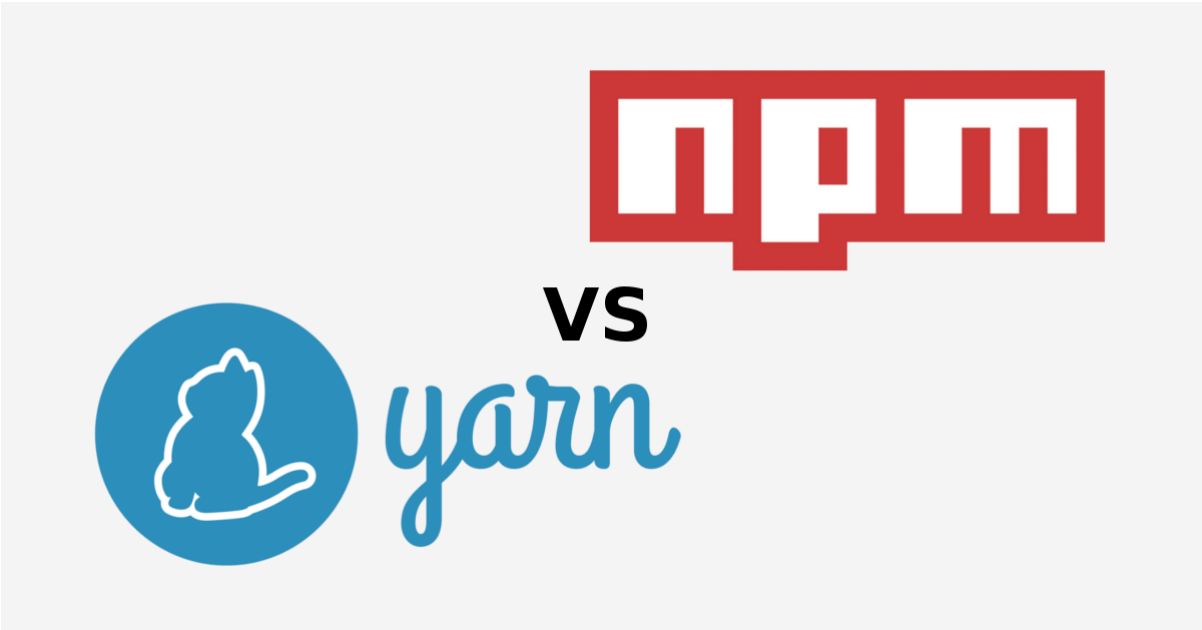
[FE] 패키지 매니저(npm, yarn, yarn berry)



우리가 자주 사용하는 단짝친구 **npm**과 **yarn**!

오늘은 **패키지 매니저**에 대해 알아보겠습니다..!!

1. 패키지 매니저란?

> 패키지 매니저는 앱이나 소프트웨어에서 사용되는
>
> **패키지(라이브러리, 모듈 등)들을 관리하는 도구**

패키지 매니저를 사용하면 프로젝트에 필요한 **외부 라이브러리를 쉽게 설치하고, 버전 관리**를 할 수 있습니다.

이때, 패키지란?

👉 **코드의 배포를 위해서 사용되는 코드의 묶음**을 뜻합니다.

패키지는 라이브러리와 비슷한 개념입니다.

라이브러리는 코드 작성을 위해 필요한 코드 모음이며, 패키지는 코드의 배포를 위해 사용되는 코드 모음을 뜻합니다.

패키지는 일반적으로 다음을 포함합니다.

1. 라이브러리
2. 실행파일
3. .bin(컴파일한 binary)
4. congiguration(환경설정 정보)
5. dependency(의존성 정보)

Node.js의 **npm,yarn**, Python의 **pip**, Java의 **Maven, Gradle** 등 많은 프로그래밍 언어들은 각각 자신만의 패키지 매니저와 software repository(패키지를 저장하고 관리하는 저장소)를 가지고 있습니다.

저는 프론트엔드이기 때문에 npm과 yarn 그리고 yarn berry(yarn pnp)에 대해 알아보도록 하겠습니다.

2. npm은 문제덩어리?

npm은 **Node Package Manager**의 약자입니다.

