**[Node.js](https://namu.wiki/w/Node.js)**

최근 수정 시각: 2021-09-18 10:05:20

* [시스템 소프트웨어](https://namu.wiki/w/%EB%B6%84%EB%A5%98:%EC%8B%9C%EC%8A%A4%ED%85%9C%20%EC%86%8C%ED%94%84%ED%8A%B8%EC%9B%A8%EC%96%B4)

|  |  |
| --- | --- |
| **Node.js®** | |
|  | |
| **종류** | [JavaScript](https://namu.wiki/w/JavaScript) 런타임(실행환경) |
| **개발** | Node.js contributors |
| **버전** | 16.9.1 2021년 9월 10일 업데이트 |
| **공개일** | 2009년 5월 27일 |
| **라이선스** | [MIT 라이선스](https://namu.wiki/w/MIT%20%EB%9D%BC%EC%9D%B4%EC%84%A0%EC%8A%A4) |
| **링크** |  |

[**1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**개요**[**[편집]**](https://namu.wiki/edit/Node.js?section=1)

2009년 5월 27일 처음 소개된 Node.js는 [오픈 소스](https://namu.wiki/w/%EC%98%A4%ED%94%88%20%EC%86%8C%EC%8A%A4) [JavaScript](https://namu.wiki/w/JavaScript) 엔진인 [크롬](https://namu.wiki/w/%ED%81%AC%EB%A1%AC(%EC%9B%B9%20%EB%B8%8C%EB%9D%BC%EC%9A%B0%EC%A0%80)) V8에 비동기 이벤트 처리 라이브러리인 libuv를 결합한 플랫폼이다.

**다시 말해, JavaScript로 브라우저 밖에서**[**서버**](https://namu.wiki/w/%EC%84%9C%EB%B2%84)**를 구축하는 등의 코드를 실행할 수 있게 해주는 런타임 환경이다.**Ryan Dahl이 처음 개발했으며,[[1]](https://namu.wiki/w/Node.js?from=Nodejs#fn-1) 처음엔 리눅스와 macOS만 지원되었으나 2011년 7월에 Windows 버전도 발표되었다.  
  
**빈번한 I/O처리에 있어서의 우수한 성능, 서버 확장의 용이성, 무엇보다도 JavaScript라는 프론트엔드 필수 언어로 백엔드까지 작성할 수 있다는 엄청난 장점 때문에 출시 이후로 빠르게 점유율을 높여가고 있다. 특히**[**트위터**](https://namu.wiki/w/%ED%8A%B8%EC%9C%84%ED%84%B0)**등 엄청나게 많은 양의 인풋 아웃풋 데이터를 처리해야 하는 서비스에 있어서 강점이 두드러진다.**  
2014년 12월 한때 Node.js의 포크인 [io.js](https://namu.wiki/w/io.js)가 나타나면서 Node.js 0.12 버전, io.js 3.3 버전까지 서로 분열된 모습으로 이어지는 듯 했지만, 2015년 9월에 Node.js 4.0 버전으로 병합되어 현재에 이르렀다.  
  
2021년 6월 18일 기준으로 최신 버전은 16.3.0, LTS 버전은 14.17.1이다. 최신 버전은 기능이 불안정하거나 일부 모듈[[2]](https://namu.wiki/w/Node.js?from=Nodejs#fn-2)(패키지)이 작동하지 않을 수 있으므로 안정성을 보장하고 싶으면 LTS 버전을 사용하는 게 좋다.

[**2.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**사용해 보기**[**[편집]**](https://namu.wiki/edit/Node.js?section=2)

[**2.1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**설치**[**[편집]**](https://namu.wiki/edit/Node.js?section=3)

대표적인 방법은 홈페이지에 대문짝만하게 나와있는 다운로드 버튼을 누르는 것이다.  
  
다만 Node.js를 이용해 구현된 프로그램 혹은 웹사이트를 실행하거나 빌드하다 보면 프로그램마다 다른 노드 버전을 지원하는 것을 경험할 수 있는데, 이 때는 NVM(Node Version Manager)을 이용해서 쉽게 다른 버전의 node.js를 설치하고 전환할 수 있다. 사실 이 목적이 아니라도, node.js를 쉽게 업데이트하기 위해 nvm을 쓰게 된다.  
  
소스코드에서 직접 빌드해서 쓸 수도 있다. [이 곳](https://github.com/nodejs/node/blob/master/BUILDING.md)을 참고하면 된다. 다만 방대한 노드 코드를 모두 빌드해야 하기에 시간이 많이 걸린다는 점을 고려하자.

