
의존성 관리. 어디까지 알고 있니

발표자 : 이휘찬

2022. 10. 20

작성자 : 부스트캠프 웹·모바일 이휘찬

1. Npm에 대해서 잘 알고 있나요?

1.1 외부 의존성, 그리고...

1.2 npm이 작동하는 방식

1.3 npm의 경쟁자들

2. Yarn 그리고 Yarn berry!

2.1 기존의 node_modules는 가라! Plug n Play

3. Performant npm, pnpm!

3.1 pnpm의 메인 컨셉트

3.2 굳이 hoisting을 해야할까? Symlinked node_modules

CHAPTER 1.

NPM에 대해서 잘 알고있나요?

단순히 외부 라이브러리를 설치한다 정도로 알고 계셨나요?
사실 저도 그랬답니다.

1.1 외부 의존성, 그리고...

의존성 (dependency)

간단하게 말해서 개발에 필요한 외부 라이브러리.

하지만 과연 의존성 관리도 이 말 한마디처럼 쉬울까?

1.1 외부 의존성, 그리고...

Npm이 해결하기 위해 노력해온 문제

사실 node가 아니더라도 다른 언어에서도 비슷해요...

01

의존성 지옥

외부 의존성이 되는 라이브러리도 의존성을 가질 수 있습니다.
라이브러리들의 의존성이 꼬일 때 문제가 생길 수 있습니다.

02

의존성 트리의 결정성

Npm은 설치 순서에 따라서 의존성 트리가 달라질 수 있습니다.
이는 의존성 lock 파일의 해시값이 달라지는것을 초래하고
느린 CI/CD 라는 문제를 불러올 수 있습니다.

1.1 외부 의존성, 그리고...

의존성 지옥 (dependency hell)

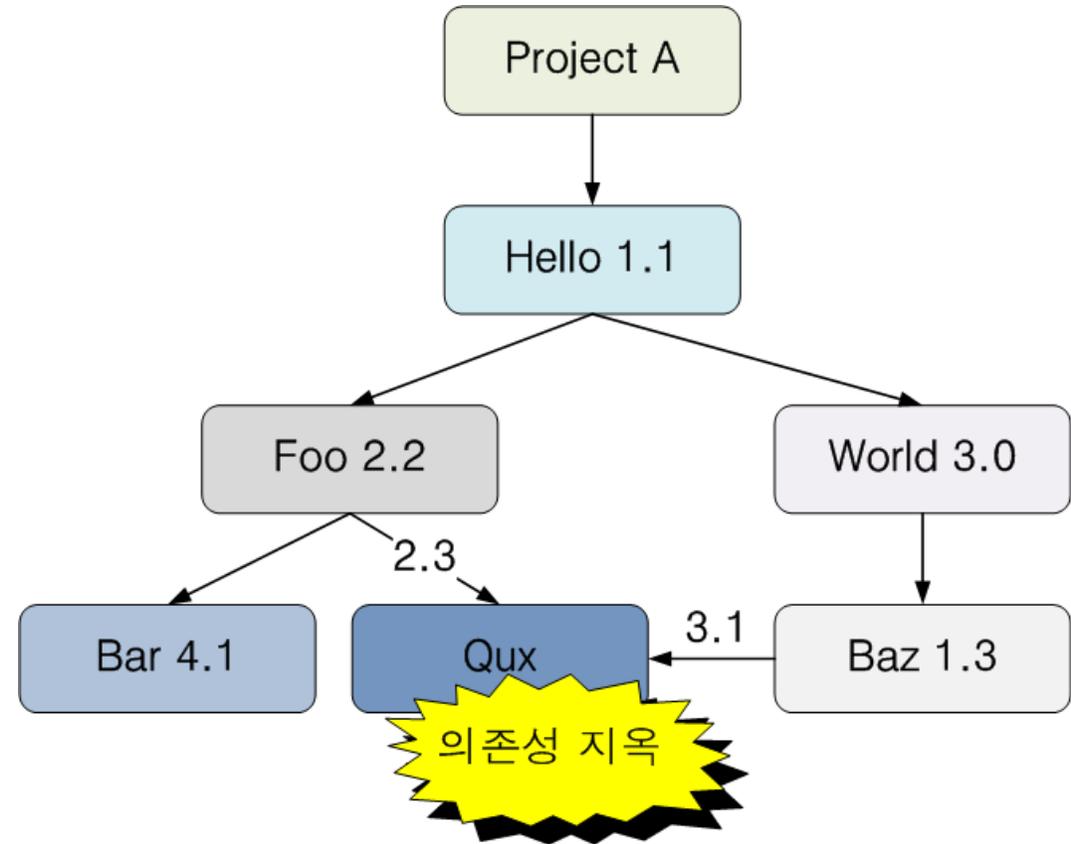
프로젝트를 개발하면서 여러 외부 의존성을 사용

외부 의존성도 여러 의존성을 가질 수 있음 (의존성의 의존성)

외부 의존성이 의존하는 라이브러리의 버전이 다를 수 있음

오른쪽 그림에서는 Foo가 의존하는 Qux의 버전과 Baz가 의존하는 버전이 다름

이럴 경우에는 Qux를 2개 설치해야겠지..?



실제 프로젝트에서는 의존성의 의존성으로 여러 개 물려있음

다들 많이 알고 있을 정적 사이트 생성기 Gatsby로 예를 들어보자

1.1 외부 의존성, 그리고...

