

# Node.js

Quick Guide

# Node.js

## : 간략한 소개

### ▶ Node.js

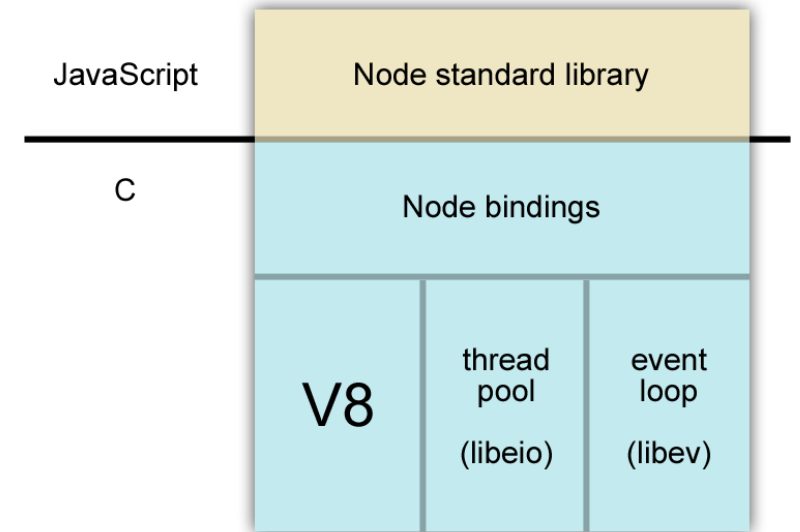
- ▶ 이벤트 주도(Event-Driven) 방식의 Server-Side JavaScript 환경
- ▶ 응답 속도가 빠르고 확장이 용이한 서버용 응용 프로그램 작성에 강점

### ▶ Node.js의 주요 특징

- ▶ 모듈 기반(필요한 것들을 추가, 조합하여 사용할 수 있다) : Node Package Management(NPM)로 패키지를 관리
- ▶ Non-Blocking (Event-Driven) I/O

### ▶ CommonJS와 V8 Runtime

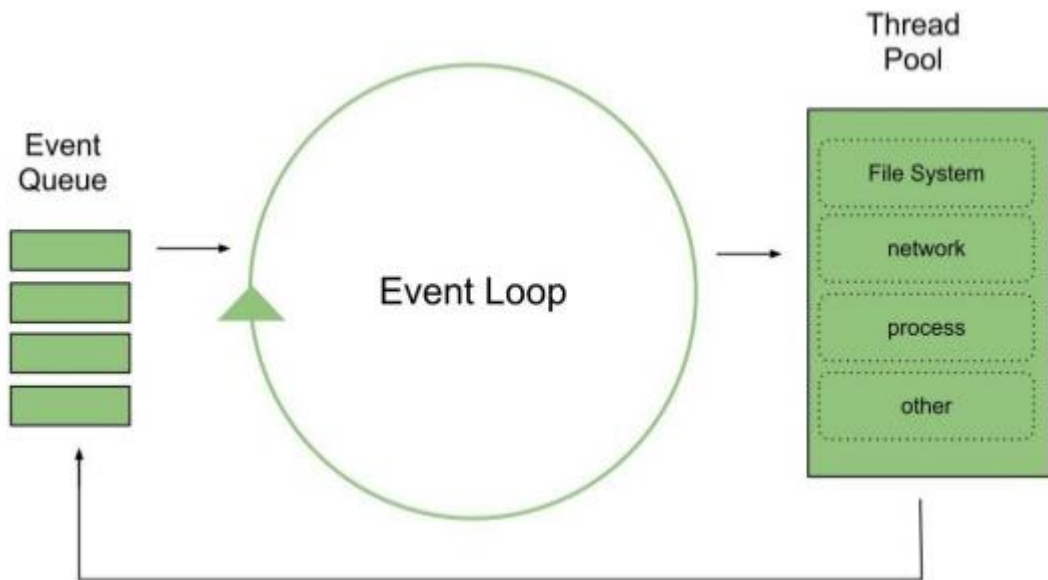
- ▶ 웹 브라우저를 벗어나 JavaScript를 사용하려는 다양한 시도들 중
  - ▶ 구글 크롬 브라우저에 탑재된 V8 Engine이 JavaScript 실행 속도를 획기적으로 개선 (2008)
  - ▶ ServerJS 프로젝트(2009) -> CommonJS 표준이 제정
- ▶ CommonJS 표준과 V8 Engine을 기반으로 Ryan Dahl이 Node.js 개발