[**2.1.1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**NVM을 이용한 설치**[**[편집]**](https://namu.wiki/edit/Node.js?section=4)

비공식 Node.js 버전 관리자 프로그램인 [NVM](https://github.com/nvm-sh/nvm)을 이용해 간단히 설치 및 관리를 할 수 있다.  
  
bash가 지원되는 환경인 리눅스, 맥[[3]](https://namu.wiki/w/Node.js?from=Nodejs#fn-3)에서는 아래 코드 두 줄 중 하나를 터미널에 입력하여 설치할 수 있다.

$ curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash

$ wget -qO- https://raw.githubusercontent.com/nvm-sh/nvm/v0.38.0/install.sh | bash

만약 자신이 bash와 명령어가 호환되지 않는 fish를 사용한다면 [여기](https://github.com/jorgebucaran/nvm.fish#readme)서 fish용 nvm을 설치하거나 [이 곳](https://github.com/FabioAntunes/fish-nvm#readme)에서 fish용 bash-nvm의 wrapper를 추가로 설치해주어야 한다. zsh 사용자는 [여기](https://github.com/lukechilds/zsh-nvm#readme)서 zsh용 플러그인을 설치해주면 된다.  
  
설치 이후 다음의 명령어를 이용해 Node.js를 설치할 수 있다.

$ nvm install 14.15.1 # 예시로 14.15.1 버전의 node와 그에 맞는 npm 설치

$ nvm use 14.15.1 # 해당 버전 활성화

[**2.1.2.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**리눅스에서 패키지 매니저를 이용한 설치**[**[편집]**](https://namu.wiki/edit/Node.js?section=5)

주요 리눅스 배포판에서는 NodeSource에서 제공하는 바이너리를 버전별로 패키지 매니저를 이용하여 받아 쓸 수 있다. [우분투](https://namu.wiki/w/%EC%9A%B0%EB%B6%84%ED%88%AC), [데비안](https://namu.wiki/w/%EB%8D%B0%EB%B9%84%EC%95%88) 리눅스에서는 아래와 같이 하면 된다.

$ curl -sL https://deb.nodesource.com/setup\_[메이저 버전].x | sudo -E bash -

$ sudo apt-get install -y nodejs

예를들어 Node.js 14 버전대를 설치하고자 한다면 다음과 같이 실행한다.

$ curl -sL https://deb.nodesource.com/setup\_14.x | sudo -E bash -

$ sudo apt-get install -y nodejs

다만 이렇게 Node.js를 설치하면 한 번에 하나의 버전대만 쓸 수 있다는 단점이 있다.  
  
만약 다른 리눅스 배포판을 쓴다면 [이 곳](https://github.com/nodesource/distributions)에서 직접 자신의 배포판에 맞는 명령줄을 찾아 입력하자.[[4]](https://namu.wiki/w/Node.js?from=Nodejs#fn-4)

[**3.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**특징**[**[편집]**](https://namu.wiki/edit/Node.js?section=7)

[**3.1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**비동기 이벤트**[**[편집]**](https://namu.wiki/edit/Node.js?section=8)

Node.js는 고성능의 비동기 애플리케이션 작성 플랫폼으로 구상되었다. 최초 작성자인 Ryan Dahl은 웹 프로그램 분야에서 브라우저의 JavaScript 엔진이 UI 스레드를 공유할 수 밖에 없는 환경[[7]](https://namu.wiki/w/Node.js?from=Nodejs#fn-7)에서 락을 피하고자 시분할과 이벤트로 비동기 프로그램을 작성하는 모습을 보고 괜찮은 방법론이라 생각해 만들었다고 한다.  
  
이때 프로그램은 리액터 패턴에 의한 [CPS(Continuation-Passing Style)](http://en.wikipedia.org/wiki/Continuation-passing_style)로 작성되는데, 이것이 Node.js에 고성능을 선사하는 기법인 동시에 비숙련자에게 디버깅 지옥을 만드는 애증의 관계를 지니고 있다. 궁금하면 위의 간단한 http 서버에서 createServer가 인자로 전달받은 함수를 http 요청에 이어지는 실행지점으로 사용한다는 점을 보도록 하자. Node.js가 자랑하는 고성능 비동기 프로그램을 작성하기 위해서는 파일시스템을 포함한 프로그램 전체가 비동기화 지점마다 저렇게 CPS로 연결되어야 한다.  
  
이 CPS 특징 때문에 Node.js 초심자는 **콜백 지옥**을 경험하게 된다. 쉽게 설명하자면 코드가 무한 들여쓰기로 작성돼서 소스코드가 대각선으로 쓰여지는 현상을 말하는데, 이를 피하는 방법 중 하나는 익명 함수를 포기하고 아래와 같이 작성하는 것이 있다.

