

# Express로 웹 서버 구축하기

🕒 생성일	@2024년 11월 4일 오후 12:25
☰ 태그	

▼ Node.js 프로젝트 생성

1. package.json

- 정의
  - 프로젝트에서 사용하는 패키지와 버전관리, 패키지의 의존성 관리를 위한 문서
- 생성

```
npm init
```

- 설정 항목

입력 항목	구성 항목	설명
pacakge name	name	프로젝트 이름이자 패키지 이름
version	version	패키지 버전
description	description	패키지 설명
entry point	main	자바스크립트 실행 파일
test command	scripts[test]	테스트 명령어
git repository		코드를 저장한 깃(git) 저장소 주소
keywords	keywords	npm 내에서 검색을 위한 키워드
license	license	패키지 라이선스 정보 ( default : ISC )

2. 필요한 패키지 설치

```
npm install 패키지명
```


3. 프로젝트 실행 명령어

```
node 파일명.js
```

▼ Express.js

Express - Node.js web application framework

Express 5.0 beta documentation is now available.  
ex <https://expressjs.com/>



- 정의

**Fast, unopinionated, minimalist web framework for Node.js**  
Node.js를 위한 빠르고 개방적인 간결한 웹 프레임워크

- 설치

```
npm install express
```

▼ 라우팅 처리하기

▼ 라우팅(Routing)

- 정의

특정 엔드 포인트에 대한 클라이언트 요청에 애플리케이션이 응답하는 방법을 결정

- 구조

```
app.METHOD(path, handler)
```

항목	설명
app	express의 인스턴스
METHOD	HTTP 요청 메소드 ( GET, POST 등 )
path	서버에서의 경로
handler	라우트(Route)가 일치할 때 실행되는 함수 ( 매개변수로 request 객체, response 객체를 기본으로 받음 )

▼ 라우트(Route) 메소드

- 지원가능 메소드

**get, post, put, head, delete, options, trace, copy, lock, mkcol, move, purge, propfind, proppatch, unlock, report, mkactivity, checkout, merge, m-search, notify, subscribe, unsubscribe, patch, search, connect**

- 주 사용 메소드
  - get, post, put, delete

▼ 라우트(Route) 경로 설정

- 문자열 기반
- 문자열 패턴 기반
- 정규 표현식 기반

▼ 라우트(Route) 핸들러

- 매개변수

순서	대상	데이터 타입	설명
1	Request	객체	클라이언트 요청 객체
2	Response	객체	클라이언트 응답 객체
3	next	객체	다음 미들웨어 함수를 가리키는 객체

▼ 사용방법

- 핸들러를 하나만 선언

```
app.get('/', function(req, res, next){
  // 실행코드
})
```

- 핸들러를 여러 개 선언

```
app.get('/', function(req, res, next){
  // 해당 핸들러 실행코드
  next(); // 다음 콜백 함수 호출
}, function(req, res, next){
  // 해당 핸들러 실행코드
})
```

- 핸들러를 배열로 선언

```
const func1 = function(req, res, next){
  // 해당 핸들러 실행코드
```

```

    next(); // 다음 콜백 함수 호출
};

const func2 = function(req, res, next){
    // 해당 핸들러 실행코드
    next(); // 다음 콜백 함수 호출
};

app.get('/', [ func1, func2 ])

```

#### ▼ 응답 메소드

메소드	설명
res.download()	파일을 다운로드
res.end()	응답 프로세스를 종료
res.json()	JSON 응답을 전송
res.jsonp()	JSONP 지원을 통해 JSON 응답을 전송
res.redirect()	특정 경로로 재요청
res.render()	화면을 렌더링
res.send()	다양한 유형의 응답을 전송
res.sendFile()	파일을 octet 스트림(8비트 데이터)으로 전송
res.sendStatus()	응답 상태 코드를 설정한 후 해당 코드를 응답 본문(body)에 담아서 전송

#### ▼ app.route()

- 모듈식 라우터
- 하나의 라우트 경로를 기반으로 각 라우트 메소드를 처리

#### ▼ express.Router

- 용도
  - 라우트 처리를 여러 개의 파일로 분리해서 구현
- 사용
  1. 용도별 파일을 생성해서 라우트 모듈로 설정

```

// route.js

const express = require('express');
const router = express.Router();

// 라우트 설정 : express에 등록할 때 사용한 경로를 기본 base로 함

module.exports = router;

