best-practices/

Node.js 동작 방식 (예시)

1. 자바스크립트 코드 → V8 엔진이 실행
2. 파일 읽기는 느리니까 → 이벤트 루프가 libuv에게 맡김
3. libuv 스레드 풀이 파일 읽음 (비동기 처리)
4. 완료되면 결과를 이벤트 큐에 전달
5. 이벤트 루프가 큐를 확인하고 콜백 실행 → 사용자에게 응답

참고: LIBUV는 Linux, Windows, macOS, Android 등 다양한 플랫폼에서 비차단 I/O 작업을 위한 추상화를 제공하는 C 라이브러리다

Node.js 단일 스레드 아키텍처

Node.js는 자바스크립트 코드를 실행할 때는 싱글 스레드이지만,

백그라운드 작업은 libuv의 스레드 풀을 이용해 멀티 스레드로 처리한다

💻 1. 싱글 스레드 (메인 스레드)

- Node.js가 자바스크립트 코드를 실행하는 **기본 실행 환경**
- 이 안에서 **이벤트 루프**가 동작
- 예: 변수 선언, 조건문, `console.log()` 같은 일반 로직 실행

> 즉, "코드를 읽고 실행하는 뇌는 하나"

⚙️ 2. 스레드 풀 (Thread Pool)

- **시간이 오래 걸리는 작업**(I/O, 파일 읽기/쓰기, DNS 조회 등)은
  메인 스레드를 멈추게 하면 안 되기 때문에
  백그라운드에서 별도 스레드들이 처리함
- 이 백그라운드 작업용 스레드들이 모여 있는 공간이 **스레드 풀**
- Node.js에서는 기본적으로 **4개의 스레드**로 구성 (`UV_THREADPOOL_SIZE` 환경 변수로 조절 가능)

> 즉, "힘든 일은 뇌가 시키고 백업 팀이 대신 처리"

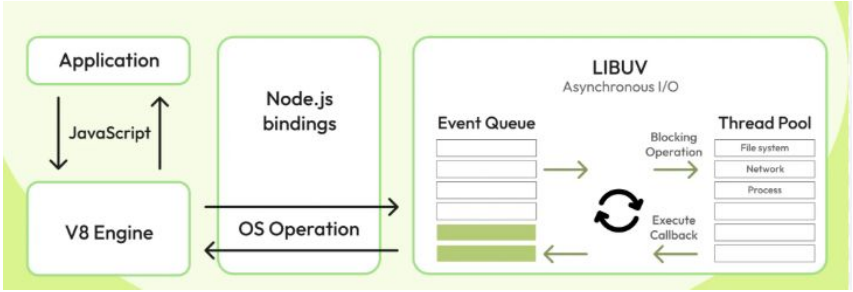
실제 흐름 예시

```js
fs.readFile('file.txt', (err, data) => {
  console.log(data.toString());
});
```

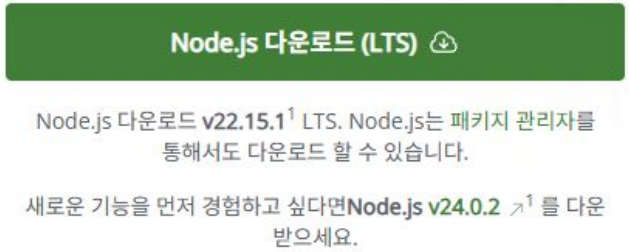
1. `readFile()` 이 호출되면, Node.js는 이 작업을 바로 처리하지 않고
2. libuv를 통해 **스레드 풀에게 파일 읽기 작업을 위임**
3. 스레드 풀이 백그라운드에서 파일을 읽고 완료되면
4. **이벤트 큐에 완료 콜백 등록 → 이벤트 루프가 콜백 실행**

👉 이 전체 과정에서 메인 스레드는 멈추지 않고 계속 다음 일을 처리함

구분	설명
싱글 스레드	Node.js는 자바스크립트 실행을 하나의 스레드로 처리
멀티 스레드(스레드 풀)	I/O 작업 등은 libuv의 스레드 풀(기본 4개)이 처리
장점	높은 처리 성능, 논블로킹, 실시간성



Node.js 설치



https://nodejs.org/ko

- LTS vs Current
  - LTS (Long-Term Support) : 장기 지원을 약속하는 버전
  - Current : 가장 최신 버전
- 홀수 버전 vs 짝수 버전
  - 홀수 버전 : 개발 버전
  - 짝수 버전 : 릴리즈 버전
- 실제 서비스에 적용할 예정이면 LTS 버전 중, 가장 최신으로 선택 권장

Node.js, npm 버전 확인

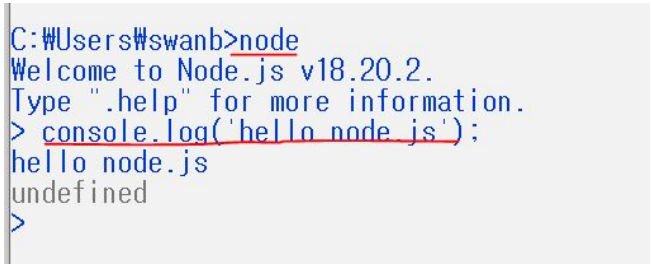
Command Shell에서

node –v

npm -v

Node REPL (Read Eval Print Loop)

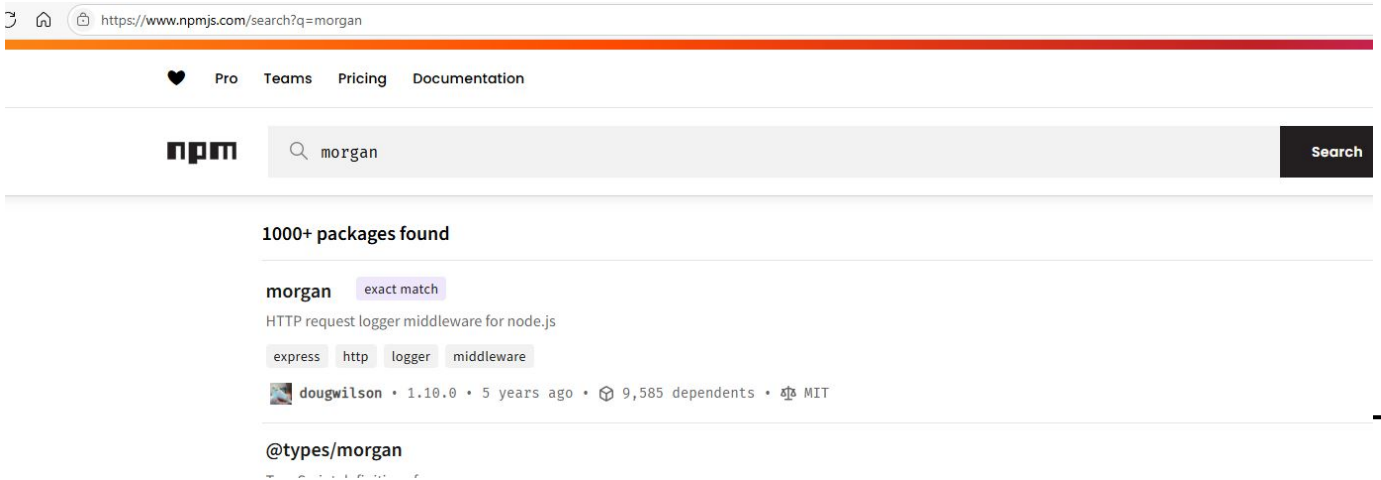
: 명령줄에서 직접 커맨드를 입력하고 코드를 테스트할 수 있는 라인 개발환경