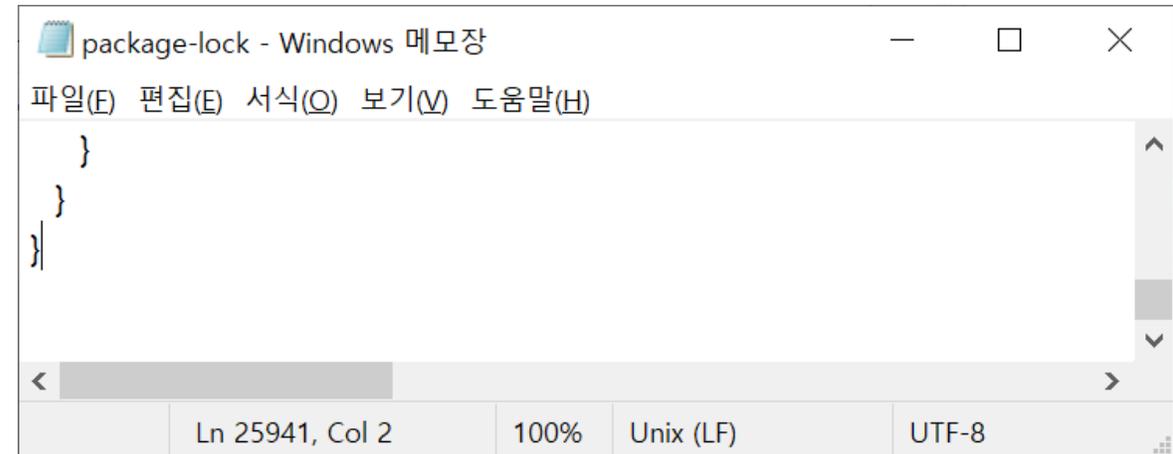
Package.json

Gatsby 하나만 dependency로 지정됨

```
{
  "name": "my-proj",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "gatsby": "^4.24.4"
  }
}
```

Package-lock.json

25000줄이 넘는 거대한 파일



1.1 외부 의존성, 그리고...

의존성 트리의 결정성 (determinism of dependency tree)

한줄 요약 : 설치 순서에 따라서 node_modules 의 형태가 달라질 수 있음!!

개발자의 기기와 해당 라이브러리가 배포된 기기의 node_modules가 달라지고, 설치된 라이브러리의 버전이 달라질 수 있음

Package.json을 보면 알겠지만, package.json은 abc 순으로 정렬되지만, 개발자의 설치 순서는 다를 수 있다.

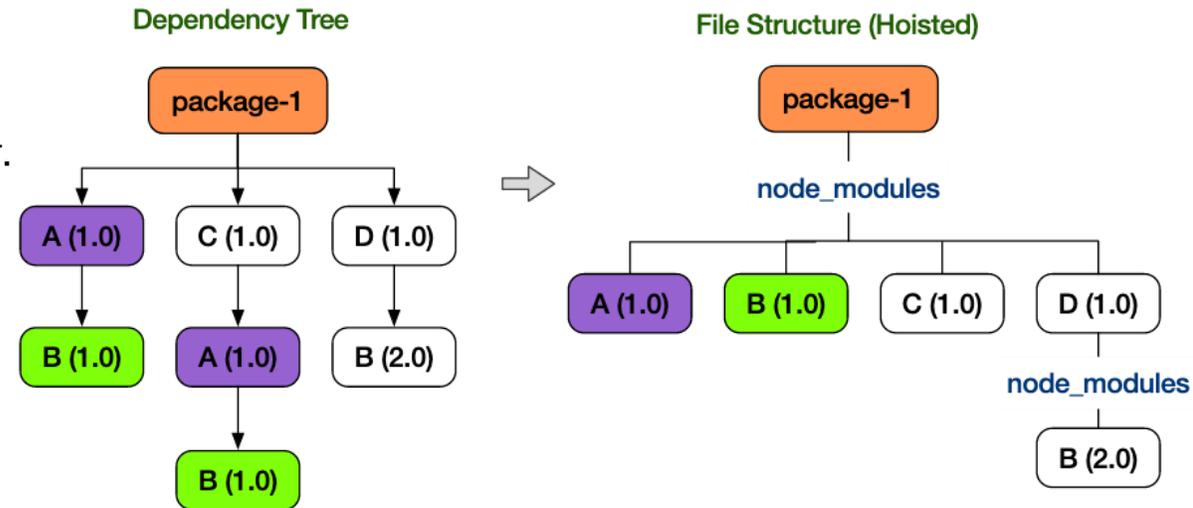
예시를 통해서 알아보자!

1.1 외부 의존성, 그리고...

끌어올림 (hoist)

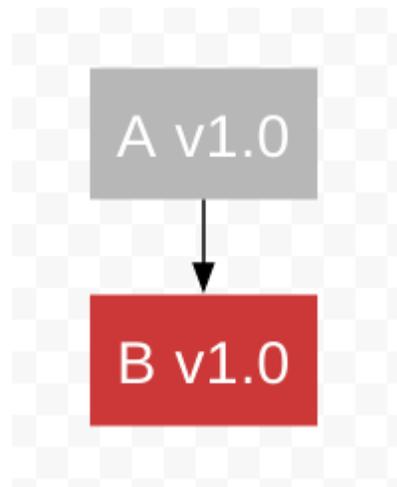
Node에서 `import`나 `require`로 패키지를 불러오면 파일 I/O 연산을 한다.
I/O 연산은 상당히 시간이 많이 걸리고 비효율적임

I/O 연산을 줄이기 위해 디렉토리의 깊이를 npm에서는 줄이려고 함
의존성 트리의 깊이를 줄이기 위한 방법론이 끌어올림 (hoist)