```

#### 2. express 설정 시 해당 라우트 모듈 등록

```

const express = require('express');
const route = require('./routes/route');
const app = express();

// express 기본 설정

app.use('/* 라우트 모듈 등록시 기본 경로*/', route);

```

#### ▼ 에러 처리하기

- Express 내장 에러 핸들러 사용

```
app.Method(path, function(req, res, next){
    throw new Error('에러 발생');
})
```

- Express에 별도 등록한 에러 핸들러 사용

```
// Express에 미들웨어 함수로 에러 핸들러 등록
app.use(function(err, req, res, next){
    // err을 이용하여 에러와 관련된 정보를 가져옴
    // 에러 처리 코드
});

// 각 라우팅 처리 시 next()를 이용하여 에러 처리 핸들러 호출
app.Method(path, function(req, res, next){
    next(new Error('에러 발생'));
})
```

#### ▼ 정적 파일 제공하기

```
// 미들웨어 express.static 사용
// 해당 폴더가 '/'로 설정됨
app.use(express.static('/* 정적 파일 폴더 */'));

// 별도 마운트 경로 설정
app.use('/static', express.static('/* 정적 파일 폴더 */'));
```

#### ▼ 미들웨어 모듈

- 정의

**Middleware** functions are functions that have access to the request object ( `req` ), the response object ( `res` ), and the next middleware function in the application's request-response cycle

미들웨어 함수는 요청 객체, 응답 객체 그리고 애플리케이션의 요청-응답 주기에서 다음 함수에 접근할 수 있는 함수

#### ▼ 적용범위

```
// 전체 라우터 적용
app.use(middleware());

// 특정 라우터에 적용
app.METHOD(path, middleware(), handler);
```

#### ▼ 종류

미들웨어 모듈	설명
<b>body-parser</b>	HTTP 요청 body를 파싱
<b>compression</b>	HTTP 요청들을 압축
<b>connect-rid</b>	고유한 요청 ID를 생성
<b>cookie-parser</b>	쿠키 헤더를 파싱하고 req.cookies에 할당
<b>cookie-session</b>	쿠키 기반의 세션을 생성
<b>cors</b>	Cross-origin resource sharing (CORS)를 활성화
<b>csrf</b>	CSRF 취약점을 방어
<b>errorhandler</b>	개발 중에 발생하는 에러를 핸들링하고 디버깅

미들웨어 모듈	설명
<b>method-override</b>	헤더를 이용해 HTTP method를 덮어씀
<b>morgan</b>	HTTP 요청 로그를 남김
<b>multer</b>	multi-part 폼 데이터를 처리
<b>response-time</b>	응답 시간을 기록
<b>serve-favicon</b>	파비콘을 제공
<b>serve-index</b>	주어진 경로의 디렉토리 리스트를 제공
<b>serve-static</b>	정적 파일을 제공
<b>express-session</b>	서버 기반의 세션을 생성
<b>connect-timeout</b>	HTTP 요청 처리를 위해 timeout을 생성
<b>vhost</b>	가상 도메인을 생성

#### ▼ body-parser

- Express 4.16 version 이상 Express에 내장 별도의 설치 없이 내장 함수 이용

#### • compression

#### ▼ cookie-session

세션 생성 옵션

- **name**: 설정할 쿠키 이름 (default: 'session')
- **keys**: 쿠키에 서명하기 위해 사용할 키 목록
- **secret**: 키가 제공되지 않는 경우 단일 키로 사용되는 문자열
- **cookie Options {}** : 쿠키 생성 옵션
  - **maxAge**: 만료 시간을 밀리초 단위로 설정
  - **expires**: 만료 날짜를 GMT 시간으로 설정
  - **path** : cookie의 경로 (default: '/')
  - **domain** : 도메인 네임 (default: 'loaded')
  - **secure** : https에서만 cookie 사용할 수 있도록 합니다.
  - **httpOnly** : 웹서버를 통해서만 cookie 접근할 수 있도록 합니다.
  - **signed** : cookie가 서명을 지정합니다. (내 서버가 쿠키를 만들었다는 검증)