NPM (Node Package Manager)

npm | Home

Node.js의 패키지(모듈)를 설치하고 관리하는 도구
즉. node.js의 앱스토어 같은 역할을 한다



NPM (Node Package Manager)

NPM 주요 명령어

명령어	설명
init	패키지 정보 설정
install	패키지 설치
uninstall	패키지 삭제
update	패키지 업데이트
search	패키지 검색

NPM (Node Package Manager)

npm init

- npm init

- 개발자로부터 애플리케이션의 정보를 입력받고
- 피키지 관리를 위한 package.json 파일을 생성

npm search

- npm search {패키지명}

- 해당 패키지명을 포함한 패키지를 검색
- 예) npm search express

NPM (Node Package Manager)

npm install

- npm install {패키지명}
  - 패키지명을 가진 모듈을 설치
- npm install {패키지명} -g
  - 패키지명을 가진 모듈을 글로벌로 설치
- --save 옵션을 사용하면 package.json 파일 내 dependencies 항목을 자동으로 갱신함

NPM (Node Package Manager)

npm uninstall

- npm uninstall {패키지명}
- {패키지명}을 가진 모듈을 삭제

npm update

- npm update {패키지명}
- {패키지명}을 가진 모듈을 업데이트

npm {script명}

- npm {script명}
- package.json 내에 있는 스크립트를 실행

```
$ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help init` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (node_orm)
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to D:\BSA\Node\Node_ORM\package.json:

{
  "name": "node_orm",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"

Is this OK? (yes)

swanb@DESKTOP-T5B94PA MINGW64 /d/BSA/Node/Node_ORM
```

실습 프로젝트 생성

[1] mynode 폴더 생성

[2] mynode 폴더에서 터미널을 열고 해당 터미널에서 아래 명령어 입력

npm init

package name등 물어보면 디폴트값으로 엔터를 입력한다

package.json 파일

package.json파일이 생성된 것 확인

Node.js 의 주요 특징

가볍고 효율적: Node.js는 사전 구축된 기능 라이브러리를 포함하지 않고, 추가 기능을 위한 광범위한 오픈소스 모듈 생태계를 수용하기 때문에 가볍다.

또한, 효율성과 자바스크립트 최적화로 유명한 크롬 V8 엔진을 기반으로 하여 Node.js의 민첩성을 더욱 향상시킨다

✅ **브라우저별 자바스크립트 엔진 정리**

브라우저	자바스크립트 엔진
Chrome	V8 (Google)
Edge (구형: EdgeHTML 기반)	Chakra (Microsoft)
Edge (신형: Chromium 기반)	✅ V8 사용
Firefox	SpiderMonkey (Mozilla)
Safari	JavaScriptCore (Apple, 일명 Nitro)
Opera	V8 (Chromium 기반)

참고로 노드는 브라우저와 상관 없이 v8엔진만 따로 떼어와서 자바스크립트를 실행하는 런타임 환경이다. 브라우저 없이도 실행 가능

Node 의 장단점

- 개발관점에서는빠르고 쉬운 장점을 갖는다
- 운영관점에서는 테스트, 장애대응, 디버깅 등에 신경써야할 부분이 많다

Node.js는 빠른 V8 엔진과 비동기 처리로 높은 성능을 제공하며, JS 하나로 프런트와 백을 모두 개발할 수 있는 장점이 있지만, CPU 집약적 작업이나 구조 복잡성에는 주의가 필요하다

Node 내장 객체

분류	내장 객체	설명
전역 객체	global	모든 모듈에서 접근 가능한 전역 객체 (window와 유사)
전역 함수	setTimeout, setInterval, setImmediate	비동기 타이머 함수
전역 함수	clearTimeout, clearInterval, clearImmediate	타이머 해제 함수
전역 함수	require()	외부 모듈을 불러오는 함수
전역 변수	__dirname	현재 모듈의 디렉토리 경로
전역 변수	__filename	현재 모듈의 파일 경로
모듈 객체	module	현재 모듈 정보를 담고 있는 객체
모듈 객체	exports	모듈에서 외부로 공개할 객체
콘솔	console	로그 출력용 객체 (e.g., console.log)
버퍼	Buffer	바이너리 데이터를 처리하는 객체
프로세스	process	현재 실행 중인 프로세스에 대한 정보 제공

Node 내장 객체

이벤트	EventEmitter	이벤트 기반 프로그래밍 지원 객체
타이머	setTimeout, clearTimeout 등	비동기 타이머 제어
경로	path	파일 및 디렉토리 경로 처리 유틸
파일 시스템	fs	파일 읽기/쓰기 등 파일 시스템 제어
운영체제	os	시스템 정보 및 자원 관련 함수
URL	URL, url	URL 처리 및 파싱 지원 객체
HTTP	http, https	서버 및 클라이언트 구현 객체
Net	net	TCP/IPC 서버 및 클라이언트 구현 객체
Zlib	zlib	데이터 압축/해제 지원 객체

Node 내장 객체

- 전역에서 접근할 수 있는 객체
- Node.js의 런타임 환경 내에서 전역적으로 사용할 수 있는 다양한 속성과 함수들을 제공한다.
- 브라우저 환경의 window 객체와 유사하지만, Node.js의 환경에 맞게 구성
- (브라우저가 아니므로 window객체 없음. 대신 global객체로 이용)
- 이 객체에 직접 접근하거나 자신만의 전역 변수를 추가하는 것이 가능

global 객체

- node는 브라우저가 아니므로 window객체가 없음
- 대신 global 객체를 이용한다

```
global.val = 'Global Variable';
global.func = () => {
    console.log('Global Function');
};

console.log(val);
func();
```

현재 실행 중인 파일명: D:\BSA\Node\Node_server_\ex01_global.js
현재 실행 중인 파일의 상위 경로: D:\BSA\Node\Node_server_