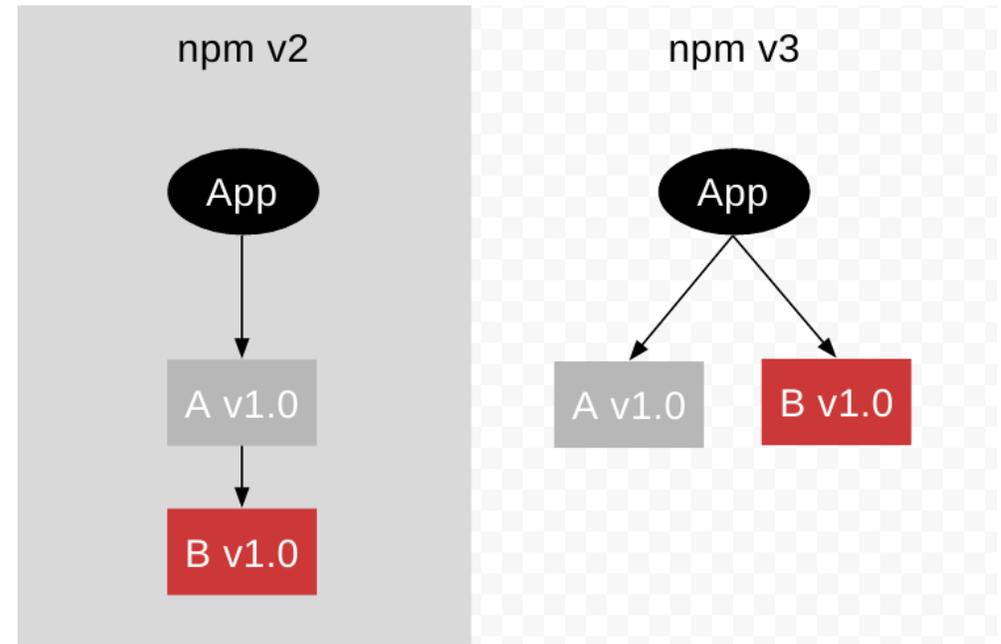


1.1 외부 의존성, 그리고...

1. A 1.0 설치

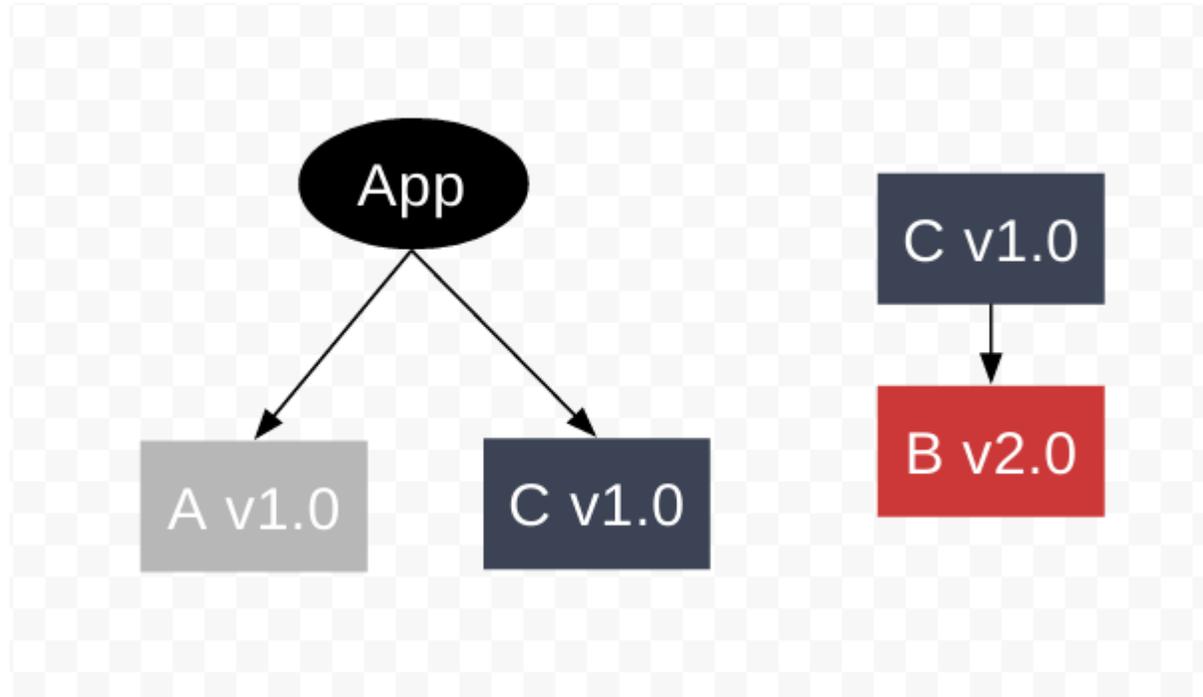


Npm의 끌어올림(hoist)



1.1 외부 의존성, 그리고...