**const** http = require('http');

**const** server = http.createServer(helloResponse);

**function** helloResponse(req, res) {

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

res.end('Hello World\n');

}

server.listen(8080, '127.0.0.1');

console.log('Server running at http://127.0.0.1:8080/');

함수를 선언하기 전에 사용하고 있지만 JavaScript는 함수를 굳이 코드의 위쪽에 선언할 필요 없이 해당 스코프 안에 정의만 돼 있으면 자동 호이스팅이 돼서 사용할 수가 있다. 이게 깔끔하지 않다고 생각해서 콜백 함수를 createServer 위쪽에 선언할 생각을 할 수도 있는데, 이렇게 짜면 피하려고 했던 콜백 지옥에 다시 뛰어드는 꼴이 된다. 콜백을 실행하는 순서는 함수의 자연스러운 실행 순서와 정반대인 아래에서 위로 올라가는 방향이기 때문이다.  
  
콜백 지옥에서 벗어나기 위한 발버둥의 산물로 Generator와 Promise가 등장하였다. 이후 ES2017에서는 Promise를 기반으로 동작하는 async/await 키워드가 도입되어 타 프로그래밍 언어와 거의 차이가 없는 방식으로 비동기적 코드를 쓸 수 있게 되었다.

[**3.2.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**오류처리**[**[편집]**](https://namu.wiki/edit/Node.js?section=9)

노드는 기본적으로 오류 핸들링을 안하면 예외발생으로 프로그램이 꺼진다. 하다못해 보안접속 하는 형태에 피치못할 사정으로 접속을 항시 유지한다면 매 주기별로 재응답 해주겠으나 혹시모를 접속끊김이 발생할 여지는 많은데 절대로 간과하면 안되는 형태로 되어있다. 따라서 오류 핸들링을 통한 재접속을 하던가 다음 처리가 요청될 때까지 접속 끊김을 명확하게 제어해야한다.  
  
웹 GET 요청의 예제 error 핸들링 참고.

**const** http = require("http");

**const** req = http.request( {host:"ip", path:"/"}, (res) => {

**let** text = "";

res.on("data", (chunk) => {

text += chunk;

});

res.on("close", () => {

console.log(text);

});

res.on("error", () => {

*//예를들면 여기에 횟수제한, 지연대기, req.end(); 형태로 쓴다*

});

});

req.on("timeout", () => {

*//예를들면 이러한 중복되는 재시도 기능을 특정 함수로 재사용한다.*

});

req.on("error", (e) => {

*//예를들면 이러한 중복되는 재시도 기능을 특정 함수로 재사용한다.*

});

req.end();

[**3.3.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**멀티 프로세싱**[**[편집]**](https://namu.wiki/edit/Node.js?section=10)

Node.js는 단일 프로세스에서 작동하기 때문에 멀티코어를 완전히 사용하려면 코어 갯수만큼의 프로세스를 띄우고 라우터나 로드 밸런서 등으로 요청을 각 프로세스로 분산시켜주어야 한다. 다만 서버에서 Node.js 하나만 돌리지는 않고 Node.js는 CPU보다는 I/O에 유리한 녀석이므로 꼭 멀티프로세싱을 할 필요는 없다. 멀티코어를 더 잘 활용하는 [DB](https://namu.wiki/w/%EB%8D%B0%EC%9D%B4%ED%84%B0%EB%B2%A0%EC%9D%B4%EC%8A%A4) 등 다른 프로세스에 자원을 양보해주자. 물론 컨텐츠 딜리버리 서버 같이 양보해 줄 다른 프로세스가 없는 경우는 예외다.

[**4.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**패키지 매니저**[**[편집]**](https://namu.wiki/edit/Node.js?section=11)

[**4.1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**공식 패키지 매니저**[**[편집]**](https://namu.wiki/edit/Node.js?section=12)

[**4.1.1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**[npm](https://namu.wiki/w/npm" \o "npm)**[**[편집]**](https://namu.wiki/edit/Node.js?section=13)

Node.js Package Manager의 약자로, 공식 홈페이지는 [이곳](https://www.npmjs.com/)이다.  
  
2010년 1월에 도입된 Node.js 패키지 관리자이며, [Java](https://namu.wiki/w/Java)의 [Maven](https://namu.wiki/w/Maven), [Python](https://namu.wiki/w/Python)의 pip 정도의 입지를 가진 플랫폼이기도 하다. 대부분 Node.js와 함께 설치된다.