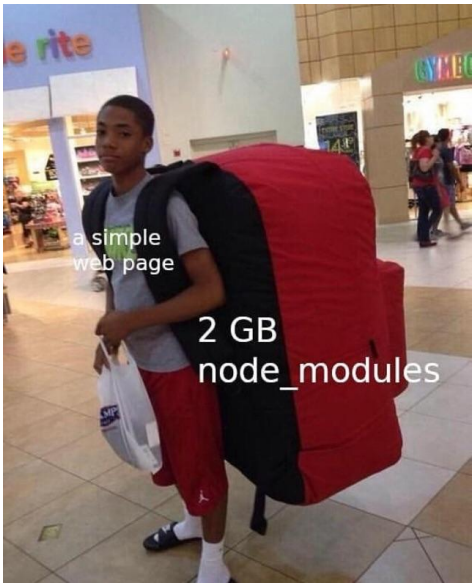
Node.js에서 사용하는 모듈들을 배포하고 설치할 수 있게 해주는 도구입니다. 간단히 얘기하면 구글의 'Play Store' 혹은 아이폰의 'App Store'과 같이 봐도 무방합니다.

npm은 node.js를 설치하면 자동으로 설치되는 패키지 매니저로 가장 범용적으로 사용이 되고 있습니다.

그만큼 커뮤니티도 크고, 오류 해결에 대한 정보도 많습니다.

npm은 2010년 첫 발표 이후 Javascript 생태계에 많은 변화를 가져왔지만, 그 구조적 한계 때문에 여러 문제점들을 가지고 있었습니다.

비효율적인 설치



크고 무거운 우리 노드모듈스^^

npm에서 구성하는 **node_modules 디렉토리 구조는 매우 큰 공간을 차지**합니다.

일반적으로 간단한 CLI 프로젝트도 수백 메가바이트의 node_modules 폴더가 필요합니다. 용량만 많이 차지할 뿐 아니라, 큰 node_modules 디렉토리 구조를 만들기 위해서는 많은 I/O 작업이 필요합니다.

node_modules 폴더는 복잡하기 때문에 설치가 유효한지 검증하기 어렵습니다.

예를 들어, 수백 개의 패키지가 서로를 의존하는 복잡한 의존성 트리에서 node_modules 디렉토리 구조는 깊어집니다. 이렇게 깊은 트리 구조에서 의존성이 잘 설치되어 있는지 검증하려면 많은 수의 I/O 호출이 필요합니다.

일관적이지 않은 패키지 버전

많은 사람들이 인식하지 못하는 문재점 중 하나가 **시멘틱 버저닝**이라는 기법입니다.

시멘틱 버저닝이란 간단히 말해 `1.2.3` 처럼 버전을 세 가지 숫자가 들어갈 수 있는 자리로 구분하고, 각각의 자리에 현재 버전이 이전 버전과 어떤 관계가 있는지 암시하도록 하는 방법입니다.

```
"dependencies": {
  "@heroicons/react": "^2.0.17",
  "@material-tailwind/react": "^1.4.2",
  "@stomp/stompjs": "^7.0.0",
  "@types/node": "^16.18.24",
  "@types/react": "^18.0.38",
  "@types/react-dom": "^18.0.11",
  "@types/react-router-dom": "^5.3.3",
```

위 사진을 보면 모듈의 버전에 캐럿기호(`^`)가 들어가 있는데, 이는 가장 앞 숫자인 **메이저 버전을 제외한 두 자리(마이너, 패치) 버전까지는 변경을 허용**할 수 있다는 의미입니다.

현재 "@types/node"의 버전이 `^16.18.24` 라는 것은 메이저 버전인 16이 변경되지 않은 범위에서는 모듈의 버전이 `16.18.23`, `16.19.0` 버전 등이 모두 사용될 수 있다고 볼 수 있습니다.

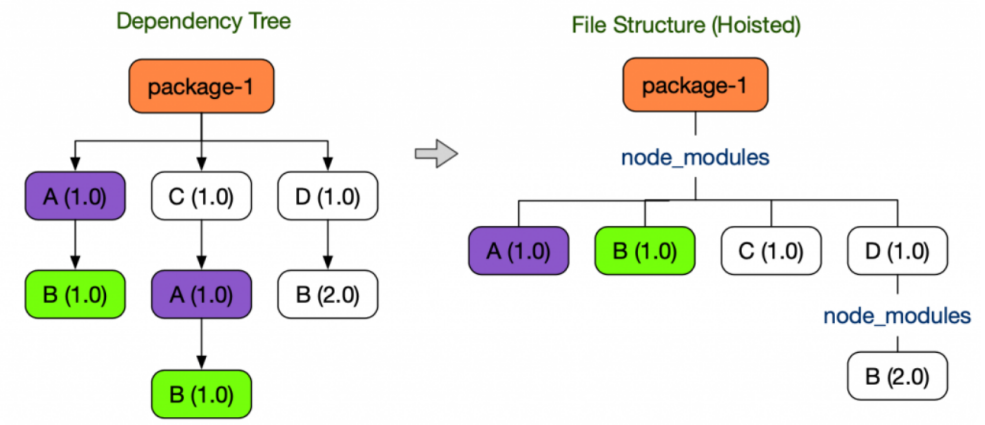
이때 `npm install` 명령어를 통해 모듈들을 설치하면 해당 메이저 버전 중 최신 버전을 다운받게 됩니다.

