패키지 매니저에 대해서 알아보자

타입스크립트 해적단 - 7/20(일)

패키지 매니저란?

패키지 매니저

개발자가 필요한 **라이브러리(의존성)를 편리하게 설치, 관리, 삭제**할 수 있도록 돕는 도구

패키지란?

패키지

하나의 **기능 또는 도구를 담은 코드 묶음**. (코드의 배포를 위해 사용되는 코드의 묶음)

패키지

하나의 **기능 또는 도구를 담은 코드 묶음**. (코드의 배포를 위해 사용되는 코드의 묶음)

About packages

A **package** is a file or directory that is described by a `package.json` file. A package must contain a `package.json` file in order to be published to the npm registry. For more information on creating a `package.json` file, see "Creating a package.json file".

Packages can be unscoped or scoped to a user or organization, and scoped packages can be private or public. For more information, see

- "About scopes"
- "About private packages"
- "Package scope, access level, and visibility"

About package formats

A package is any of the following:

- a) A folder containing a program described by a `package.json` file.
- b) A gzipped tarball containing (a).
- c) A URL that resolves to (b).
- d) A `<name>@<version>` that is published on the registry with (c).
- e) A `<name>@<tag>` that points to (d).
- f) A `<name>` that has a `latest` tag satisfying (e).
- g) A `git` url that, when cloned, results in (a).

패키지

하나의 **기능 또는 도구를 담은 코드 묶음**. (코드의 배포를 위해 사용되는 코드의 묶음)

패키지는 다음의 3가지 정보를 가지고 있는 코드의 배포 단위이다. **보통 하나의 폴더 구조로 이루어지며, 내부에 package.json 파일이 포함**되어 있다.

1. 컴파일한 소프트웨어의 바이너리(binary)
2. 환경 설정(configuration)에 관련된 정보
3. 의존(dependency)에 관련된 정보

패키지

하나의 **기능 또는 도구를 담은 코드 묶음**. (코드의 배포를 위해 사용되는 코드의 묶음)

패키지의 구성 예시

```
my-awesome-package/
├── index.js              # 패키지 기능이 구현된 코드
├── package.json          # 이름, 버전, 의존성, 진입점 등을 정의
├── README.md             # 사용법 설명
├── LICENSE               # 라이선스 정보
```

패키지

하나의 **기능 또는 도구를 담은 코드 묶음**. (코드의 배포를 위해 사용되는 코드의 묶음)

항목	설명
이름과 버전	각각 고유하며, react@18.2.0처럼 식별됨
의존성	다른 패키지들을 내부에서 사용할 수 있음
배포 가능	npm registry 등에 업로드해 누구나 설치 가능
재사용성	프로젝트 여러 곳에서 재활용 가능

패키지 매니저

개발자가 필요한 **라이브러리(의존성)를 편리하게 설치, 관리, 삭제**할 수 있도록 돕는 도구

패키지 매니저

개발자가 필요한 **라이브러리(의존성)를 편리하게 설치, 관리, 삭제**할 수 있도록 돕는 도구

패키지 매니저

개발자가 필요한 **라이브러리(의존성)를 편리하게 설치, 관리, 삭제**할 수 있도록 돕는 도구

왜 생겨났을까?

개발의 복잡도 증가와 코드 재사용의 필요성

AS-IS

개발자가 **필요한 라이브러리를 직접 복사하거나 수동으로 관리**

AS-IS

개발자가 **필요한 라이브러리를 직접 복사하거나 수동으로 관리**

문제점	세부 내용
1. 코드 재사용 문제	· 같은 기능을 여러 프로젝트에서 반복 작성하는 비효율이 발생 · 타인의 라이브러리를 쉽게 사용할 방법 필요
2. 의존성 관리 복잡성	· 서로 다른 버전 요구사항으로 인한 수동 관리의 어려움 · 복잡한 의존성 트리를 자동으로 해결할 도구 필요
3. 환경 일관성 보장	· 개발자마다 다른 패키지 버전으로 인한 호환성 문제 · lockfile을 통한 **동일한 개발 환경 재현** 필요성
4. 글로벌 생태계 구축	· 개발자 커뮤니티의 코드 공유 촉진 · npmjs.com,yarnpkg.com같은 중앙 저장소 등장