전역 변수

변수명	설명
__filename	현재 실행중인 코드의 파일 경로
__dirname	현재 실행중인 코드의 폴더 경로

```
//노드의 전역변수: __filename, __dirname
console.log('현재 실행 중인 파일명: %s', __filename);
console.log('현재 실행 중인 파일의 상위 경로: %s', __dirname);
```

```
현재 실행 중인 파일명: D:\BSA\Node\Node_server_\ex01_global.js
현재 실행 중인 파일의 상위 경로: D:\BSA\Node\Node_server_
```

전역 객체

객체명	설명
console	콘솔 화면 관련 기능 수행
process	프로그램과 프로세스 관련 기능 수행
exports	모듈과 관련된 기능 수행 (CommonJS)

Console 객체

```
> console.error("Error");
▶Error
undefined
> console.warn("Warning");
▶Warning
undefined
> console.debug("Debug");
undefined
> console.log("Log");
Log
undefined
> console.info("Information");
Information
undefined
> console.assert(1 == 2);
▶Assertion failed: console.assert
undefined
```

메서드명	설명
error	에러 출력 (x) 표시
warn	경고 출력 (!) 표시
debug	디버깅 관련 정보
log	로그 출력
info	정보 출력
assert	검증(Assertion)
time(tag)	시간 측정 시작
timeEnd(label)	시간 측정 종료

Console의 특수 문자

특수문자	설명
%d	숫자
%s	문자열
%j	JSON
%%	% 문자 자체

- util.format 함수에서도 동일한 특수문자 사용 가능

Process 객체의 속성

프로그램과 프로세스 관련 기능을 수행

속성명	설명
argv	실행 매개변수
env	실행 환경 관련 정보
version	Node의 버전
versions	종속된 프로그램 버전
arch	프로세서의 아키텍쳐 표시
platform	플랫폼 정보 표시

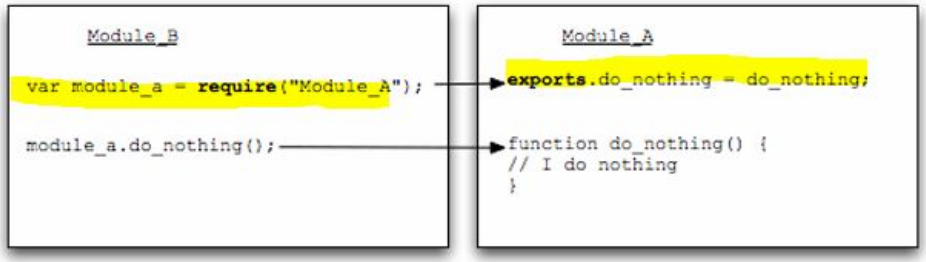
Process 객체 메서드

메서드명	설명
exit	프로그램 종료
memoryUsage	메모리 사용 정보 객체 반환
uptime	현재 프로그램이 실행된 시간

Exports 객체 및 모듈 사용법

모듈이란

모듈은 특정한 기능을 하는 함수나 변수들의 집합이다.
코드의 길이를 줄이고, 유지보수를 용이하게 할 수 있다는 장점이 있다.
이러한 모듈을 export하는 두 가지 방법인 module.exports와 exports를 살펴보자



Exports

Exports 방법 1

```
/**모듈화해서 내보내는 방법
 * 1. exports전역객체 이용: exports.프로퍼티
   2. module.exports이용 :module.exports=프로퍼티
   ----------------------------------
   exports  : 값 자체를 할당하는 것이 아닌 외부로 보낼 요소를
              exports 객체의 프로퍼티 또는 메서드로 추가한다
   module.exports : 객체에 하나의 값(기본자료형,함수,객체)만
              할당할 수 있다.
    ----------------------------------
 */
exports.num = 100;
exports.plus = function (a, b) {
    console.log(`%d + %d = %d`, a, b, a + b);
};
exports.minus = (a, b) => {
    console.log(`%d - %d = %d`, a, b, a - b);
};
```

```
// var obj = require('./ex05_module.js');
const obj = require('./ex05_module');
const obj2 = require('./sample');
/**
 * require('모듈파일명')
 * - 이 때 확장자 .js는 생략해도 된다
 * [1] require()하면 먼저 module1.js를 찾는다.
 * [2] 해당 파일이 없으면 module1이라는 디렉토리를 찾는다.
 * [3] 디렉토리가 있으면 해당 디렉토리의 index.js를 찾는다.
 */
console.log(obj.num);
obj.plus(910, 21);
obj.minus(55, 32);
```

Exports

Exports 방법 2

```
var area = {
    square: function(length) {
        return length * length;
    },
    circle: function(radius) {
        return radius * radius * Math.PI;
    },
    rectangle: function(width, height) {
        return width * height;
    }
}

module.exports = area;
```

```
var area = require('./modules/area_module');

console.log('area_module.square : %d', area.square(6));
console.log('area_module.circle : %d', area.circle(8));
console.log('area_module.rectanble : %d', area.rectangle(5,6));
```

module.export 와 exports의 차이

단순한 코드로 exports와 module.exports의 차이를 설명하자면 다음과 같다.

```
const module = { exports: {} };
const exports = module.exports;
// your code
return module.exports;
```

exports객체와 module.exports객체는 동일하며, exports 가 module.exports객체를 call by reference(함수에서 값을 전달하는 대신 주소값을 전달하는) 방식으로 바라보고 있으며, 최종적으로 리턴값은 module.exports 라는것이다.

위의 예제중에서 exports는 property 방식을 쓰고 module.exports는 그냥 바로 썼는데 그 이유는 exports를 바로 써버리면 module.exports의 call by reference 관계를 끊어버려서 exports라는 변수가 되버리기 때문이다.

> ✅ 네, `exports.함수명 = () => {}` 는 **정상적으로 작동합니다.**

하지만

> ❌ `exports = () => {}` 는 **작동하지 않습니다.**

```
const test = {
    name: "Kim",
    text: "Hi"
}

module.exports.person = test; // (o)
module.exports = test; // (o)

exports.person = test; // (o)
exports = test; // (x)
```

https://velog.io/@weed7968/Node.JS-module.export-%EC%99%80-exports%EC%9D%98-%EC%B0%A8%EC%9D%B4

require와 import 차이

항목	`require()` (CommonJS)	`import` (ES Modules)
사용 방식	전통적인 Node.js 방식	최신 표준 JavaScript 모듈 방식
표준 여부	Node.js 전용 (비표준)	**ECMAScript 공식 표준**
동기/비동기	동기적 로딩	비동기적 로딩 가능
파일 확장자	`.js` (기본)	`.js`, `.mjs`, 또는 `type: module` 필요
트리 쉐이킹	❌ 불가	✅ 가능
최근 추세	점차 감소	✅ **점점 널리 사용됨**

ES6 모듈 사용 방법

Node.js에서 ES6 모듈을 사용하려면,
- 모듈 파일의 확장자를 .mjs로 변경하거나,
- package.json 파일에서 "type": "module" 설정을 추가해야 한다.