[**4.2.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**비공식 패키지 매니저**[**[편집]**](https://namu.wiki/edit/Node.js?section=14)

[**4.2.1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**yarn**[**[편집]**](https://namu.wiki/edit/Node.js?section=15)

Facebook에서 2016년 8월에 발표한 패키지 및 배포 관리자이다. 공식 홈페이지는 [이곳](https://yarnpkg.com/)이다.  
  
초창기에는 npm을 압도하는 성능으로 인기를 끌었는데, 2020년 현재는 npm이 개선을 거듭하여 설치 및 링크 퍼포먼스 측면에서 npm과 yarn은 큰 차이가 없다고 한다.  
  
npm과 동일한 저장소를 쓰기 때문에 yarn import만 해주면 npm에서 yarn으로 쉽게 마이그레이션할 수 있다.  
  
다만 yarn의 패키지 링크 알고리즘 상 npm과는 달리 모든 환경에서 동일한 의존성을 보장한다고 한다.  
  
Yarn 개발자들은 현재 node\_modules의 근본적인 문제점을 해결하기 위해 `.pnp.js` 라는 새로운 방식의 통합 의존성 파일을 이용하는 방법을 모색하고 있고, 이는 yarn 2버전대에서 이용해볼 수 있다.

[**5.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**여담**[**[편집]**](https://namu.wiki/edit/Node.js?section=16)

* [JavaScript](https://namu.wiki/w/JavaScript)의 인식이 '쉬운 언어'이다 보니, JavaScript로 서버도 만들 수 있다는 말만 듣고 도전했다가 [헬게이트](https://namu.wiki/w/%ED%97%AC%EA%B2%8C%EC%9D%B4%ED%8A%B8)를 경험하는 이들이 많다. 헌데 안타깝게도 JavaScript는 '난해한' 언어다. 다른 언어들이 그냥 에러를 내고 실행을 멈춰 버리는 버그를 JavaScript는 **잘못된** 방법으로라도 어떻게든 실행해버린다.[[8]](https://namu.wiki/w/Node.js?from=Nodejs#fn-8)[[9]](https://namu.wiki/w/Node.js?from=Nodejs#fn-9) 특히 this 객체와 모듈 시스템은 카오스 그 자체인데 이 둘을 안 쓸 수도 없다.
* [**나무위키**](https://namu.wiki/w/%EB%82%98%EB%AC%B4%EC%9C%84%ED%82%A4)**는 이 Node.js와 Node.js에서 가장 큰 규모의**[**웹 프레임워크**](https://namu.wiki/w/%EC%9B%B9%20%ED%94%84%EB%A0%88%EC%9E%84%EC%9B%8C%ED%81%AC)**인 Express로 개발되어 있다.**
* 2016년 6.0 버전부터 [ES6](https://developer.mozilla.org/ko/docs/Web/JavaScript/New_in_JavaScript/ECMAScript_6_support_in_Mozilla) 문법이 대거 추가되면서 [ES6 Promise](https://developer.mozilla.org/ko/docs/Web/JavaScript/Reference/Global_Objects/Promise)를 사용 가능하게 되었다. ES7에서는 Promise 패턴을 더 간편하게 사용할 수 있는 async/await 문법을 추가했다. 특히 async/await는 키워드 하나 추가로 비동기 함수를 동기식으로 짤 수 있는 마법 같은 편의성을 제공한다.[[10]](https://namu.wiki/w/Node.js?from=Nodejs#fn-10) 기존 Promise와의 조합도 물론 가능하다. 다만 VSCode 기준에서 디버깅을 할때 비동기함수로 진입하려고 하는 경우 비주얼 스튜디오에서 C#과 같이 함수에 들어가지는게 아니라 태스크 처리모듈로 들어간다.
* 12.0 버전부터 ES Modules도 실험적으로 도입하기 시작했다. 다만 지금까지 CommonJS로 만들어진 패키지들이 많이 존재하므로 기본적인 모듈 시스템은 14.0 버전까지도 CommonJS이다.
* 6.0부터 Windows XP와 Vista를,[[11]](https://namu.wiki/w/Node.js?from=Nodejs#fn-11) 14.0부터 **Windows 7**(...)을 지원하지 않는다. 따라서 Windows XP 등에서 최신 코드를 호스팅하고 싶다면 코드를 트랜스파일하거나[[12]](https://namu.wiki/w/Node.js?from=Nodejs" \l "fn-12) Windows 7 이상의 가상 머신을 활용해야 한다.