AS-IS

JavaScript 표준인 ECMAScript에 따르면, 원래는 정확한 절대 경로나 상대 경로를 통해서만 import 가능

```
import React from '/Users/raon0211/path/to/react/index.js';
import { sum } from '/Users/raon0211/path/to/@koh/utils/index.js'
```

1. 모든 파일에 정확한 버전 관리 가능?
2. 모든 파일에 정확한 경로 설정 가능?

```
import React from '/Users/raon0211/path/to/react/index.js';
import { sum } from '/Users/raon0211/path/to/@koh/utils/index.js'
```

1. 모든 파일에 정확한 버전 관리 가능?
2. 모든 파일에 정확한 경로 설정 가능?

NOB!!

```
import React from '/Users/raon0211/path/to/react/index.js';
import { sum } from '/Users/raon0211/path/to/@koh/utils/index.js'
```

TO-BE

정확한 정보를 소스 코드보다 상위 디렉토리인 package.json 파일에 명시.

```
{
  "dependencies": {
    "react": "^18.2.0" // react는 ≥ 18.2.0, <19 사이의 어떤 버전이든지 쓸 수 있다고 명시
  }
}
```

TO-BE

편안하게 '절대 경로'로 사용 가능

```
import React from 'react';
import { sum } from '@koh/utils';

sum(1, 2, 3);
```

```
const _ = require('lodash');
```

TO-BE

편안하게 '절대 경로'로 사용 가능

```
import React from 'react';
import { sum } from '@koh/utils';

sum(1, 2, 3);
```

```
const _ = require('lodash');
```

패키지 매니저

개발자가 필요한 **라이브러리(의존성)를 편리하게 설치, 관리, 삭제**할 수 있도록 돕는 도구

> 패키지 매니저 주요 기능 요약

기능	설명
의존성 관리	필요한 패키지를 package.json에 명시하고, 그에 따른 의존 패키지들도 자동으로 설치
node_modules 생성	모든 패키지를 실제 파일로 저장하는 폴더 (yarn 2부터는 생략 가능)
Lockfile 생성	yarn.lock, package-lock.json 등을 통해 버전을 고정해 재설치 시 동일 환경 보장
버전 충돌 해결	여러 버전이 필요한 경우 중복 설치 or 공유 설치로 문제 해결
설치 속도 최적화	캐싱, 병렬 설치 등으로 빠르게 의존성 구성

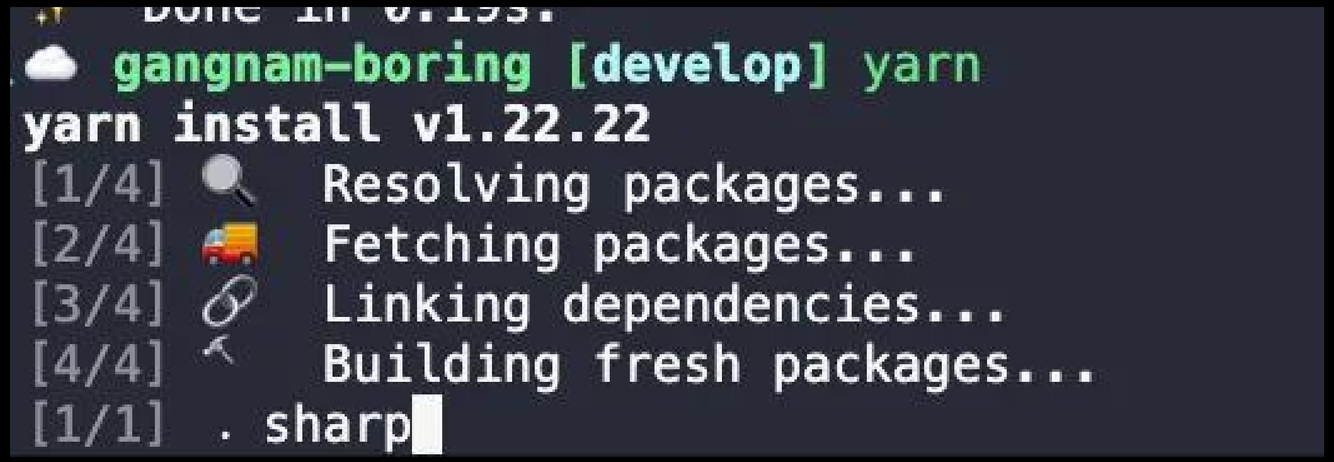
HOW?