[1] 확장자를 .mjs로 변경: 모듈 파일을 .mjs 확장자로 저장하면 Node.js가 이를 ES6 모듈로 인식한다.
[2] package.json 설정 추가: 프로젝트의 package.json 파일에 "type": "module"을 추가하면 .js 확장자를 가진 파일들도 ES6 모듈로 인식된다.

```
* ---package.json--------
{
  "type": "module"
}
--------------------------
```

ES6 모듈 사용 방법

ex08_module.mjs > ...

```
1  // print.mjs
2  export function printStar(num) {
3      for (let i = 0; i < num; i++) {
4          console.log('★★★★★');
5      }
6  }
```

ex08_import.mjs

```
1  import { printStar } from './ex08_module.mjs';
2  printStar(10);
3
```

OS 모듈

운영체제와 정보를 담고있는 모듈

메서드명	설명
hostname	운영체제의 호스트명을 반환
type	운영체제의 이름을 반환
platform	운영체제의 플랫폼을 반환
arch	운영체제 아키텍처 반환
release	운영체제의 버전을 반환
uptime	운영체제 실행된 시간을 반환
loadavg	로드 평균값 정보 반환 (Array)
totalmem	운영체제 총 메모리 사이즈 반환
freemem	시스템 사용 가능한 메모리 반환
cpus	CPU 정보 반환 (Array)
getNerworkInterfaces	네트워크 인터페이스 정보 반환 (Array)

path 모듈

폴더와 파일의 경로를 쉽게 조작하도록 도와주는 모듈

메서드/속성	설명	예시
path.basename(path)	경로에서 파일명 추출	basename('/a/b/c.txt') → 'c.txt'
path.basename(path, ext)	확장자 제외한 파일명	basename('/a/b/c.txt', '.txt') → 'c'
path.dirname(path)	디렉토리 경로만 추출	dirname('/a/b/c.txt') → '/a/b'
path.extname(path)	확장자만 추출	extname('file.tar.gz') → '.gz'
path.join(...paths)	경로 조각을 연결	join('/a', 'b', 'c.txt') → '/a/b/c.txt'
path.resolve(...paths)	절대 경로 반환	resolve('a', 'b') → '/현재경로/a/b'
path.normalize(path)	구문 오류 수정	normalize('/a/b/../c') → '/a/c'
path.isAbsolute(path)	절대경로 여부 확인	isAbsolute('/a/b') → true
path.relative(from, to)	상대 경로 계산	relative('/a/b', '/a/c/d') → '../c/d'
path.parse(path)	경로를 객체로 분해	parse('/a/b/c.txt') → { root, dir, base, name, ext }
path.format(obj)	객체를 경로 문자열로	format({ dir: '/a/b', base: 'c.txt' }) → '/a/b/c.txt'
path.sep	경로 구분자 ('/' or '\')	Unix: '/', Windows: '\'
path.delimiter	환경변수 구분자 (':' or ';')	Unix: ':', Windows: ';'

url 모듈

인터넷 주소를 쉽게 조작하도록 도와주는 모듈

url 처리에는 크게 2가지 방식이 있다.

=>[1] WHATWG방식과 [2] 예전부터 사용하던 방식의 url

구분	설명	예시
URL 클래스	WHATWG 방식의 URL 파싱	new URL('https://example.com/path')
url.parse()	구 Node.js 방식 (Legacy)	url.parse('https://...')
url.format()	URL 객체 → 문자열	url.format(obj)
url.resolve()	base URL + 상대 경로 → 전체 URL	url.resolve('https://a.com', '/b')

url 모듈

new URL() - 권장 방식(WHATWG 표준)

```
const { URL } = require('url');

const myURL = new URL('https://example.com:8080/p/a/t/h?query=1#hash');

console.log(myURL.hostname);  // 'example.com'
console.log(myURL.port);      // '8080'
console.log(myURL.pathname);  // '/p/a/t/h'
console.log(myURL.search);    // '?query=1'
console.log(myURL.hash);      // '#hash'
console.log(myURL.href);      // 전체 URL
```

옛날 방식 (호환성 위해 존재)

```
const url = require('url');

const parsed = url.parse('https://example.com:8080/path?query=123#abc');

console.log(parsed.hostname);  // 'example.com'
console.log(parsed.port);      // '8080'
console.log(parsed.pathname);  // '/path'
console.log(parsed.query);     // 'query=123'
console.log(parsed.hash);      // '#abc'
```

URL의 searchParams

URL 객체의 searchParams는 URL 쿼리 문자열을 쉽게 **추가**, **수정**, **삭제**, **읽기** 할 수 있도록 도와주는 인터페이스

```
// URL 생성
const myUrl2 = new URL('https://example.com/product?category=books&id=123');

// 쿼리 읽기
console.log(myUrl2.searchParams.get('category')); // 👉 'books'
console.log(myUrl2.searchParams.get('id'));       // 👉 '123'

// 쿼리 추가
myUrl2.searchParams.append('sort', 'price');
console.log(myUrl2.href); // 👉 https://example.com/product?category=books&id=123&sort=price

// 쿼리 수정
myUrl2.searchParams.set('id', '999');
console.log(myUrl2.href); // 👉 https://example.com/product?category=books&id=999&sort=price

// 쿼리 삭제
myUrl2.searchParams.delete('category');
console.log(myUrl2.href); // 👉 https://example.com/product?id=999&sort=price

// 반복 (모든 key=value 출력)
for (const [key, value] of myUrl2.searchParams) {
  console.log(`${key} = ${value}`);
}
// 👉 id = 999
// 👉 sort = price
```

querystring 모듈

WHATWG방식 이전의 URL을 사용할 때 쿼리스트링을 쉽게 추출하기 위한 모듈

```
1  //url모듈 학습후 querystring을 살펴보자
2  const url = require('url'); //교재 챕터3장
3  const querystring = require('querystring');
4  //WHATWG방식 이전의 URL을 사용할 때 쿼리스트링을 쉽게 추출하기 위한 모듈
5
6  const str = `https://shopping.naver.com/luxury/cosmetic/category?optionFilters=CH_101923673&page=1&per_page=10#hash`;
7
8  const parsedUrl = url.parse(str);
9  const qStr = parsedUrl.query; //optionFilters=CH_101923673&page=1&per_page=10
10 console.log(qStr);
11 const query = querystring.parse(qStr);
12 console.log(query); //객체로 반환. 객체의 속성을 이용해  파라미터값을 추출할 수 있다
13 console.log('page번호: ', query.page);
14 console.log('한 페이지 목록 개수: ', query.per_page);
15 console.log('옵션 필터: ', query.optionFilters);
16
17 console.log('querystring.stringify():', querystring.stringify(query));
18 //분해된 쿼리 객체를 문자열로 직렬화
19
```

fs 모듈 -File System Module

파일 시스템에 접근하는 모듈.