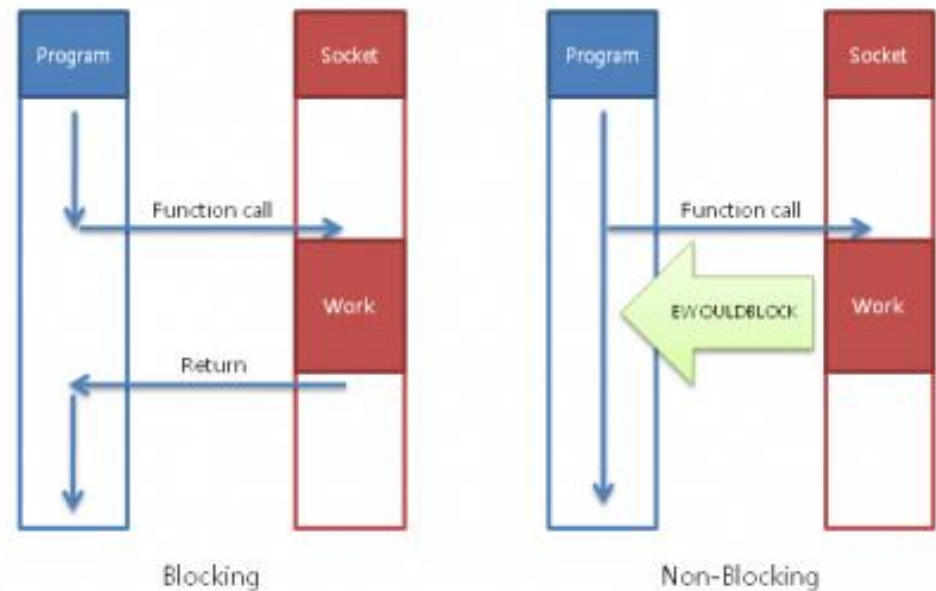
# Event-Driven Model과 Non-Blocking I/O

- ▶ **Event** : 한 쪽에서 다른 쪽으로 상태 혹은 실행 결과물을 전송하는 것
- ▶ **Non-Blocking** : 하나의 작업이 끝나기를 기다리지 않고 별도 **Thread**에 전달 후 작업을 속행, 해당 작업이 끝났음을 알리는 이벤트를 수신하면 추가 작업을 진행
- ▶ Event + Non-Blocking -> Node.js의 핵심

## Event-Driven



## Non-Blocking I/O



# Node.js의 Global 객체

## ▶ Global 객체 : 코드의 어느 부분에서나 사용할 수 있는 객체

### ▶ Global 변수

변수명	설명
<code>__filename</code>	현재 실행 중인 코드의 파일 경로
<code>__dirname</code>	현재 실행 중인 코드의 디렉터리 경로

### ▶ Global 객체

객체명	설명
<code>console</code>	콘솔 창에 결과를 출력하는 객체
<code>process</code>	프로세스의 실행에 대한 정보를 다루는 객체
<code>exports</code>	모듈을 다루는 객체

# process 객체

## : 속성과 메서드

### ▶ process 객체의 속성

속성명	설명
argv	실행 매개 변수
env	실행 환경 관련 정보
version	Node 버전
versions	종속된 프로그램의 버전 정보
arch	프로세서의 아키텍처 정보
platform	플랫폼 정보 표시

### ▶ process 객체의 메서드

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

# exports 객체

- ▶ JavaScript 파일 내부의 객체를 외부에서 접근할 수 있도록 "내보내는" 객체
- ▶ 모듈: 독립적인 실행영역을 가지는 개별 자바스크립트 파일
- ▶ 외부의 모듈을 불러오려면 **require** 함수로 모듈명을 명시하여 할당

module\_example.js

```
const add = require("./modules/test_module1").add;
const square = require("./modules/test_module1")
    .square;
const area = require("./modules/test_module2");

console.log(add(10, 20));
console.log(square(10));
console.log(area.circle(10));
```

모듈 내 개별 객체 내보내기

모듈 전체 내보내기

test\_module1.js

```
exports.add = function(num1, num2) {
    return num1, num2;
};

exports.square = function(length) {
    return length * length;
};
```

test\_module2.js

```
const area = {
    circle: function(radius) {
        return radius ** 2 * Math.PI;
    },
    // ...
}
module.exports = area;
```

# Event와 EventEmitter

- ▶ **Event** : 한 쪽에서 다른 쪽으로 상태 혹은 실행 결과물을 전송하는 것
- ▶ **Node.js**에는 **Event**를 주고 받기 위해 **EventEmitter**라는 것이 구현
  - ▶ 메시지를 전송할 때는 **emit()** 메서드를 이용
  - ▶ 메시지를 수신하려면 이벤트 리스너를 등록하여 사용