패키지 설치 원리

npm install, yarn install, pnpm install 같은 명령어가 실행될 때 내부에서 어떤 일이 일어날까?

패키지 설치 원리

npm install, yarn install, pnpm install 같은 명령어가 실행될 때 내부에서 어떤 일이 일어날까?



패키지 설치 원리

npm install, yarn install, pnpm install 같은 명령어가 실행될 때 내부에서 어떤 일이 일어날까?

Resolution

Fetch

Link

패키지 설치 원리

3단계를 알아보자!!

> 동작 요약

단계	기능	설명
Resolution	의존성 해결	- 라이브러리 버전 고정 - 라이브러리의 다른 의존성 확인 - 라이브러리의 다른 의존성 버전 고정 - 결과물을 lockfile에 저장
Fetch	다운로드	- Resolution 단계에서 결정된 버전의 파일을 다운로드 - 주로 npm 레지스트리에서 패키지 다운로드
Link	연결	- 다운로드된 라이브러리를 소스 코드에서 사용할 수 있도록 연결 - 패키지 매니저별로 다른 방식으로 구현(npm, pnpm, PnP)

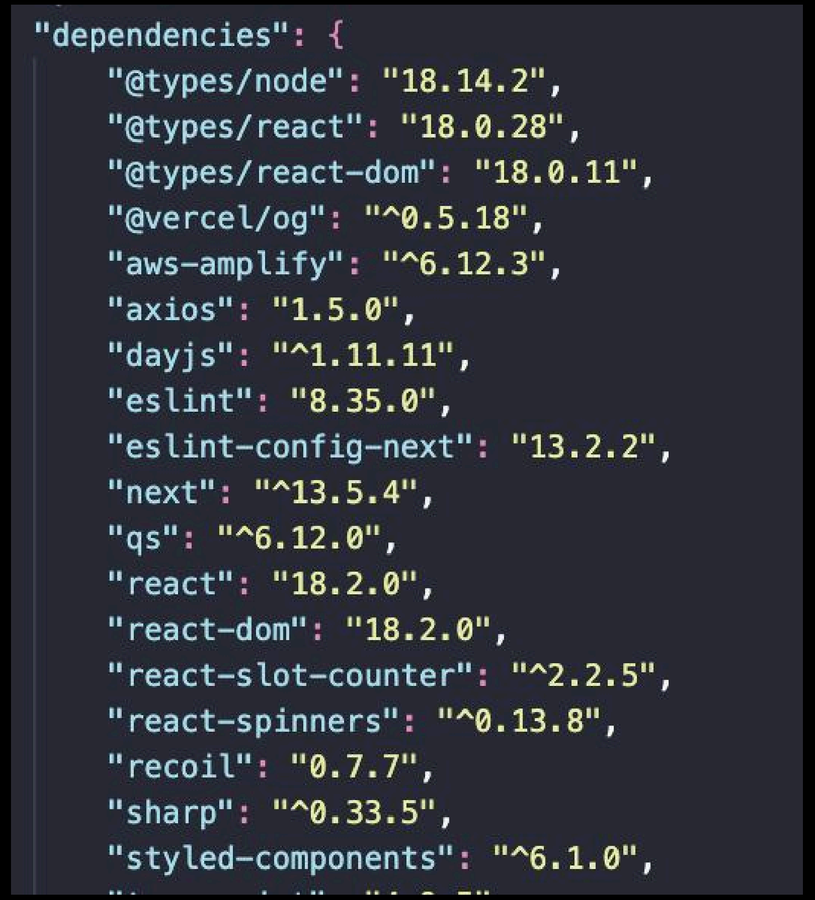
Resolution(해결)

뭔가를 해결하는 단계

1. `package.json` 파일에 명시된 버전 범위에 따라 정확한 버전을 결정.
2. 설치한 라이브러리가 사용하는 다른 라이브러리, 즉 의존성의 의존성 문제를 해결
3. 그 의존성의 버전도 고정해야 한다는 문제를 해결.