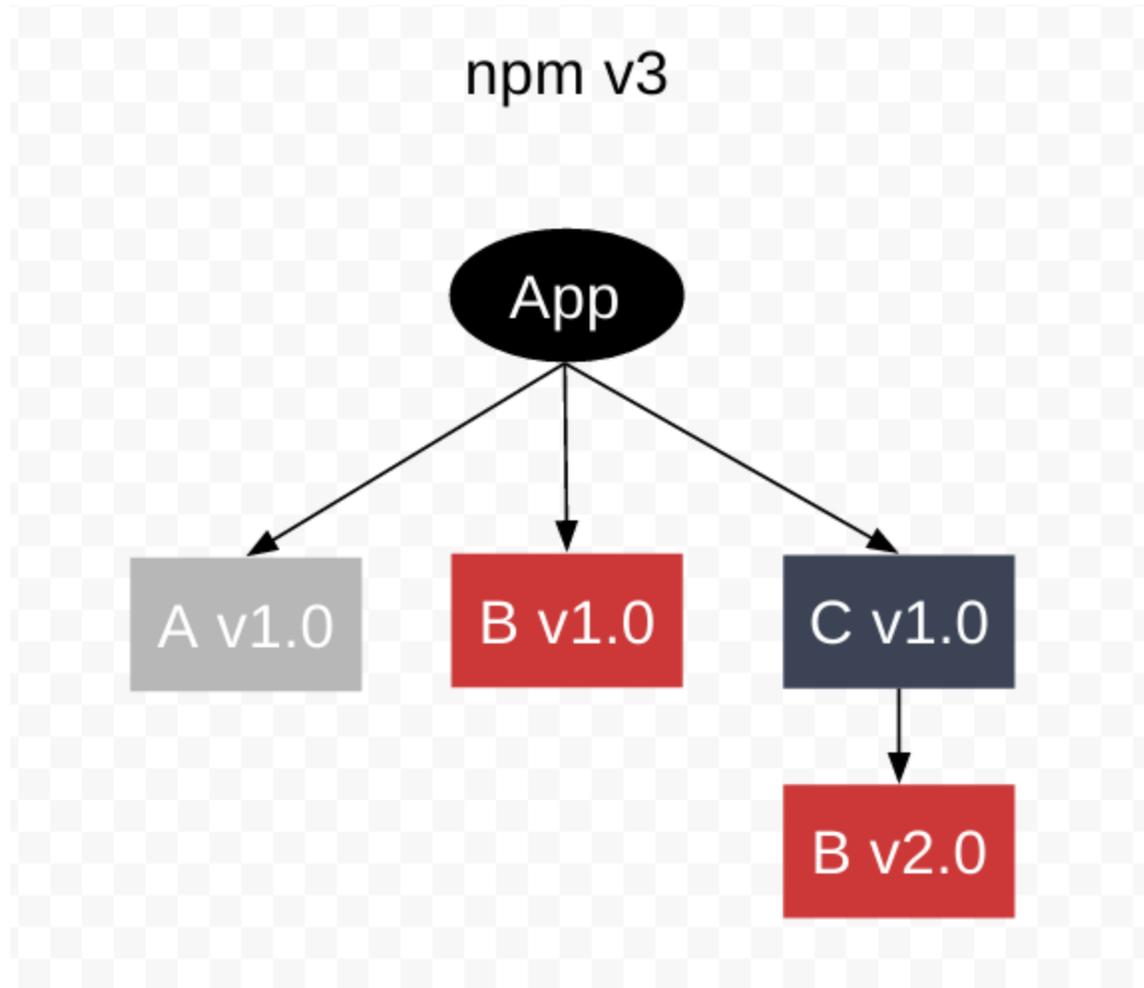
2. C 1.0 설치



C는 B 2.0에 의존적

1.1 외부 의존성, 그리고...

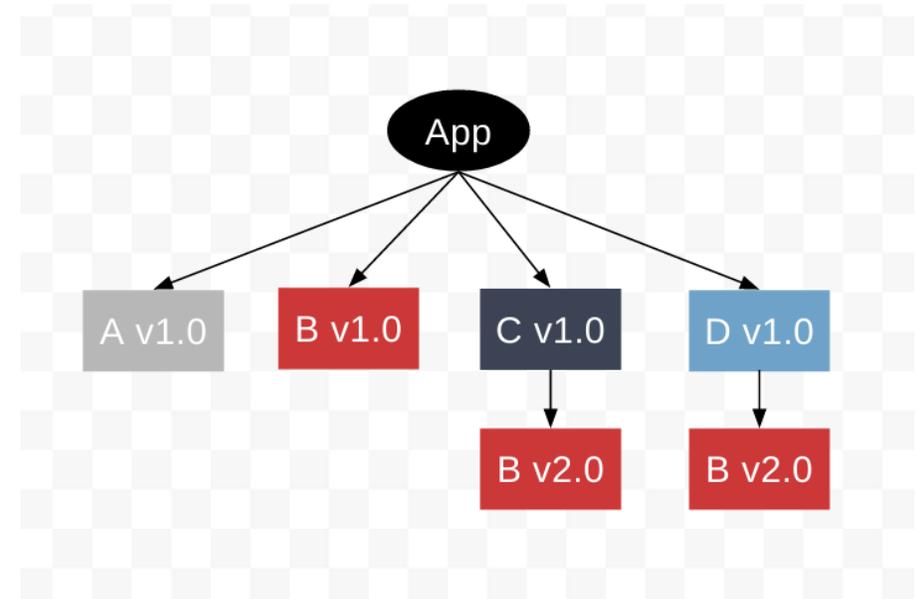
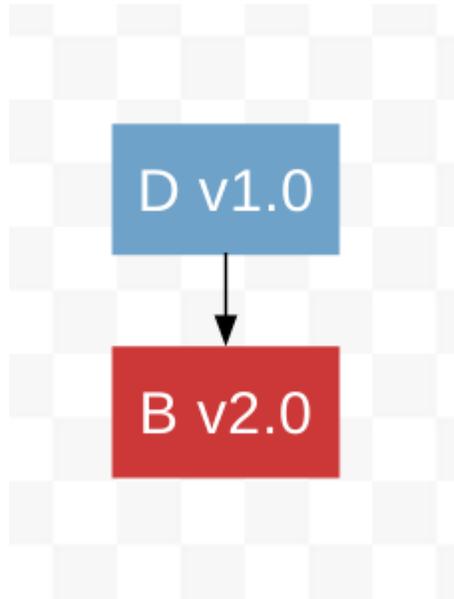
2. C 1.0 설치



B 1.0이
최상위에 있으므로
B 2.0은 C의 하위에
들어감

1.1 외부 의존성, 그리고...