메서드	설명
<code>on(event, listener)</code>	지정한 이벤트의 리스너를 추가
<code>once(event, listener)</code>	지정한 이벤트의 리스너를 추가 한 번 호출된 이후에는 리스너 자동 제거
<code>removeListener(event, listener)</code>	지정한 이벤트 리스너에 대한 리스너 제거

- ▶ **process** 내장 객체는 기본적으로 **EventEmitter**를 탑재하고 있지만, 사용자 정의 객체에 **on()**과 **emit()**을 사용하기 위해서는 **EventEmitter**를 상속받을 수 있음
  - ▶ **Node.js**의 내장 모듈 **util**을 이용하면 편리하게 **prototype** 상속을 받을 수 있음

# Event와 EventEmitter

## : by Example

함수명	설명
setTimeout(callback, ms)	ms 시간이 지난 후 callback 함수 실행
setInterval(callback, ms)	ms 시간마다 callback 함수 실행
clearInterval(timer)	지정된 timer 해제

### general\_example.js

```
const Ticker = require("./modules/ticker");

process.on("tick", function() {
  seconds++;
  console.log(seconds + " second passed.");

  if (seconds > 10) {
    ticker.emit("stop");
  }
});

let ticker = new Ticker(process);
ticker.start();
```

이벤트 전송

### ticker.js

```
const util = require('util');
const EventEmitter =
  require("events").EventEmitter;
let tick_target = null;
const Ticker = function(target) {
  tick_target = target;

  this.on("stop", function() {
    clearInterval(ticker);
  });

  Ticker.prototype.start = function() {
    ticker = setInterval(function() {
      tick_target.emit("tick");
    }, 1000);
  }
  util.inherits(Ticker, EventEmitter);
  module.exports = Ticker;
```

EventEmitter 상속



# Node Package Manager

## : 모듈과 패키지, 패키지 매니저 (npm)

```
npm install 패키지명 // 현재 디렉터리에 패키지 설치  
npm install -g 패키지명 // 패키지를 전역 설치  
npm install 패키지명 --save // 설치하고 package.json에 추가
```

- ▶ 모듈: 독립적인 실행영역을 가지는 개별 자바스크립트 파일
- ▶ 패키지 : 여러 개의 연관된 모듈을 모아둔 것
- ▶ 패키지 매니저
  - ▶ 자신이 만든 모듈들을 하나의 패키지로 관리할 수 있고 다른 사람들이 만들어둔 패키지를 자신의 패키지에 라이브러리로 설치하여 사용할 수 있다.

- ▶ npm : 노드의 기본 패키지 매니저
  - ▶ 패키지 초기화 : package.json 생성

```
npm init
```

- ▶ 외부 패키지 설치 (예: express 패키지 설치)

```
npm install express --save
```

### package.json

```
{  
  "name": "express-spt",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": ""  
  },  
  "author": "",  
  "license": "ISC",  
  "dependencies": {  
    "express": "^4.16.4"  
  }  
}
```



# Express

- ▶ Node.js의 http 내장 모듈을 이용하면 웹서버를 구성할 수 있지만, 많은 내용을 직접 만들어야 한다
  - ▶ Express 모듈은 웹 서버 구현을 위한 미들웨어와 라우터 기능을 제공  
-> 간단한 코드로 웹 서버의 기능 구현 가능
  - ▶ Express 모듈의 설치

```
npm install express --save
```

- ▶ Express 객체의 주요 메서드

메서드명	설명
set(name, value)	서버 설정 속성 지정
get(name)	서버 설정 속성 읽기
use([path,] function)	미들웨어 사용
get([path,] function)	GET 요청 정보 처리
post([path,] function)	POST 요청 정보 처리

app.js

```
// Express 모듈 불러오기
const express = require('express'),
http = require('http');

// Express 객체 생성
const app = express();

// Express 기본 포트 속성을 설정
app.set('port', process.env.PORT || 3000);

// Express 서버 Start
http.createServer(app).listen(app.get('port'), () => {
  console.log('Web Server is running');
});
```