메서드명	설명
readFile(file, encoding, callback)	비동기적 파일 읽기
readFileSync(file, encoding)	동기적 파일 읽기
writeFile(file, encoding, callback)	비동기적 파일 쓰기
writeFileSync(file, encoding)	동기적 파일 쓰기
appendFile(file, encoding, callback)	비동기적 파일 추가
appendFileSync(file, encoding)	동기적 파일 쓰기

fs 모듈

```
1  //내장 모듈 파일 시스템 모듈 사용해보기
2
3  const fs = require('fs');
4  //파일을 읽고 쓰기 할 때 사용
5  //1. 동기방식으로 파일을 읽을 경우
6  var data = fs.readFileSync('readme.txt', 'utf-8');
7  console.log(data);
8  console.log('Bye Bye~~');
9
10 //2. 비동기방식으로 파일을 읽는 경우
11 fs.readFile('readme2.txt', 'utf-8', function (err, data) {
12     if (err) throw err; //console.error(err);
13     console.log(data);
14 });
15 console.log('잘가~~~');
16
```

스트림을 이용한 파일 읽고 쓰기

파일을 읽거나 쓰는 방식에는 두 가지 방식이 있다.

[1] 버퍼를 이용하는 방식 (파일 데이터를 버퍼에 모아 읽거나 씀)

: 파일을 읽을 때 메모리에 파일 크기만큼 공간을 마련해 파일 데이터를 메모리에 저장한 뒤 사용자가 조작할 수 있도록 한다

[2] 스트림을 이용하는 방식 (데이터를 조금씩 전송하는 방식)

: 데이터를 조금씩 나눠서 읽거나 쓰며 나눠진 조각을 chunk라고 한다

fs.readFile() 은 파일 전체를 한 번에 메모리에 로드하나 fs.createReadStream()의 경우 파일을 조각(chunk)단위로 스트리밍 한다.
fs.readFile() 의 경우 대용량 파일 처리시 비효율적 (메모리 부족 위험), 콜백 방식
스트림 방식은 효율적 점진적 읽기 가능하며 이벤트 기반으로 처리함 (큰 파일에서 유리함) 큰 로그파일/동영상/오디오 등에 사용

스트림을 이용한 파일 읽고 쓰기

```
1  // 스트림 기반 파일 입력받기
2  const fs = require("fs");
3
4  console.log("Start");
5
6  const file = "./fs/test.pdf";
7
8  const readStream = fs.createReadStream(file, { highWaterMark: 16 });
9  // 이벤트방식(비동기)의 입력
10 const data = [];
11 readStream.on("data", (chunk) => {
12     // console.log(chunk);
13     // console.log(chunk.length);
14     data.push(chunk);
15 });
16
17 readStream.on("end", () =>{
18     const buffer = Buffer.concat(data);
19     console.log(buffer.toString());
20 });
21
22 readStream.on("error", (error) => {
23     console.log(error.message);
24 });
25
26 console.log("End");
```

이벤트

Event 연결 메서드

메서드명	설명
addEventListener	이벤트 연결
on	이벤트 연결 (추천)

이벤트 제거 메서드

메서드명	설명
removeListener(이벤트명, 핸들러)	특정 이벤트의 리스너 제거
removeAllListeners([이벤트명])	모든 이벤트의 리스너 제거

이벤트

이벤트 강제 발생

메서드명	설명
emit(이벤트명[, args ...])	이벤트를 실행함

Custom Event

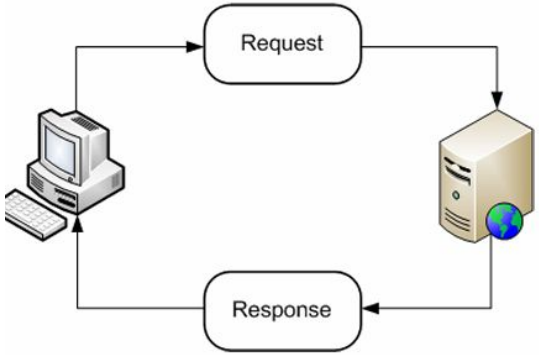
```
const customEvent = require('events');
```

메서드명	설명
addEventListener(이벤트명, 핸들러)	이벤트 연결
on(이벤트명, 핸들러)	이벤트 연결 (추천)
setMasListeners(개수)	이벤트 연결 개수를 조절
removeListener(이벤트명, 핸들러)	특정 이벤트의 리스너를 제거
removeAllListeners([이벤트명])	모든 이벤트 리스너를 제거
once(이벤트명, 이벤트 핸들러)	1회성 이벤트 연결

http 모듈

웹 서버를 만들거나 클라이언트로 HTTP요청을 보낼 수 있게 해준다.

Express 없이도 웹서버를 직접 구현할 수 있는 저수준 API를 제공한다



`http` 모듈 주요 기능

기능	설명
HTTP 서버 생성	클라이언트 요청 수신 및 응답
HTTP 클라이언트 요청 전송	외부 서버에 요청 보내기
요청(request)/응답(response) 처리	헤더, 본문 등 조작 가능

http 모듈

메서드명	설명
listen(port[, callback])	서버를 실행함
close()	서버를 종료함

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

var server = http.createServer();

server.listen(3000);
```

http 모듈

```js
const http = require('http');

const server = http.createServer((req, res) => {
    // 요청 정보 확인
    console.log(req.method, req.url);

    // 응답 작성
    res.statusCode = 200;
    res.setHeader('Content-Type', 'text/html; charset=utf-8');
    res.end('<h1 style="text-align:center;color:red">Hello, Node.js HTTP Server!</h1>');
});

server.listen(3000, () => {
    console.log('서버가 http://localhost:3000 에서 실행 중입니다');
});
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