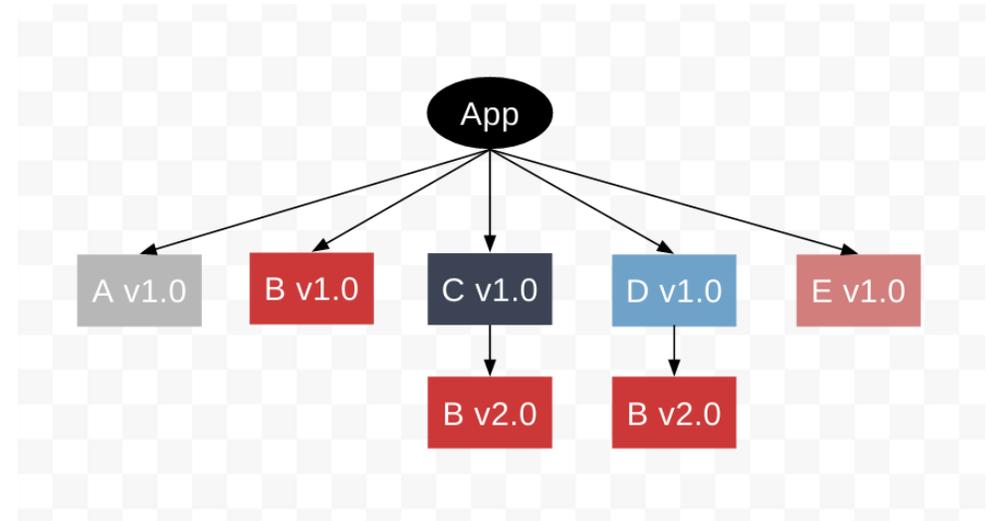
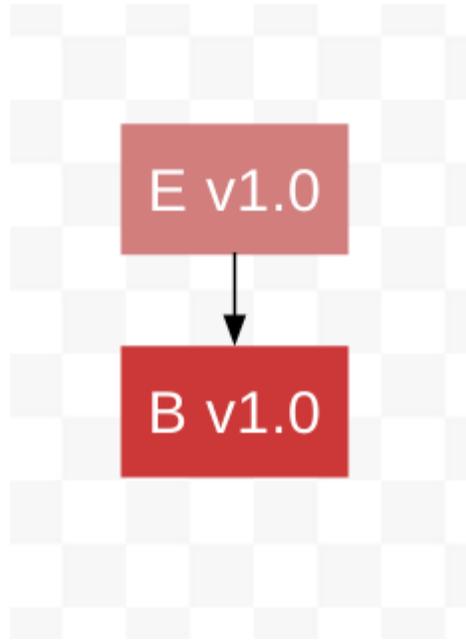
3. D 1.0 설치



B 1.0이 최상위에 있기에
D 밑에 B 2.0을 중복되게 설치해야함

1.1 외부 의존성, 그리고...