# Express

## : 클라이언트 요청의 처리

### ▶ 정적 파일의 제공

#### ▶ `express.static` 미들웨어 함수

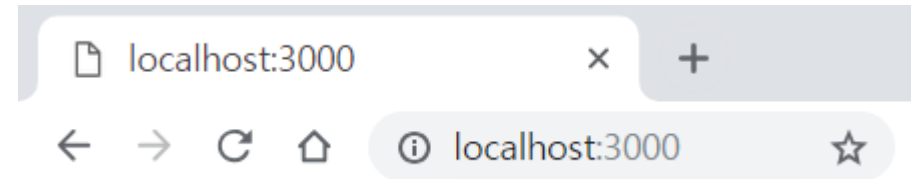
```
// ...  
// Static 파일 제공  
app.use(express.static("public"));
```

public 디렉터리 내부의 파일을 정적 파일로 제공

### ▶ GET 메서드 요청의 처리

#### ▶ `app.get([path,] function)`

```
// GET 메서드 요청의 처리  
app.get("/", function(req, res) {  
  console.log("[GET] / ");  
  res.writeHead(200, {'Content-Type': 'text/html; charset=utf8'});  
  res.write("Express Welcomes You");  
  res.end();  
});
```



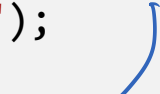
# Express

## : Response 객체

### ▶ http 모듈의 응답객체 주요 메서드

메서드	설명
<code>writeHead(statusCode [, statusMessage][, headers])</code>	응답 헤더를 생성. 200이면 성공 코드
<code>write(data [, encoding][, callback])</code>	응답 본문(body) 데이터 생성. 중복 호출 가능
<code>end([data][, encoding][, callback])</code>	응답을 클라이언트로 전송

```
res.writeHead(200, {'Content-Type': 'text/html; charset=utf8'});  
res.write("Express Welcomes You");  
res.end();
```



### 주요 MIME Type

Content-Type	설명
text/html, text/css, text/json	텍스트 (HTML, CSS, JSON 등)
image/jpeg, image/png	이미지 파일 (JPEG, PNG 등)
video/mpeg, audio/mp3	미디어 파일 (MPEG, MP3 등)
application/zip	압축파일

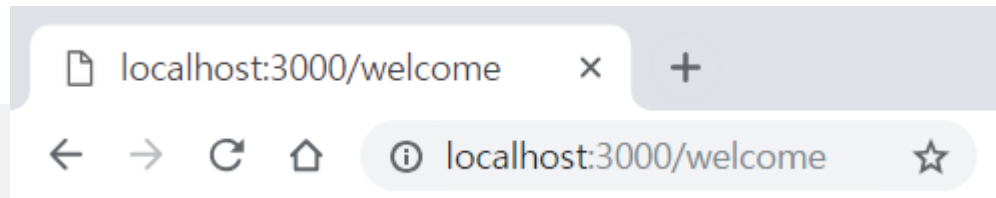
# Express

## : Response 객체

### ▶ express 모듈의 응답객체 추가 주요 메서드

메서드	설명
<code>header(field [, value])</code>	응답 헤더를 설정
<code>status(code)</code>	HTTP 상태 코드를 반환
<code>send([body])</code>	클라이언트에 응답 데이터를 전송
<code>json(obj)</code>	JavaScript 객체를 json으로 전송
<code>redirect([status,] path)</code>	웹 페이지 경로를 강제로 이동
<code>render(view [, locals][, callback])</code>	뷰 엔진을 사용해 문서를 만든 후 전송

```
app.get("/welcome", function(req, res) {  
  res.status(200)  
    .header("Content-Type", "text/html; charset=utf-8")  
    .send("Express Welcomes You");  
});
```



# Express

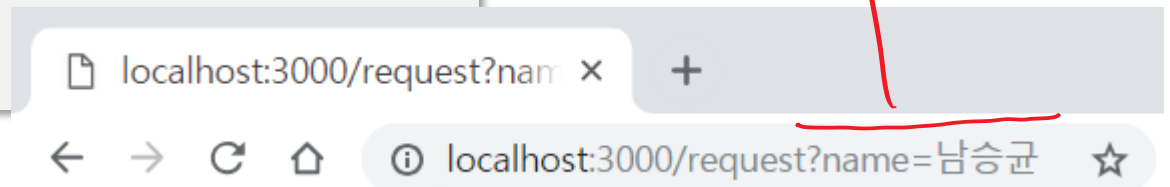
## : Request 객체

### ▶ express 모듈의 요청 객체 주요 속성

속성	설명
query	클라이언트에서 <b>GET</b> 방식으로 전송한 요청 파라미터를 확인
body	클라이언트에서 <b>POST</b> 방식으로 전송한 요청 파라미터를 확인 단, <b>body-parser</b> 등 외장 모듈을 이용
header(field)	요청 헤더를 확인

```
app.get("/request", function(req, res) {  
  let paramName = req.query.name;  
  if (paramName == undefined || paramName.length == 0) {  
    // 처리 코드  
    // ...  
  } else {  
    // 처리 코드  
    // ...  
  }  
});
```

