



Home

2022-06-27

Lerna는 공식적으로 죽었다. 모노레포여 영원하길

원문: <https://betterprogramming.pub/lerna-is-officially-dead-long-live-monorepos-9853c80a7b0e>

4가지 도구를 사용하여 JavaScript 모노레포를 효율적으로 관리할 수 있습니다.

JS | MONOREPOS

모노레포 방식은 모든 종속성을 관리하는 좋은 방법입니다. 그리고 이는 몇 년 동안 뜨거운 주제였습니다. 모노레포는 잘 정의된 관계를 통해 개별 프로젝트를 유지할 수 있는 좋은 방법입니다. 즉각적인 이점 중 하나는 ESLint, Prettier, TypeScript 등과 같은 구성을 서로 다른 프로젝트에서 공유하여 일관성을 보장할 수 있다는 점입니다.


npm, yarn 및 pnpm과 같은 도구는 모노레포를 자체적으로 지원합니다. 그러나 몇가지 필요한 기능을 지원하지 않아 아쉬움이 있었습니다. 이러한 부족함을 Lerna에서는 지원했습니다. Lerna의 목표는 패키지를 빌드, 조정 및 배포할 수 있도록 확장성을 향상시키는 것이었습니다.

Lerna는 최근에 공식적으로 수명이 다했습니다. 비록 메인테이너였던 @evocateur가 그의 번아웃과 함께 프로젝트를 포기하려는 의도를 밝혔지만, 그것은 2020년 4월까지 공식화되지 않았었습니다.

중요한 참고: 이 프로젝트는 현재 **활발 유지 관리되지 않습니다**. 모노레포 관리를 위한 대체 도구 채택을 고려하십시오. — Lerna의 Readme.md에 명시된 내용입니다.

이 프로젝트는 매일 130만 건 이상의 다운로드가 지속적으로 발생하고 있지만 다른 곳에 이관되지 않을 것입니다. 왜 그럴까요? 제작자는 그것을 수정하기보다는 다시 만들기를 열망할 것입니다. 최신 툴링의 성능과 경쟁할 수 없었습니다. 비록 많은 사람들이 여전히 Lerna를 사랑하고 즐긴다고 해도, 이제는 나아가야 할 때입니다.

그렇다면 JavaScript/TypeScript 모노레포 에코시스템을 구축할 때 우리가 선택할 수 있는 것은 무엇 일까요? 이 글에서는 가장 인기 있는 도구를 알아보려고 합니다.

업데이트: 이 글이 게시된 후 내용이 뒤바뀌었습니다. Lerna의 제작자가 프로젝트를 **Nrwl** 에게 넘기기로 결정했습니다. 그들의 로드맵은 여전히 공개되어야 할 것입니다. 지난 한 해 동안 Lerna가 비활성화되어 뒤쳐지게 되었습니다.

1. Yarn, Npm, and Pnpm

앞서 언급했듯이 npm, yarn, pnpm은 이미 자체적으로 모노레포 지원으로 출하되고 있습니다. 그렇다면 "Lerna"나 다른 도구가 정말 필요한지 자문해 볼 필요가 있습니다. 여러분의 모노레포는 그렇게 복잡합니까?

yarn은 모노레포 기능 지원에 있어서 선구자였고, npm은 2020년 10월 출시된 버전 7부터 지원하고 있습니다. npm, yarn, pnpm 세 가지 패키지 매니저 사이에는 많은 유사점이 있습니다.

pnpm에서 작업 영역 구성은 pnpm-workspace.yaml에 있고 yarn 및 npm에서는 루트 package.json에 있습니다.

npm에서 작업영역(workspaces)을 사용한 모노레포 설정 예제입니다.

```
// /.package.json
{
  // ...
  "workspaces": ["/packages/*"]
}
```

테스트와 같은 명령을 실행하려면 npm의 --workspaces와 pnpm의 -r 또는 recursive를 사용할 수 있습니다.

```
# 모든 워크스페이스를 새롭게 정리합니다.
pnpm -r exec -- rm -rf node_modules && rm pnpm-lock.yaml
# @doppelmutzi 범위의 모든 작업영역에서 모든 테스트를 수행합니다.
pnpm recursive run test --filter @doppelmutzi/
```

yarn 의 한 가지 특징은 workspace: 참조를 동적으로 대체하여 종속성 프로세스를 단순화할 수 있다는 것입니다. 이는 매우 편리합니다.

```
{
  "dependencies": {
    "star": "workspace:*",
    "caret": "workspace:^",
    "tilde": "workspace:~",
    "range": "workspace:^1.2.3",
    "path": "workspace:path/to/baz"
  }
}
```

이 세 가지 중에서 어떤 것이 가장 좋을까요? 비록 npm 이 빠르게 따라잡고 있지만 좀 더 성숙한 특징을 가지고 있는 yarn 을 뽑을 것 같습니다. pnpm 은 디스크를 사용함으로써 얻는 효율성과 같은 다른 이점도 있지만, 이는 yarn berry 에서도 지원하고 있습니다.