빌드에 따라 유연하게 버전을 선택할 수 있다는 장점도 있지만, 모듈간의 버전 불일치로 인해 문제가 발생할 수도 있습니다.

해당 상황을 재현하기 까다로워지면 디버깅하는데 많은 시간소요가 필요하므로 환경에 따라 동작이 변하는 것은 나쁜 징조입니다.

유령 의존성 (Phantom Dependency)

npm 및 yarn v1에서는 **중복해서 설치되는 node_modules를 아끼기 위해 끌어올리기(Hoisting) 기법을 사용**합니다.



예를 들어, 의존성 트리가 왼쪽 모습을 하고 있다고 가정을 합시다.

왼쪽 트리에서 `A(1.0)`과 `B(1.0)` 패키지는 두 번 설치되므로 디스크 공간을 낭비합니다. 그렇기 때문에 npm과 yarn v1에서는 디스크 공간을 아끼기 위해 원래 트리의 모양을 오른쪽 트리처럼 바꿉니다.

오른쪽 트리로 의존성 트리가 바뀌면서 package-1 에서는 원래 `require()` 할 수 없었던 `B(1.0)` 라이브러리를 불러올 수 있게 됩니다.

이렇게 **끌어올리기에 따라 직접 의존하고 있지 않은 라이브러리를 `require()` 할 수 있는 현상을 유령 의존성**이라고 부릅니다.

유령 의존성 현상이 발생할 때, `package.json`에 명시하지 않은 라이브러리를 조용히 사용할 수 있게 됩니다. 다른 의존성을 `package.json`에서 제거했을 때 소리없이 같이 사라지기도 합니다.

이런 특성은 의존성 관리 시스템을 혼란스럽게 만듭니다.

고정되지 않은 설치 순서

개발자의 환경에 따라 모듈들의 설치 순서가 변경될 수 있습니다.

npm은 모듈 이름을 사전 순서대로 정렬하여 순차적으로 설치하기 때문에, 내가 어떤 순서로 모듈을 의존성에 추가했는지에 상관없이 설치되곤합니다.

순차적인 설치로 인한 긴 소요 시간

npm은 모듈들을 한 번에 하나씩만 순차적으로 설치합니다.

설치해야하는 모듈이 많으면 많을수록, 총 설치 시간이 길어지게 됩니다.

첫 모듈 설치 시간이 길어지면 빌드 및 배포 시간에도 부정적인 영향을 끼치게 됩니다.

3. yarn은 npm의 단점을 보완하고자 등장했다

npm에서 패키지들을 node_modules에 적용하며 발생하는 유령 의존성, 프로젝트가 커질수록 속도와 성능 저하를 동반하는 과한 I/O를 유발하는 구조적 문제들 때문에 yarn이 혜성처럼 등장했습니다.

🐱 yarn

yarn은 npm의 동작방식과 비슷합니다.

메타(구 페이스북)에서 **npm의 단점을 보완하여 성능과 속도를 개선한 라이브러리 관리 도구**입니다. ~~(이정도면 메타는 신이 아닐까?)~~

크게 3가지의 컨셉을 밀고있습니다.

1. **고속** : 다운로드한 모든 패키지를 캐시하므로 다시 다운로드할 필요가 없다.
2. **보안** : 체크섬을 사용하여 코드가 실행되기 전에 설치된 모든 패키지의 무결성을 확인한다.
3. **신뢰성** : 시스템에서 작동하는 설치가 다른 시스템에서 정확히 동일한 방식으로 작동하도록 보장한다.

하지만 yarn은 `1.22` 버전을 마지막으로 Classic 버전의 관리를 중단했습니다. `2.X` 버전 이후부터는 Yarn Modern(berry)이라 부르고 새로운 구조로 패키지를 관리합니다.

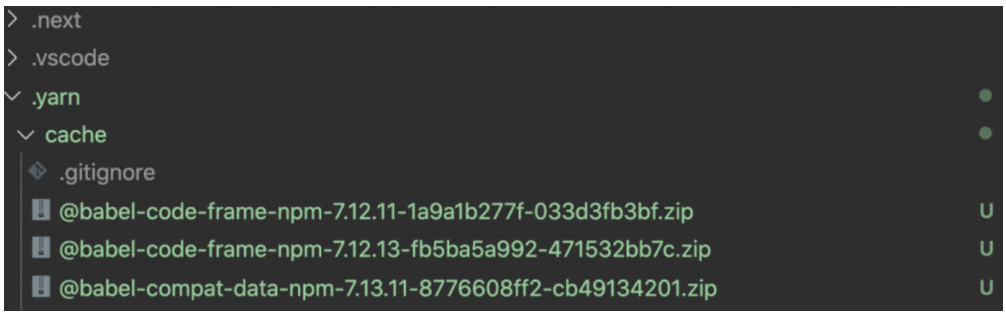
즉, 구 버전을 yarn 이라 부르고, `2.X` 버전 이후 부터는 yarn berry라 부릅니다.