Resolution(해결)

package.json 파일에 명시된 버전 범위에 따라 정확한 버전을 결정.



Resolution(해결)

설치한 라이브러리가 사용하는 다른 라이브러리, 즉 의존성의 의존성 문제를 해결한다

```
"react": "18.2.0",
"react-dom": "18.2.0",
```

Resolution(해결)

그 의존성의 버전도 고정해야 한다는 문제를 해결.

```
"react": "18.2.0",
"react-dom": "18.2.0",
```

Resolution(해결)

그 의존성의 버전도 고정해야 한다는 문제를 해결.

단계	설명
패키지 분석	· package.json을 읽어 설치할 패키지와 버전 범위를 파악합니다. · 예: `axios: "^1.5.0"` (1.5.0 이상, 2.0.0 미만)
메타데이터 조회	· npm 레지스트리에서 패키지 정보와 사용 가능한 버전을 가져옵니다. · 지정된 버전 범위에서 최적의 버전을 선택합니다.
의존성 트리 구성	· 각 패키지의 하위 의존성을 재귀적으로 분석하여 전체 의존성 트리를 구축합니다. · 버전 충돌 발생 시 호이스팅 또는 중복 설치 전략을 적용합니다.

Resolution(해결)

Resolution 단계는 **모든 기기에서 고정된 버전을 사용**할 수 있도록 합니다.
의존성 버전을 전부 고정시키고, 의존성의 의존성을 다 찾아서 그 버전도 고정시키며, **결과물을 yarn.lock 이나 package-lock.json에 저장.**

Fetch

Resolution의 결과로 결정된 버전을 실제로 다운로드하는 과정.

99%는 npm 레지스트리에서 다 받아옴

Fetch

Resolution의 결과로 결정된 버전을 실제로 다운로드하는 과정.

99%는 npm 레지스트리에서 다 받아옴

```
"@ampproject/remapping@^2.2.0":
  version "2.3.0"
  resolved "https://registry.yarnpkg.com/@ampproject/remapping/-/remapping-2.3.0.tgz#ed441b6fa600072520c
  integrity sha512-30iZtAPgz+LTIYoeivqYo853f02jBYSd5uGnGpkFV0M3xOt9aN73erkgYAmZU43x4VfqcnLxW9Kpg3R5LC4YY
  dependencies:
    "@jridgewell/gen-mapping" "^0.3.5"
    "@jridgewell/trace-mapping" "^0.3.24"

"@aws-amplify/analytics@7.0.68":
  version "7.0.68"
  resolved "https://registry.yarnpkg.com/@aws-amplify/analytics/-/analytics-7.0.68.tgz#bb505c40158b27454
  integrity sha512-rHsQi+kBTQPAiKBqXp6nWRa2Z8401DYRMba26tTaD7UPbDYMaXW5yWRv8NHVa1qADgBNXpUXKIn0ww5lnbokY
  dependencies:
    "@aws-sdk/client-firehose" "3.621.0"
    "@aws-sdk/client-kinesis" "3.621.0"
    "@aws-sdk/client-personalize-events" "3.621.0"
    "@smithy/util-utf8" "2.0.0"
    tslib "^2.5.0"
```