[**5.1.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**Node.js는 프레임워크?**[**[편집]**](https://namu.wiki/edit/Node.js?section=17)

우리나라에서는 프레임워크로 표현되기도 하는데,[[13]](https://namu.wiki/w/Node.js?from=Nodejs#fn-13) Node.js는 JavaScript 엔진인 V8에 스케쥴링 라이브러리(libuv)를 연결한 응용 런타임 플랫폼이라고 볼 수 있기 때문이다.  
  
과거 Node.js 홈페이지의 About에는 "As an asynchronous event driven framework, Node.js is..." 하는 설명이 있는데 이는 Node.js가 이벤트 드리븐에 있어 특정하게 지정된 디자인을 제공한다는 의미의 설명이다. 근데 이게 Node.js에 이벤트 드리븐 프레임워크를 포함된다는 것이지 이벤트 드리븐 프레임워크 자체가 Node.js라는 말은 아니다. 만약 저걸 근거로 Node.js가 프레임워크라고 불려야 한다면 특정 방법론을 구현하는 디자인을 내장한 거의 모든 현대 프로그래밍 언어가 프레임워크라 불려야 한다. 논란이 많기는 많았는지 현재의 홈페이지에서는 아예 "As an asynchronous event driven JavaScript runtime"로 내용이 바뀌었다. 굳이 6 버전대에 와서 홈페이지의 내용을 바꾼 것으로 보아, 이전의 문구가 실제로 오해를 불러일으켰다는 점을 인정했다고 볼 수 있다. 다만 이제 홈페이지의 내용도 정정되었으니, Node.js를 프레임워크로 표현하는 것은 명백한 잘못이라 할 수 있을 것이다.

[**5.2.**](https://namu.wiki/w/Node.js?from=Nodejs#toc)**성능 이야기**[**[편집]**](https://namu.wiki/edit/Node.js?section=18)

비동기 처리로 인하여 I/O 처리가 고성능이고 기반이 되는 구글 V8도 꽤 빠른 성능을 내는 [JIT](https://namu.wiki/w/JIT) 구현이지만 개별 프로세스 내부에서 작동하는 JavaScript 코드는 무조건 싱글 스레드로 실행한다는 제약이 걸려있어 병렬연산에는 취약하다. 단, Node.js에서 싱글 스레드는 성능상의 제약이 아니라 프로그래밍 모델에 포함된 사양이다. 복잡한 비동기 I/O 응용 프로그램을 싱글 스레드 JavaScript로 작성하여 결과물의 실행 속도와 개발 편의성 측면에서 좋다는 것이 Node.js의 근간을 이루는 개념이며, CPU의 높은 연산력을 요구하는 프로그램을 Node.js로 짜면 영 좋지 않은 성능이 나온다. 작성자인 Ryan Dahl은 순수하게 연산력이 중요한 요소는 [C](https://namu.wiki/w/C(%ED%94%84%EB%A1%9C%EA%B7%B8%EB%9E%98%EB%B0%8D%20%EC%96%B8%EC%96%B4))로 짜서 붙이라는 상식적인 사실을 이야기했다.  
  
Node.js가 고성능을 발휘하는 분야는 비교적 가벼운 I/O가 발생하는 대다수의 웹 서버나 동영상, 스트리밍 등의 컨텐츠 딜리버리 서버다. 사실 C로 똑같이 비동기 처리를 구현하면 Node.js보다 빨라지지만 코딩 난이도가 높은 편이다.  
  
다만, 비동기 처리 자체에 대해서는 Node.js만의 장점이라고 말하기는 힘들다. [C#](https://namu.wiki/w/C%23)의 경우 await/async 키워드 및 Task 등에 기반하여 비동기 프로그래밍을 언어적으로 지원하며, 이쪽은 멀티 스레드 기법까지 활용하기 때문에 더 완성도가 높다. 사실상 같은 싱글 스레드 언어인 [Python](https://namu.wiki/w/Python) 역시 비슷한 수준의 지원을 가지고 있다. 아예 언어 문법 차원에서 지원하는 케이스에는 도저히 비할 바가 못 되나 [Java](https://namu.wiki/w/Java)와 [C++](https://namu.wiki/w/C%2B%2B) 등도 별도의 라이브러리를 통해 어느 정도 비동기 구현의 난이도를 낮추는 방법을 제공한다. 아직 초안인 C++20는 이를 공식적으로 지원할 예정이다.  
  
또한, Boost.Asio와 같이 잘 설계된 C++ 라이브러리가 네트워크에 대해 비동기 처리를 지원하며 동기 처리를 하더라도 Node.js보다 성능이 뛰어나다는 결과를 보여주고 있다. 라이브러리가 잘 구현되더라도 C++가 비교적 어렵고 복잡한 언어이기 때문에 확산이 잘 안 될 뿐이다.