swanb@DESKTOP-T5B94PA MINGW64 /d/BSA/Node/Node_server_
$ node ex12_1http.js
서버가 http://localhost:3000 에서 실행 중입니다
GET /
GET /favicon.ico
```

Hello, Node.js HTTP Server!

요청 객체 (req)

클라이언트의 요청 정보를 담고 있으며, 다양한 속성과 메서드를 제공한다.

📦 요청 객체(req) 속성 및 메서드 정리

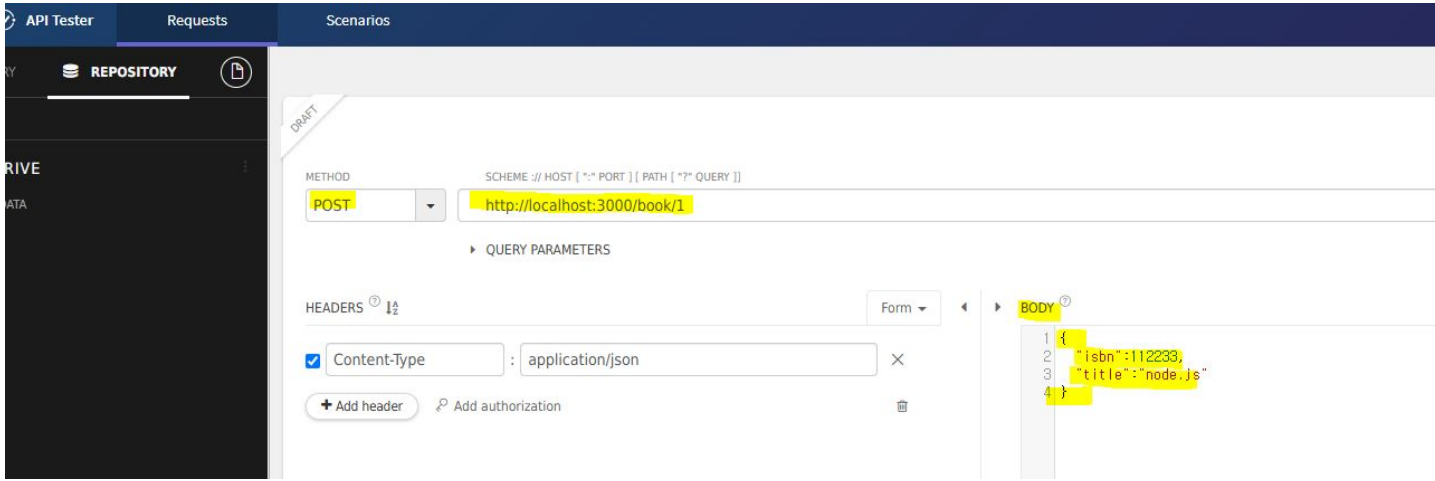
구분	속성/메서드	설명
기본 정보	`req.method`	요청 메서드 (예: `'GET'`, `'POST'`, `'PUT'`, `'DELETE'`)
	`req.url`	요청 URL 경로 (`/user?id=1` 등)
	`req.headers`	요청 헤더 객체 (`{ host: ..., connection: ..., ... }`)
	`req.httpVersion`	HTTP 버전 (예: `'1.1'`)
본문 처리	`req.on('data')`	요청 본문 데이터를 받을 때 사용 (스트림 방식)
	`req.on('end')`	요청 본문 데이터 수신이 끝났을 때 실행
기타	`req.socket`	연결된 소켓 객체 (IP 정보 등 확인 가능)
	`req.setEncoding()`	본문 데이터의 인코딩 설정 (예: `'utf8'`)

요청 객체 (req)

```
1  const http = require('http');
2
3  const server = http.createServer((req, res) => {
4      console.log('메서드:', req.method); // ex: GET
5      console.log('URL:', req.url); // ex: /users
6      console.log('헤더:', req.headers); // ex: { host: ..., connection: ... }
7
8      let body = '';
9      req.setEncoding('utf8');
10
11     req.on('data', (chunk) => {
12         body += chunk;
13     });
14
15     req.on('end', () => {
16         console.log('본문 데이터:', body);
17         res.writeHead(200, { 'Content-Type': 'text/plain; charset=utf-8' });
18         res.end('요청 수신 완료');
19     });
20 });
21
22 server.listen(3000, () => {
23     console.log('서버 실행 중: http://localhost:3000');
24 });
25
```

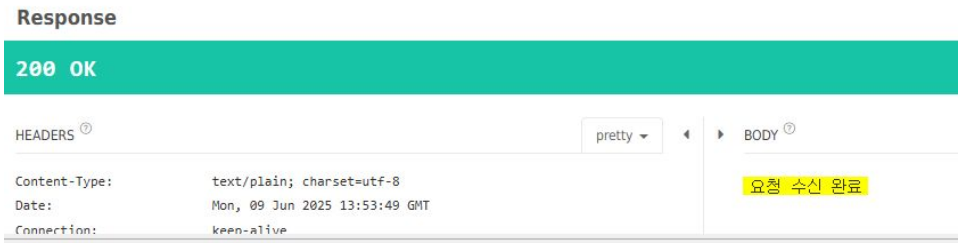
요청 객체 (req)

Talend API Tester를 chrome webstore에서 설치한 뒤 테스트



요청 객체 (req)

테스트 결과




```
메서드: POST
URL: /book/1
헤더: {
  host: 'localhost:3000',
  connection: 'keep-alive',
  'content-length': '40',
  'sec-ch-ua-platform': '"Windows"',
  '___internal-request-id': '3abbf494-570a-4f87-ac48-7e265b231d12',
  'sec-ch-ua': '"Google Chrome";v="137", "Chromium";v="137", "Not/A)Brand";v="24"',
  'content-type': 'application/json',
  'sec-ch-ua-mobile': '?0',
  accept: '*/*',
  'sec-fetch-site': 'none',
  'sec-fetch-mode': 'cors',
  'sec-fetch-dest': 'empty',
  'sec-fetch-storage-access': 'active',
  'sec-fetch-dest': 'empty',
  'sec-fetch-storage-access': 'active',
  'sec-fetch-storage-access': 'active',
  'accept-encoding': 'gzip, deflate, br, zstd',
  'accept-encoding': 'gzip, deflate, br, zstd',
  'accept-language': 'ko-KR,ko;q=0.9,en-US;q=0.8,en;q=0.7',
  cookie: 'Idea-85dd8faf=4b7c1b30-fcbc-4352-bdc7-d328c0ed2c69'
}
본문 데이터: {
  "isbn":112233,
  "title":"node.js"
}
```

응답 객체 (res)

`http` 모듈에서 **서버가 클라이언트에 응답할 때 사용하는 객체**.

`ServerResponse` 클래스의 인스턴스이며, **쓰기(writeable) 스트림** 이다.

1. 간단한 HTML 응답

```js
const http = require('http');

http.createServer((req, res) => {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/html');
  res.write('<h1>Hello World</h1>');
  res.end(); // 응답 종료
}).listen(3000);
```

2. JSON 응답