4. node_modules로부터 우리를 구원해 줄 Yarn Berry

yarn berry는 npm과 yarn v1의 다양한 문제점들을 해결할 수 있습니다.

yarn berry는 비효율적인 의존성 검색, 비효율적인 설치, 유령 의존성 등의 문제점을 **PnP(Plug n Play) 방식으로 개선한 새로운 패키지 관리 시스템**입니다.

Plug'n'Play(PnP)



yarn berry는 node_modules를 생성하지 않습니다.

대신 위와 같은 독특한 의존성 모습을 볼 수 있습니다.

- `.zip` : 패키지에 대한 정보를 zip파일로 압축
- `.yarn/cache` : 압축된 정보들이 저장되는 위치
- `.pnp.cjs` : 저장된 정보를 찾기 위한 정보를 기록

`.pnp.cjs` 를 이용하면 별도의 I/O 작업 없이도 정확한 패키지 위치를 알 수 있습니다.

이로 인해,

1. **시간 단축**
2. **중복 설치 방지**
3. **유령 의존성 문제 해결**
4. **패키지 압축으로 인한 용량 감소**

위와 같은 장점을 얻을 수 있습니다.

Zero install

yarn berry는 PnP 전략으로 node_modules를 설치하지 않기 때문에 의존성까지 github에 올릴 수 있습니다.

(더 이상 .gitignore 하지 않아도 됩니다😎)

github는 파일당 최대 용량을 500mb으로 제한하고, 원활한 이용을 위해 저장소당 1gb 미만의 크기를 유지할 것을 권장하고 있습니다.

yarn berry를 통해 만든 의존성 폴더는 웬만하면 200mb를 넘지 않습니다.

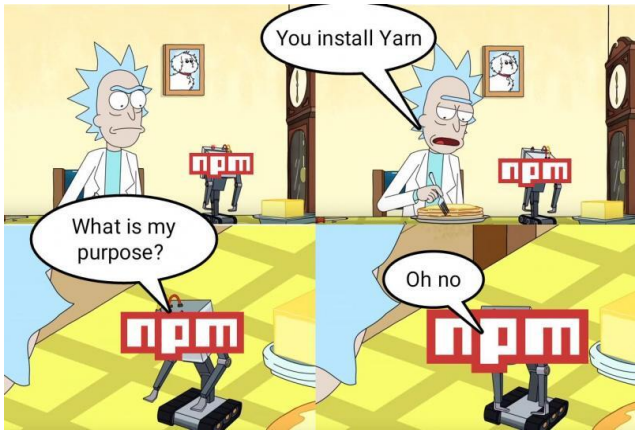
덕분에 `git clone` 이후 **별도의 설치가 필요 없이, 바로 사용할 수 있도록하는 zero-intall**을 시도해 볼 수 있습니다.

(npm i 야 안녕~ 👋👋👋)

zero-install을 사용할 경우 로컬에서의 귀찮음도 줄어들지만, **CI/CD 파이프라인을 구축한 경우 더 큰 효과**를 볼 수 있습니다.

클론이 끝나자마자 곧바로 빌드가 가능해진 덕분에 배포까지 걸리는 속도가 대폭 단축됩니다.

5. 결론



yarn berry의 등장 이후 많은 기업에서 패키지 매니저를 yarn berry을 사용하는 경우가 늘었습니다.

(토스, 배민, 화해, 채널톡 등등)

그렇다면 yarn berry는 무조건 다 좋기만 할까요?

그건 또 아닙니다.

또한, Yarn이 현재까지 v4까지 올라오고 있음에도 여전히 PnP를 지원하지 않는 패키지가 존재합니다.

프로젝트에 이런 패키지가 하나라도 존재한다면 PnP 방식이더라도 node_modules가 따라오게 되어서, 앞서 언급한 장점들을 극한으로 누릴 수는 없게 됩니다.

반면, npm과 yarn classic이 유구한 역사가 있는만큼 다양한 커뮤니티와 라이브러리 및 패키지를 사용할 수 있습니다.

그러니 프로젝트의 규모와 현재의 상황에 빗대어 알맞은 패키지 매니저를 사용하는 것이 좋겠습니다.