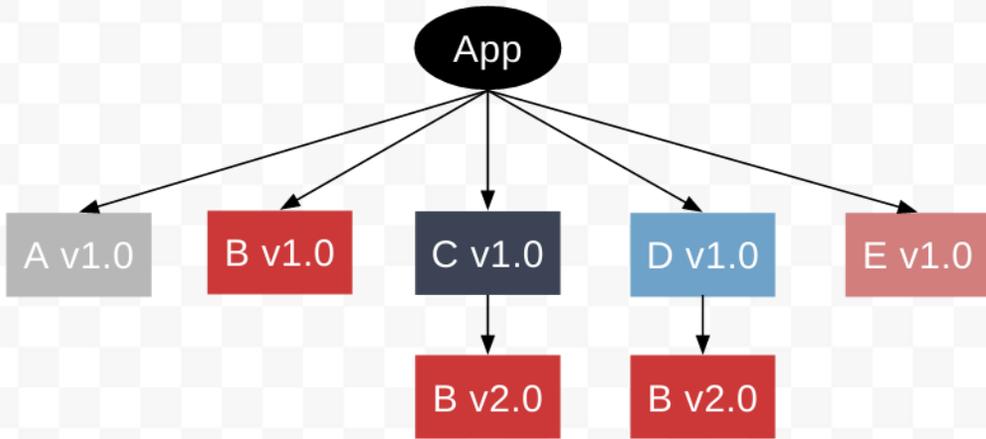
4. E 1.0 설치



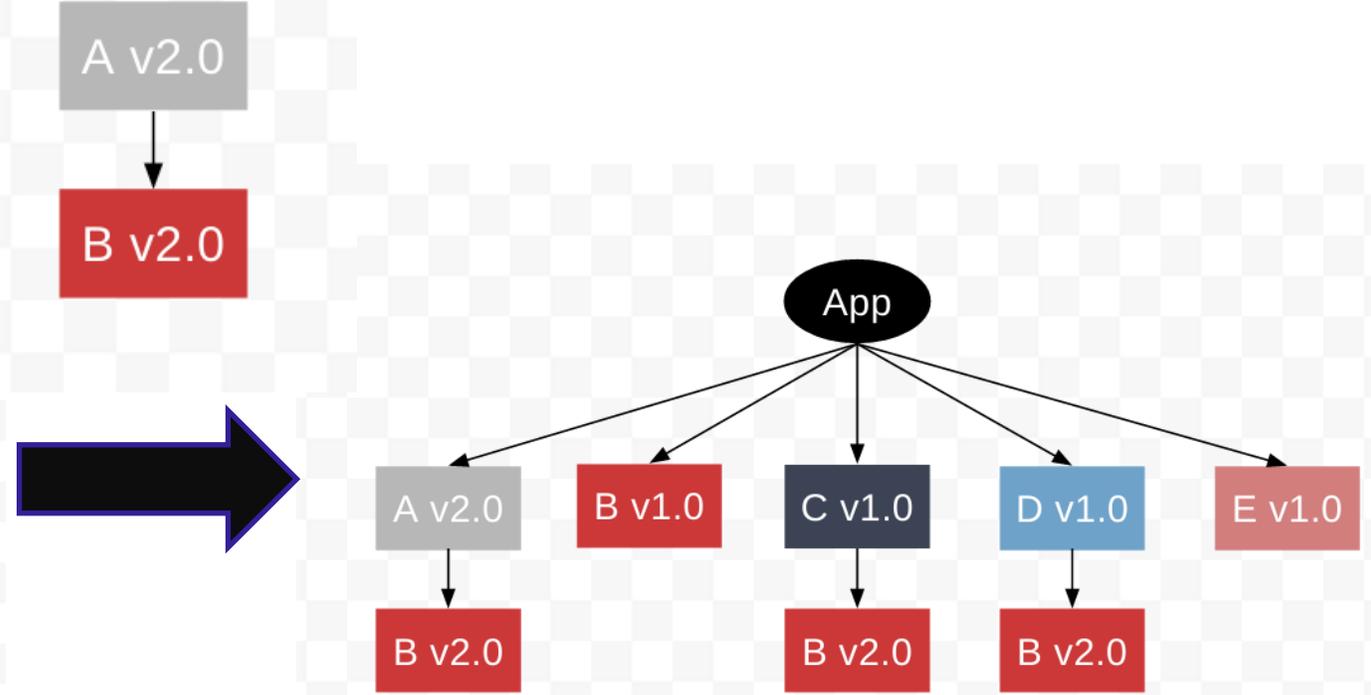
B 1.0이 최상위에 있기에
E는 따로 B에 대한 작업 필요 X

의존성 트리의 결정성

의존성 A 2.0으로 교체
A 2.0은 B 2.0에 의존



기존의 dependency tree

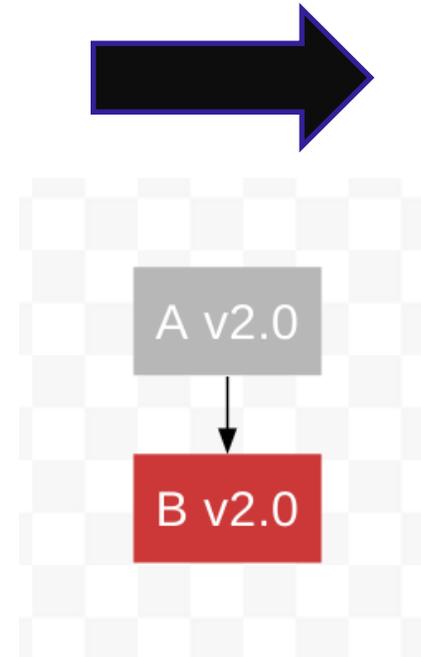


새로운 dependency tree

1.1 외부 의존성, 그리고...

```
{
  "name": "example3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mod-a": "^1.0.0",
    "mod-c": "^1.0.0",
    "mod-d": "^1.0.0",
    "mod-e": "^1.0.0"
  }
}
```

기존의 package.json



의존성 A 2.0으로 교체
A 2.0은 B 2.0에 의존

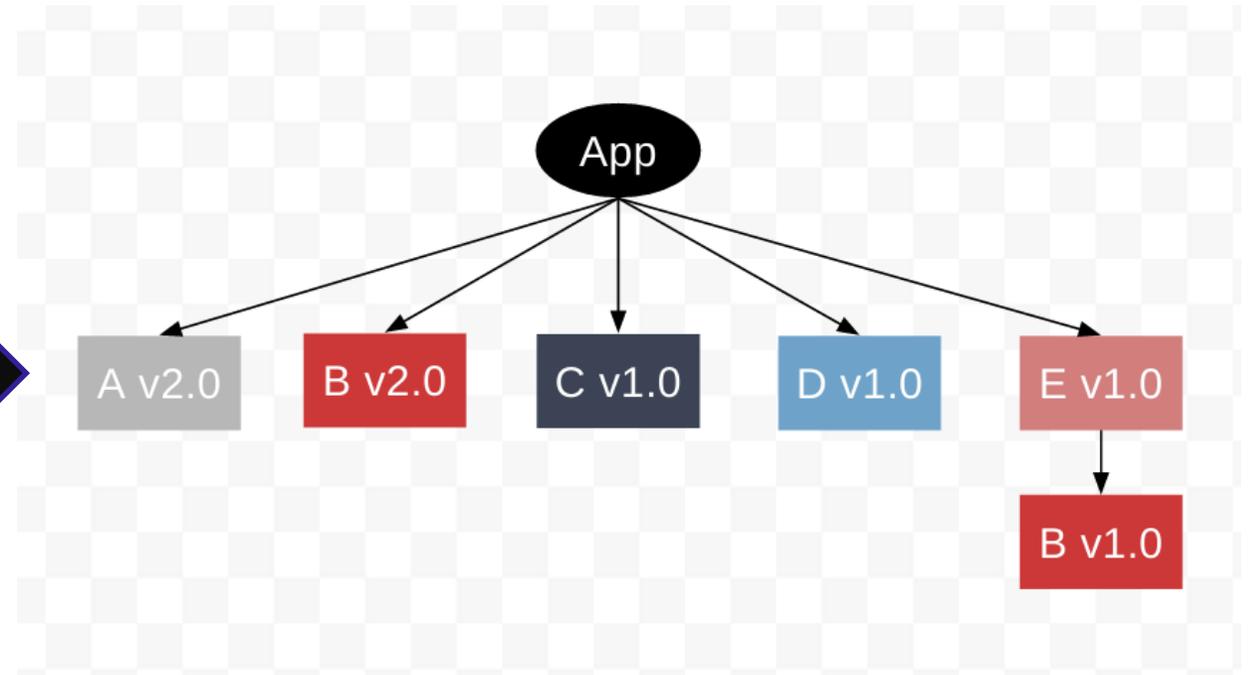
```
{
  "name": "example3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mod-a": "^2.0.0",
    "mod-c": "^1.0.0",
    "mod-d": "^1.0.0",
    "mod-e": "^1.0.0"
  }
}
```

새로운 package.json

1.1 외부 의존성, 그리고...

```
{
  "name": "example3",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "mod-a": "^2.0.0",
    "mod-c": "^1.0.0",
    "mod-d": "^1.0.0",
    "mod-e": "^1.0.0"
  }
}
```

Package.json



배포 서버에 설치되는 dependency tree

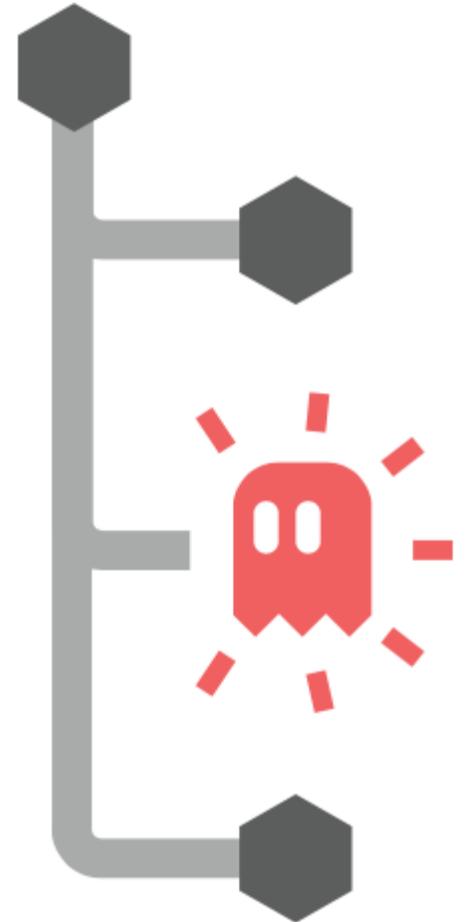
1.2 npm이 작동하는 방식

유령 의존성 (Ghost dependency)

한줄 요약 : 설치도 안한 라이브러리가 import 될 수 있는것

아까 살펴봤듯, 호이스팅을 통해서 명시적으로 설치하겠다 하지도 않은 라이브러리가 최상위에 있다
너무나도 당연하게도 이 친구들을 의존하는 라이브러리가 없어지면 호이스팅 된 라이브러리는 날라갈것

생각치도 못한 동작을 일으키는 이것은 문제라고 할 수 있다.