?name=남승균



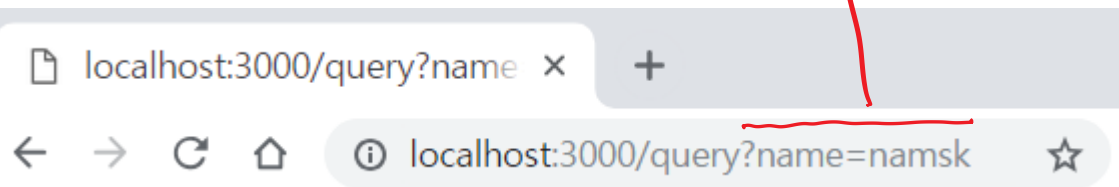
# Express

## : Request

### Query String

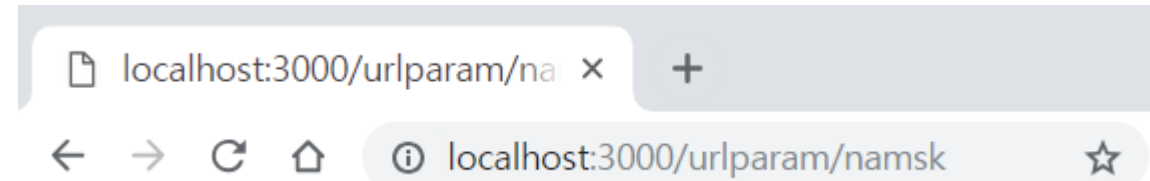
- ▶ URL 뒤에 ? 기호를 붙여 전송  
`?key1=value1&key2=value2`
- ▶ Request의 `query` 객체로 접근 가능

```
app.get('/profile', function(req, res) {  
  let userName = req.query.name;  
  // ...  
});
```



### URL Parameter

- ▶ URL 주소의 일부로 파라미터를 전송  
`/profile/sean`
- ▶ 요청 경로 문자열 상에 토큰을 설정해 두면 Request의 `params` 객체로 접근 가능



```
app.get("/urlparam/:name", function(req, res) {  
  let userName = req.params.name;  
  // ...  
});
```

# Express

## : View Engine

- ▶ View Engine : 클라이언트에 보낼 응답 문서를 만들 때 사용
  - ▶ 템플릿을 미리 만들어 두고 그 템플릿을 사용해 응답 웹 문서를 만들어냄
  - ▶ 주요 View Engine : EJS, PUG 등

```
npm install ejs --save
```

← EJS 설치

- ▶ Express에 View Engine과 Views (템플릿 디렉터리) 설정

```
// View 엔진 설정
app.set("view engine", "ejs");
app.set("views", __dirname + "/views");
```

- ▶ Response 객체의 render 메서드를 이용하면 View Engine의 설정에 따라 템플릿을 렌더링

```
app.get('/render', function(req, res) {
  res.status(200)
    .contentType("text/html; charset=utf-8")
    .render("render");
});
```

View 이름

EJS 템플릿 문법

<%= 객체 %> : 객체 데이터의 출력

<% ~ %> : JavaScript 코드 수행



# Express

## : View Engine

- ▶ Template에 반영할 데이터를 전달하고 싶다면
  - ▶ 두 번째 인자값으로 객체 목록을 부여, 템플릿 내에 반영하여 렌더링

```
app.get("/web/friends/list", function(req, res) {  
  database.collection('friends').find().toArray(function(err, result) {  
    res.render("friends_list", { friends: result });  
  })  
});
```

View 이름

Template에 반영할 데이터 목록 객체

EJS

friend\_list.ejs

```
<% for (let i = 0; i < friends.length; i++) { %>  
  <p><%= friends[i].name %></p>  
<% } %>
```

결과 HTML

```
<p>둘리</p>  
<p>도우너</p>  
<p>마이콜</p>
```

# Express

## : POST와 body-parser 미들웨어

- ▶ HTTP POST 메서드 요청은 HTTP 요청 객체의 **Body**에 실려 넘어온다

- ▶ HTTP 요청 Body의 정보를 해석할 body-parser 미들웨어가 필요

```
npm install body-parser --save
```

← body-parser 설치

- ▶ Express에 body-parser 사용 등록

```
// body-parser
const bodyParser = require('body-parser');
// ...
// Express에 body-parser 사용 등록
app.use(bodyParser.urlencoded( { extended: false }));
```

- ▶ body-parser가 Express에 등록되어 있으면 Request의 body 객체를 이용해 데이터를 전달받을 수 있다

```
app.post("/web/friends/save", function(req, res) {
  let name = req.body.name;
  let species = req.body.species;
  // ...
})
```