```js
res.setHeader('Content-Type', 'application/json');
res.end(JSON.stringify({ message: '성공' }));
```

`res`는 **Writable Stream**이기 때문에 `write()`, `end()` 등의 스트림 메서드를 사용할 수 있다.

`res.end()`를 반드시 호출해야 응답이 종료되고 클라이언트는 응답을 받을 수 있다.

Express에서는 `res.send()`, `res.json()`, `res.redirect()` 등으로 더 간편하게 사용할 수 있습니다.

✅ `res` 객체 주요 속성과 메서드 정리

분류	메서드 / 속성	설명
상태 코드	`res.statusCode`	응답 상태 코드 설정 (`res.statusCode = 404`)
응답 헤더	`res.setHeader(name, value)`	응답 헤더 설정
	`res.getHeader(name)`	설정된 응답 헤더 가져오기
	`res.removeHeader(name)`	응답 헤더 제거
본문 작성	`res.write(chunk[, encoding])`	응답 본문 데이터를 스트림처럼 작성
	`res.end([data][, encoding])`	응답 종료 및 본문 전송 완료
	`res.writeHead(statusCode[, headers])`	상태 코드 및 헤더를 한번에 설정
리다이렉트	(직접 구현)	`res.writeHead(302, { Location: '/new-url' }); res.end();`
MIME 설정	(직접 구현)	`res.setHeader('Content-Type', 'text/html')` 등으로 설정
쿠키	(직접 구현 또는 라이브러리)	`res.setHeader('Set-Cookie', 'name=value')`

express 모듈

express란?`

- http 모듈과 fs모듈을 이용해서 서버를 일일이 구현할 수도 있지만 서버 규모가 커지면 작성해야 할 공통 코드가 많아진다
- express 프레임워크를 사용하면 이러한 문제를 해결할 수 있다
- 웹 프레임워크로 request와 response를 완전하게 통제할 수 있고 웹서비스와 웹 애플리케이션 개발을 수월하게 할 수 있도록 다양한 API 를 제공한다

express 와 nodemon 모듈 설치

package.json에
등록된 모듈 확인

express모듈 설치

```
npm i express
```

nodemon 설치

```
npm i -D nodemon
```

node로 js파일을 실행하면 코드가 변경될 때마다 재 실행해야 함.
nodemon을 사용하면 코드 변경사항 생길 때마다 자동으로 이를 감지하여 코드를 재실행해준다. 외부 모듈이므로 설치가 필요하며 -D옵션을 주면 dependencies가 아닌 devDependencies에 모듈이 추가된다. 개발용으로만 사용하겠단 것

package.json에 script 부분 기술

[참고] package.json에 의존성 라이브러리 에서 버전이 표기되는 의미를 살펴보면 SemVer표기법을 사용하는데 1.0.1 과 같이 3자리 수로 표기한다.

첫 번째 자리: 중요한 버전 업데이트
두 번째 자리: 버그 수정
세 번째 자리: 작은 수정사항 등을 반영한 업데이트를 의미

^0.0.0 : 첫 번째 자리 버전까지 고정
~0.0.0 : 두 번째 자리 버전까지 고정
0.0.0 : 모든 버전을 고정

package.json파일의 scripts 부분에 “start”속성에 “nodemon app”을 기술해주자

npm start 명령어로 서버 실행

그런 뒤 터미널에서 npm start 명령어 입력후 엔터를 치면 아래와 같이 nodemon app 이 자동 실행되면서 서버가 기동되는 것을 확인할 수 있다.

[참고]

node 파일명 대신
npx nodemon 파일명
명령어를 사용해도 된다.

우리는
package.json파일에
'script' 옵션을 주어
npm start로 실행시키는
방법을 이용한다

express 사용법

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

app.get("/", (req, res) => {
    res.send("hello Javascript")
});

app.listen(3000, ()=>{
    console.log("3000번 포트로 서버가 열렸습니다!")
});
```

express의 주요 메서드

메서드명	설명
set(key, value)	서버 설정을 위한 속성 지정
get(key)	서버 설정 속성을 불러옴
use([path,] function [, function ...])	미들웨어 함수 사용
get(path, callback)	GET 메서드의 요청을 처리
post(path, callback)	POST 메서드의 요청을 처리
all(path, callback)	모든 요청을 처리

```
const app = express();
//미들웨어 설정
app.use(express.json());
app.use(express.urlencoded({ extended: true }));
app.use(express.static(path.join(__dirname, 'public')));
```

express의 주요 속성

속성명	타입	역할
env	속성	서버 환경 설정시 사용. 실행 환경 이름 (예: development, production)
views	속성	템플릿(뷰) 파일의 디렉토리 경로 지정.
view engine	속성	기본 뷰 엔진을 설정.

```
app.set('views', path.join(__dirname, 'views')); // views 폴더 지정
app.set('view engine', 'ejs'); // ejs 템플릿 엔진 설정
```

Express 요청 함수

요청 메서드	설명	예시	비고
app.get()	GET 요청 처리	app.get('/user', handler)	데이터 조회
app.post()	POST 요청 처리	app.post('/user', handler)	데이터 생성
app.put()	PUT 요청 처리	app.put('/user/:id', handler)	전체 수정
app.patch()	PATCH 요청 처리	app.patch('/user/:id', handler)	부분 수정
app.delete()	DELETE 요청 처리	app.delete('/user/:id', handler)	데이터 삭제
app.all()	모든 HTTP 메서드 처리	app.all('/admin', handler)	모든 요청 허용
app.use()	미들웨어 또는 하위 라우터 등록	app.use('/api', router)	공통 처리
app.route()	라우팅 체이닝 처리	app.route('/user').get(...).post(...)	구조화된 라우팅

Express 응답 함수

응답 메서드	설명	예시	비고
res.send()	문자열, 객체, 배열 등을 응답	res.send('Hello')	자동 Content-Type 설정
res.json()	JSON 형식으로 응답	res.json({name:'Tom'})	API 응답 시 자주 사용
res.status()	HTTP 상태 코드 설정	res.status(404).send('Not Found')	체이닝 가능
res.redirect()	URL로 리디렉션	res.redirect('/login')	302 기본, 상태 코드 설정 가능
res.render()	뷰 템플릿 렌더링	res.render('index', {title: 'Home'})	view 엔진 필요
res.sendFile()	파일 전송	res.sendFile(__dirname + '/index.html')	정적 파일 응답
res.set()	응답 헤더 설정	res.set('Content-Type', 'text/plain')	커스텀 헤더 지원
res.cookie()	쿠키 설정	res.cookie('token', 'abc123')	옵션으로 secure, maxAge 등 설정
res.end()	응답 종료	res.end()	본문 없이 종료 가능

express와 미들웨어

Middleware란?

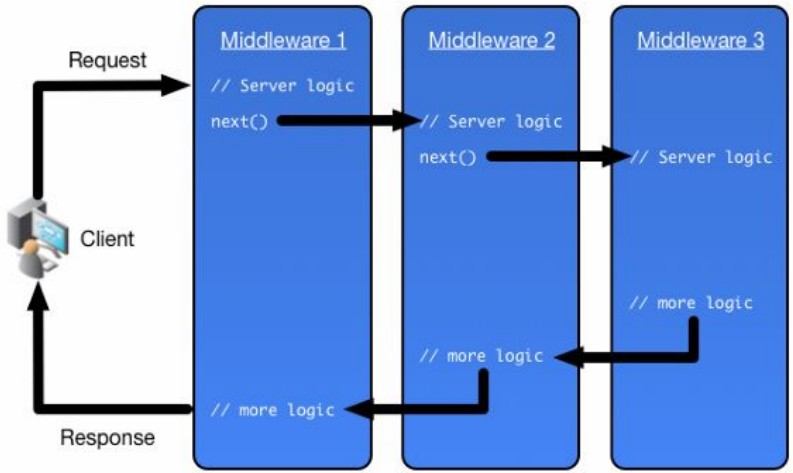
요청(request)과 응답(response) 사이에서 실행되는 중간 처리 함수

Express 앱에서는 클라이언트가 요청하면,

1. 요청(req)을 받고,
2. 미들웨어에서 가공/검사/로깅 등의 작업을 수행한 후,
3. 다음 미들웨어 또는 최종 라우터로 전달하거나,
4. 응답(res)을 보낼 수 있다.