1.2 npm이 작동하는 방식

의존성 트리의 결정성

(determinism of dependency tree)

문제가 되는 이유 : CI/CD의 기준이 dependency의 해시값에 따라 달라질 수 있다.
치명적인 문제가 아니라고는 하지만, DX를 끌어올리기 위해서는 해결해야...

1.2 npm이 작동하는 방식

파일 I/O 기반

1. 비효율적인 의존성 검색
2. 비효율적인 설치 (세상에서 가장 무거운 `node_modules`)



1.2 npm이 작동하는 방식

비효율적인 의존성 검색

Npm은 node.js를 이용해서 file I/O로 resolve
`require.resolve.paths()`를 이용해서 확인 가능

```
$ node
Welcome to Node.js v12.16.3.
Type ".help" for more information.
> require.resolve.paths('react')
[
  '/Users/toss/dev/toss-frontend-
libraries/repl/node_modules',
  '/Users/toss/dev/toss-frontend-libraries/node_modules',
  '/Users/toss/node_modules',
  '/Users/node_modules',
  '/node_modules',
  '/Users/toss/.node_modules',
  '/Users/toss/.node_libraries',
  '/Users/toss/.nvm/versions/node/v12.16.3/lib/node',
  '/Users/toss/.node_modules',
  '/Users/toss/.node_libraries',
  '/Users/toss/.nvm/versions/node/v12.16.3/lib/node'
]
```

React 하나 찾으려고 대체 얼마를 찾는데...

1.2 npm이 작동하는 방식

비효율적인 의존성 검색

Node.js가 의존성을 찾는 방식

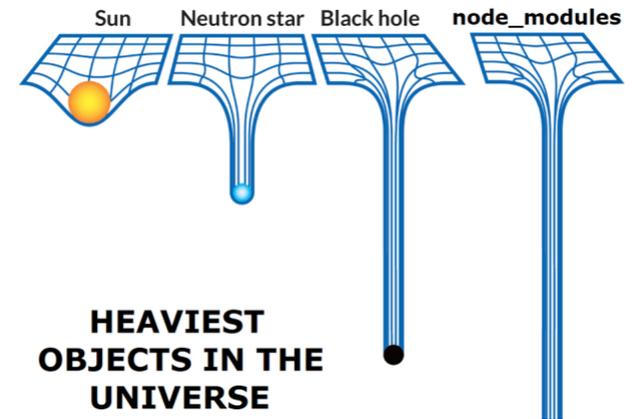
1. nodeJS 내장 코어 모듈인지 확인
2. 코어 모듈이 아니라면, 현재 디렉토리부터 `node_modules` 디렉토리를 탐색해간다.
 - 현재 디렉토리에서 찾지 못하면 계층적으로 상위에 있는 디렉토리에서 탐색한다.
3. 그래도 찾지 못했다면 환경변수로 지정된 `global folder`에서 탐색
4. 그래도 못찾으면 `Error:Cannot find module 'yourfile'`

1.2 npm이 작동하는 방식

세상에서 제일 무겁고 복잡한 node_modules

간단한 CLI 프로젝트에서도 많은 용량

Node_modules 폴더는 복잡하기 때문에
유효성만 검사, 내용 올바른지 검사 X



1.3 Npm의 경쟁자



Yarn
(2016 ~ 현재)



pnpm
(2017 ~ 현재)

1.3 npm의 경쟁자



Yarn

(2016 ~ 현재)

2016년에 등장한 새로운 패키지 매너지

- 보안, 일관성 이슈를 해결하기 위해 등장
- Native 모노레포 지원
- 새로운 lock 파일 형식
- 새로운 버전 yarn berry 등장

1.3 npm의 경쟁자



pnpm
(2017 ~ 현재)

2017년에 등장한 새로운 패키지 매너지

- Performant npm의 줄임말
- 플랫폼하지 않은 node_modules 구조
- 하드링크를 사용해 하드디스크 용량 절약
- 심볼릭 링크를 통한 빠른 탐색

CHAPTER 2.

Yarn 그리고 Yarn berry

원래 이걸 주제를 할려고 했다가 어쩌다 보니 서브 섹션이 되었네요

2.1 Yarn berry의 main concept



Yarn berry
(2020 ~ 현재)

2020년 1월에 등장한 yarn의 차기 버전

- Yarn classic에 호환되지 않음
- Plug n play & Zipfs를 이용한 의존성 관리
- 플러그인 친화적
- Monorepo를 위한 workspace 기능 강화

2.1 기존의 node_modules는 가라! Plug n Play

```
.
├─ .yarn/
│  ├─ cache/
│  ├─ releases/
│  │  └─ yarn-3.1.1.cjs
│  └─ sdk/
├─ └─ unplugged/
├─ .pnp.cjs
├─ .pnp.loader.mjs
├─ .yarnrc.yml
├─ package.json
└─ yarn.lock
```

기본값인 pnp 설정인 상태로 세팅한 예시

- Node_modules가 존재 하지 않음
- .yarn/cache 안에 zip 파일로 의존성이 저장
- .pnp.cjs 파일 안에 의존성을 확인할 수 있게 기록