Link

Resolution/Fetch 된 라이브러리를 소스 코드에서 사용할 수 있는 환경을 제공하는 과정

Link

Resolution/Fetch 된 라이브러리를 소스 코드에서 사용할 수 있는 환경을 제공하는 과정

Link

Resolution/Fetch 된 라이브러리를 소스 코드에서 사용할 수 있는 환경을 제공하는 과정

패키지 매니저마다 달라요

Link - npm Linker

가장 익숙한 node_modules 기반의 Linker

package.json에서 명시하는 모든 의존성을 그냥 node_modules 디렉토리 밑에다가 하나하나씩 쓰는 게 npm Linker의 역할

Link - npm Linker

1. 단 패키지를 찾으려고 하면 node_modules를 계속 타고 올라가면서 **파일을 여러 번 읽어**야 함.
2. 그래서 **import나 require 하는 속도가 느려짐**
3. **디렉토리 크기가 너무 커짐**. 실제로 파일 시스템에 디렉토리와 파일을 하나하나 만들고 쓰기 때문

Link - pnpm Linker

퍼포먼스가 향상된(performant) npm

Link - pnpm Linker

퍼포먼스가 향상된(performant) npm

1. pnpm Linker는 기존의 node_modules 디렉토리를 그대로 사용
2. 대신 보다 빠르고 용량을 최적화하는 방식 ⇒ **Hard link 방식(alias를 걸어둠)**

Link - pnpm Linker

퍼포먼스가 향상된(performant) npm

1. pnpm Linker는 기존의 node_modules 디렉토리를 그대로 사용
2. 대신 보다 빠르고 용량을 최적화하는 방식 ⇒ **Hard link 방식(alias를 걸어둠)**
   → alias가 생기면 거기로 바로 접근. 의존성이 디스크에 하나만 설치됨.

Link - pnpm Linker

퍼포먼스가 향상된(performant) npm

1. pnpm Linker는 기존의 node_modules 디렉토리를 그대로 사용
2. 대신 보다 빠르고 용량을 최적화하는 방식 ⇒ **Hard link 방식(alias를 걸어둠)**
   → alias가 생기면 거기로 바로 접근. 의존성이 디스크에 하나만 설치됨.
   → 용량이 적어지고, 빠른 접근이 가능해짐.

Link - pnpm Linker

퍼포먼스가 향상된(performant) npm

1. pnpm Linker는 기존의 node_modules 디렉토리를 그대로 사용
2. 대신 보다 빠르고 용량을 최적화하는 방식 ⇒ **Hard link 방식(alias를 걸어둠)**
   → alias가 생기면 거기로 바로 접근. 의존성이 디스크에 하나만 설치됨.
   → 용량이 적어지고, 빠른 접근이 가능해짐.
   → BUT, 파일 읽기가 많이 발생

Link - PnP(Plug'n'Play) Linker

node_modules 디렉토리에서 벗어나고 싶다는 생각으로 래디컬하게 접근한 게 PnP

Link - PnP(Plug'n'Play) Linker

‘패키지를 import 할 때 중요한 것은 단 두 가지’라는 관점에서 접근
1. ‘어떤 파일’에서 import 하는가
2. ‘무엇’을 import 하는가

Link - PnP(Plug'n'Play) Linker

‘패키지를 import 할 때 중요한 것은 단 두 가지’라는 관점에서 접근
1. ‘어떤 파일’에서 import 하는가
2. ‘무엇’을 import 하는가

→ npm과 pnpm처럼 node_modules를 순회하는 게 중요하지 않다고 생각
→ node_modules 디렉토리가 아니라 JavaScript 객체로 똑똑하게 처리

Link - PnP(Plug'n'Play) Linker

yarn install → .pnp.cjs라는 파일 생성

Link - PnP(Plug'n'Play) Linker

yarn install → .pnp.cjs라는 파일 생성

```
["my-service", /* ... */ [{
  // ./my-service에서...
  "packageLocation": "./my-service/",
  "packageDependencies": [
    // React를 import 하면 18.2.0 버전을 제공하라.
    ["react", "npm:18.2.0"]
  ]
]
```

Link - PnP(Plug'n'Play) Linker

yarn install → .pnp.cjs라는 파일 생성

1. 설치 속도가 빠름.

```
["my-service", /* ... */ [{
  // ./my-service에서...
  "packageLocation": "./my-service/",
  "packageDependencies": [
    // React를 import 하면 18.2.0 버전을 제공하라.
    ["react", "npm:18.2.0"]
  ]
]
```

감사합니다