Middleware

```
app.use((request, response, next) => {
    //  Do Something
    next();
});
```



미들웨어 종류

종류	설명	예시
애플리케이션 레벨	전체 라우터에 적용되는 미들웨어	`app.use(...)`
라우터 레벨	특정 라우터에만 적용	`router.use(...)`
에러 처리용	에러를 처리하는 미들웨어 (`next(err)` 호출 시)	`function(err, req, res, next)`
기본 제공 미들웨어	Express 내장 기능	`express.json()` , `express.urlencoded()`
서드파티 미들웨어	외부 라이브러리	`morgan` , `cors` , `helmet` 등

글로벌 미들웨어: 모든 요청에 대해 동작하며, 일반적으로 라우터 전에 설정된다.
예를 들어, 로깅, 보안, 요청 본문 파싱(body-parser) 등이 여기에 해당함

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

// 모든 요청에 대해 동작하는 글로벌 미들웨어
app.use((req, res, next) => {
    console.log(`${req.method} ${req.url}`);
    next();
});
```

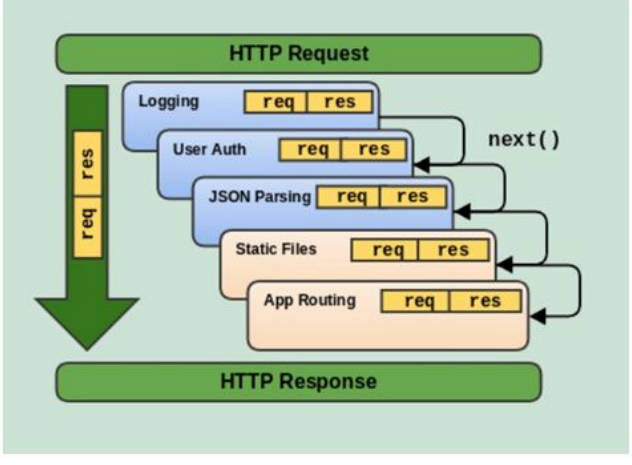
라우터 수준 미들웨어: 특정 라우트에 대해서만 동작하도록 설정된다. 이 경우, 라우터와 함께 정의된다.

```
const router = express.Router();

// 특정 라우터에만 적용되는 미들웨어
router.use((req, res, next) => {
    if (!req.headers['x-auth']) {
        return res.status(403).send('Forbidden');
    }
    next();
});

router.get('/profile', (req, res) => {
    res.send('Profile page');
});

app.use('/user', router);


//에러 처리 미들웨어
app.use((err, req, res, next) => {
    console.error(err.stack);
    res.status(500).send('Server Error');
});
```

동작 순서

- **전역 미들웨어 → 라우터 → 라우터 수준 미들웨어 → 응답 → 에러 처리 미들웨어**

Express Middleware



✅ 예시 1: 로깅 미들웨어

```js
app.use((req, res, next) => {
    console.log(`[${new Date()}] ${req.method} ${req.url}`);
    next(); // 다음 미들웨어로 이동
});
```

✅ 예시 2: JSON 파싱 미들웨어 (내장)

```js
app.use(express.json()); // 요청 body가 JSON이면 파싱
```

✅ 예시 3: 인증 미들웨어

```js
function auth(req, res, next) {
    if (!req.headers.authorization) {
        return res.status(401).json({ message: 'Unauthorized' });
    }
    next();
}

app.get('/protected', auth, (req, res) => {
    res.send('비밀 정보!');
});
```

미들웨어 배치 순서

Express에서 요청이 들어오면 **위에서 아래로 순차적으로 미들웨어가 실행된다**
따라서 **배치 순서가 중요**하다.

Express는 요청이 들어오면 `app.use()` 나 `app.get()`, `app.post()` 등에 등록된 미들웨어들을 **등록한 순서대로** 실행한다.
한 번 지나간 미들웨어는 다시 실행되지 않으며, `next()`를 호출해야 다음 미들웨어로 넘어갑니다.

배치순서를 잘못 두면 가령 라우터보다 인증 미들웨어가 뒤에 있으면 인증이 동작하지 않는 경우가 생길 수 있다.

코드 작성 순서에서 미들웨어와 라우터를 어떻게 배치하느냐에 따라 애플리케이션의 동작이 달라질 수 있다.
일반적으로 다음과 같은 규칙을 따르는 것이 좋다:

1. 전역 미들웨어는 라우터 전에 배치
로깅, 인증, 요청 본문 파싱 등 모든 요청에 적용해야 하는 미들웨어는 라우터 정의 전에 배치한다.
이렇게 하면 모든 요청이 라우터로 전달되기 전에 이 미들웨어를 거치게 된다.

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

// 전역 미들웨어
app.use(express.json()); // 요청 본문 파싱
app.use((req, res, next) => {
    console.log('Request received:', req.method, req.url);
    next();
});

// 라우터
app.get('/hello', (req, res) => {
    res.send('Hello, world!');
});
```

미들웨어 배치 순서

2. 특정 라우터에만 적용할 미들웨어는 해당 라우터와 함께 정의
특정 경로에만 적용할 미들웨어는 해당 라우터와 함께 정의한다.
이렇게 하면 특정 요청에 대해서만 미들웨어가 적용된다.

```
const router = express.Router();

// 특정 라우터에만 적용되는 미들웨어
router.use((req, res, next) => {
    if (!req.headers['x-auth']) {
        return res.status(403).send('Forbidden');
    }
    next();
});

router.get('/profile', (req, res) => {
    res.send('Profile page');
});

app.use('/user', router);
```

3. 에러 처리 미들웨어는 라우터와 다른 미들웨어 뒤에 배치
에러 처리 미들웨어는 요청 처리 중 발생한 에러를 잡아내야 하기 때문에, 가장 마지막에 배치해야 한다.
그렇지 않으면 에러가 발생했을 때, 에러 처리 미들웨어가 실행되지 않을 수 있다.

```
app.use((err, req, res, next) => {
    console.error('Error occurred:', err);
    res.status(500).send('Internal Server Error');
});
```

참고 사이트 및 도서

https://www.brilworks.com/blog/node-js-architecture-best-practices/

Node.js 교과서 - 조현영 (길벗)

백견 불여일타 Node.js로 서버 만들기 -박민경 저(로드북)