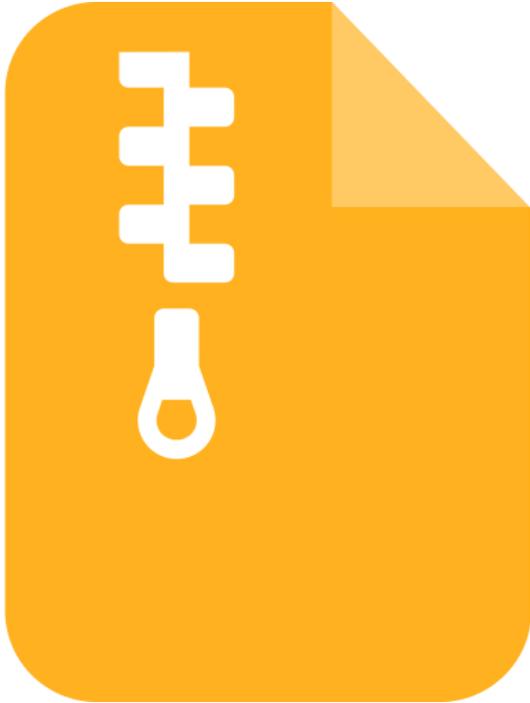
2.1 기존의 node_modules는 가라! Plug n Play

```
.
├─ .yarn/
│  ├─ cache/
│  ├─ releases/
│  │  └─ yarn-3.1.1.cjs
│  └─ sdk/
├─ unplugged/
├─ .pnp.cjs
├─ .pnp.loader.mjs
├─ .yarnrc.yml
├─ package.json
└─ yarn.lock
```

```
/* react 패키지 중에서 */
["react", [
  /* npm:17.0.1 버전은 */
  ["npm:17.0.1", {
    /* 이 위치에 있고 */
    "packageLocation": "./.yarn/cache/react-npm-17.0.1-98658812fc-a76d86ec97.zip/node_modules/react/",
    /* 이 의존성들을 참조한다. */
    "packageDependencies": [
      ["loose-envify", "npm:1.4.0"],
      ["object-assign", "npm:4.1.1"]
    ],
  }],
]]
```

.pnp.cjs 파일의 예시. 의존성의 위치를 바로 알아내어 I/O 연산의 수가 획기적으로 줄어든다

2.1 기존의 node_modules는 가라! Plug n Play



1. node_modules 구조 생성이 필요 없기 때문에 빠른 설치
2. 각 패키지는 버전마다 하나의 파일만 가짐
-> 중복되어 설치되지 않음 -> 용량 절약!
3. 의존성 구성 파일의 개수가 적기 때문에 변경사항 추적 용이
4. 아예 의존성 자체를 git에 올리는 zero install 전략 가능
-> CI/CD 때 패키지 설치 시간 X -> 빠른 CI/CD!

CHAPTER 3.

Performant npm. pnpm!

Yarn 과는 다른 전략을 택한 패키지 매니저
pnpm을 한번 알아봅시다

3.1 pnpm 의 main concept

1. 디스크 용량 절약 + 설치 속도 향상

- 한 의존성에 의존하는 프로젝트가 100개 있으면, 사본도 100개 있어야 해?
 - 그냥 전역적으로 한곳에 설치 해두고, 하드 링크를 두자!
- 라이브러리 업데이트가 한 파일만 수정해도 다 복사를 해야할까?
 - 변경사항만 기록해두지 뭐

2. Flat 하지 않은 node_modules

- Hoisting을 통한 패키지 관리는 저런 문제가 있던데
 - 우린 굳이 hoisting 안하고 심볼릭 링크로 flat하지 않게 관리해버리자

express 하나만 설치한 프로젝트의 node_modules 구조

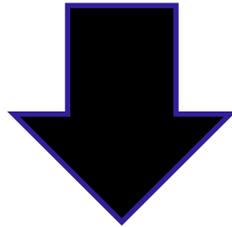
```
node_module/  
├── .bin  
├── accepts  
├── array-flatten  
├── body-parser  
├── bytes  
├── content-disposition  
├── cookie-signature  
├── cookie  
├── debug  
├── depd  
├── destroy  
├── ee-first  
├── encodeurl  
├── escape-html  
├── etag  
└── express
```

```
node_module/  
├── .pnpm  
├── .modules.yaml  
└── express
```



3.2 굳이 hoisting을 해야할까? Symlinked node_modules

사실 너무나도 단순한 아이디어
node_modules 에는 내가 명시한 의존성만 있으면 참 좋겠다.



그러면 node_modules에는 내가 명시한 의존성만 넣고
의존성의 의존성은 다른데서 관리해주자

3.2 굳이 hoisting을 해야할까? Symlinked node_modules

```
node_modules
├── foo -> ../.pnpm/foo@1.0.0/node_modules/foo
└── .pnpm
    ├── bar@1.0.0
    │   ├── node_modules
    │   └── bar -> <store>/bar
    └── foo@1.0.0
        ├── node_modules
        │   ├── foo -> <store>/foo
        │   └── bar -> ../../bar@1.0.0/node_modules/bar
```

간략하게 나타낸 pnpm에서의 node_modules 구조

3.2 굳이 hoisting을 해야할까? Symlinked node_modules

```
node_modules/
├── .pnpm/
│   ├── accepts@1.3.5
│   ├── array-flatten@1.1.1
│   ├── ...
│   └── express@4.16.3/
│       └── node_modules/
│           ├── accepts
│           ├── array-flatten
│           ├── body-parser
│           ├── content-disposition
│           ├── ...
│           ├── etag
│           └── express/
│               └── lib/
│                   ├── History.md
│                   ├── index.js
│                   ├── LICENSE
│                   ├── package.json
│                   └── Readme.md
```

실제 express가 설치된 node_modules의 일부

성능 비교

<https://pnpm.io/benchmarks>

결론



오랜 역사
많은 문서 + 검색결과

충분히 쓸만한 성능



혁신적인 pnp

많은 기업들에서 사용중



컴팩트한 디스크 사용

빠른 설치 속도

End of Document
Thank You.