#### ▼ express-session

세션 생성 옵션

- **secret**: 보안을 위한 임의의 문자열 (secret key)
- **resave**: 세션 데이터가 바뀌기 전까지 세션저장소의 값을 저장 여부 (default: false)
- **saveUninitialized**: 세션이 필요하기 전에 세션을 구동 여부 (default: true)
- **store**: 세션저장소를 지정
- **cookie**: 세션 쿠키 설정

#### • session-file-store

세션 정보를 파일로 저장해서 관리

#### ▼ cors

CORS 옵션

- **origin**: 허용할 도메인
- **method**: 허용할 HTTP 요청 (ex. ['GET', 'POST', 'PUT', 'DELETE'])
- **allowedHeaders**: 접근 허용할 CORS 요청 헤더 정보  
(ex. ['Content-Type', 'Authrization'])
- **exposedHeaders**: 접근 허용할 CORS 응답 헤더 정보  
(ex. ['Content-Range', 'X-Content-Range'])
- **credentials**: 접근 허용할 CORS 자격 증명 정보
- **maxAge**: 실행 전 요청의 결과 (Access-Controller-Allow-method) 헤더에 포함된 정보를 캐시할 수 있는 기간
- **preflightContinue**: CORS 실행 전 응답을 다음 핸들러에 전달 여부

- optionsSuccessStatus : 일부 레거시 브라우저(ex. IE11)에서  
초크에 연결되므로 성공적인 요청에 제공

## ▼morgan

```
# morgan 미들웨어 등록하기 : morgan(format, options)
  · format: 미리 정의된 이름의 문자열, 형식 문자열, 또는 로그 항목을 생성하는 함수

# format
combined: 운영 환경에서 사용, 불특정 다수가 접속하기 때문에 IP를 로그에 남깁니다.
-> :remote-addr - :remote-user [:date[clf]] ":method :url HTTP/:http-version" :status

common
-> :remote-addr - :remote-user [:date[clf]] ":method :url HTTP/:http-version" :status

dev: 개발을 위해 response에 따라 색상이 입혀진 축약된 로그를 출력합니다.
-> :method :url :status :response-time ms - :res[content-length]

short: 기본 설정보다 짧은 로그를 출력합니다. (응답시간 포함)
-> :remote-addr :remote-user :method :url HTTP/:http-version :status :res[content-length]

tiny: 최소화된 로그를 출력합니다.
-> :method :url :status :res[content-length] - :response-time ms
```

## ▼multer

```
<form action="/profile" method="post" enctype="multipart/form-data">
  <input type="file" name="avatar" />
</form>
```

### 1. 디스크 저장장소 설정

multer 모듈을 사용해서 클라이언트로부터 전송된 파일을 업로드 처리하기 위해서는  
먼저 디스크 저장장소에 대한 객체를 생성해야 합니다.

multer의 `diskStorage()` 함수를 통해 파일이 저장될 위치와 파일명을 어떻게 만들 것인지에 대한 정의를 합니다.

```
const storage = multer.diskStorage({ // 디스크 저장소 정의
  destination: function (req, file, cb) {
    cb(null, 'uploads/') // cb 콜백 함수를 통해 전송된 파일 저장 디렉토리 설정
  },
  filename: function (req, file, cb) {
    const uniqueSuffix = Date.now() + '-' + Math.round(Math.random() * 1E9);
    cb(null, file.fieldname + '-' + uniqueSuffix); // cb 콜백 함수를 통해 전송된 파일 이름 설정
  }
});
```

```
const upload = multer({ storage: storage }); // multer 객체 생성
```

### 2. 파일 업로드 처리

// 싱글 파일 업로드

```
app.post('/profile', upload.single('avatar'), function (req, res, next) {
  console.log(req.file); // avatar 이름의 싱글 파일
  console.log(req.body); // 일반적인 폼 데이터
});
```

// 다중 파일 업로드

```
app.post('/photos/upload', upload.array('photos', 12), function (req, res, next) {
  console.log(req.files); // photos 이름의 멀티 파일
```

```
});
```

- response-time
- connect-timeout