이 세 가지 중 어떤 것을 언제 사용해야 할까요? 모두 상대적으로 규모가 작은 프로젝트에서 사용하는 것이 좋습니다. 왜냐하면 프로젝트 오케스트레이션 측면에서 이점은 없기 때문입니다. 복잡한 종속성 그래프 또는 릴리스가 필요한 경우 다른 도구를 선택하는 것이 좋습니다.

(역주) [오케스트레이션](#)

2. 러쉬(Rush) 스택

러쉬는 마이크로소프트가 JavaScript/TypeScript 프로젝트를 위한 모노레포 관리를 해결하기 위해 만든 도구입니다. 내부 관리 도구로써 만들어졌고 발전해왔습니다.

이 도구의 가장 좋은 특징은 무엇일까요?

- 자동 로컬 연결(linking)
- 빠른 빌드
- 부분 또는 증분 빌드
- 순환 종속성
- 대량 발행

설치 및 시작은 매우 간단합니다. 설치 방법은 다음과 같습니다.

```
npm install -g @microsoft/rush
```

이 도구는 응용 프로그램을 가장 잘 구성하는 방법에 대한 상당한 수준의 제어 기능을 제공합니다. 이견 단지 저의 의견이 아니며 빌드 및 릴리스 오케스트레이터로 볼 수 있습니다. 3개의 상위 패키지 매니저들 (yarn , pnpm , 또는 npm) 가운데 어떤 것이라도 그 위에 구축할 수 있습니다. 이 세 가지 중에서는

npm 과 가장 호환성이 좋습니다. 유일한 주의 사항은 꽤 오래된 버전인 4.5.0과 함께 사용할 것을 제안한다는 것입니다.

어떻게 작동할까요? 이 모든 것은 `rush.json` 파일을 통해 구성됩니다. `pnpm` 도구를 사용하는 예를 보겠습니다.

```
// rush.json
/**
 * 다음 필드는 패키지 매니저가 어떤 버전으로 설치되어야 하는지 나타냅니다.
 * 러쉬는 여러분의 빌드 과정을 보장하기 위해 패키지 관리자의 로컬 복사본을 설치합니다.
 * 이 내용은 로컬 환경의 모든 도구로부터 완전히 독립적입니다.
 *
 *
 * "pnpmVersion", "npmVersion", or "yarnVersion" 중 하나를 사용할 때의 명세입니다.
 */
"pnpmVersion": "2.15.1",

// "npmVersion": "4.5.0",
// "yarnVersion": "1.9.4",
```

이 파일에서 프로젝트의 종속성이 어떻게 구성되는지 살펴보겠습니다.

```
"projects": [
  {
    "packageName": "my-toolchain",
    "projectFolder": "tools/my-toolchain"
  }
]
```

여전히 다른 툴링과 상당히 일치합니다. 러쉬의 툴링은 프로젝트 npm 종속성을 세부적으로 제어하고자 하는 대규모 조직에 적합합니다.

3. 터보레포(Turborepo)



이미지 출처: <https://github.com/vercel/turborepo>

Vercel은 최근 터보레포 회사를 인수하여 똑똑한 모노레포 도구를 선점했습니다. 터보레포는 Formik를 만든 사람이 참여했고 Go 언어를 사용하여 만들어졌습니다. 특히 병렬 실행에서 탁월한 성능을 발휘합니다.

가장 좋은 특징은 무엇일까요?

- 컴퓨팅 및 원격 캐싱
- 병렬 태스크 실행
- 사용 및 구성이 용이함
- 종속성 시각화
- 변경 사항이 지속적으로 통합됨

시작하기 전에 다음을 실행하여 터미널 설정 마법사를 트리거할 수 있습니다.

```
// 실행
npx create-turbo@latest my-turbo-repo
```

러쉬와 마찬가지로 터보레포도 패키지 관리자를 선택해야 합니다. npm, pnpm 및 yarn 옵션을 사용할 수 있습니다. 이미 기본 작업 영역들(workspace)을 사용하고 있다면 이 도구와 잘 통합되므로 쉽게 전환할 수 있습니다.

"*" 표기법을 사용하여 패키지 종속성을 관리하는 방법에 주목해주세요. 이렇게 하면 앱이 최신 버전으로 유지됩니다.

```
"dependencies": {
  "next": "12.0.8",
  "react": "17.0.2",
  "react-dom": "17.0.2",
  "ui": "*"
},
```

우리는 turbo.json을 통해 빌드 파이프라인을 만들 수 있습니다. 프로젝트가 어떻게 만들어지는지 정의하는 곳입니다.

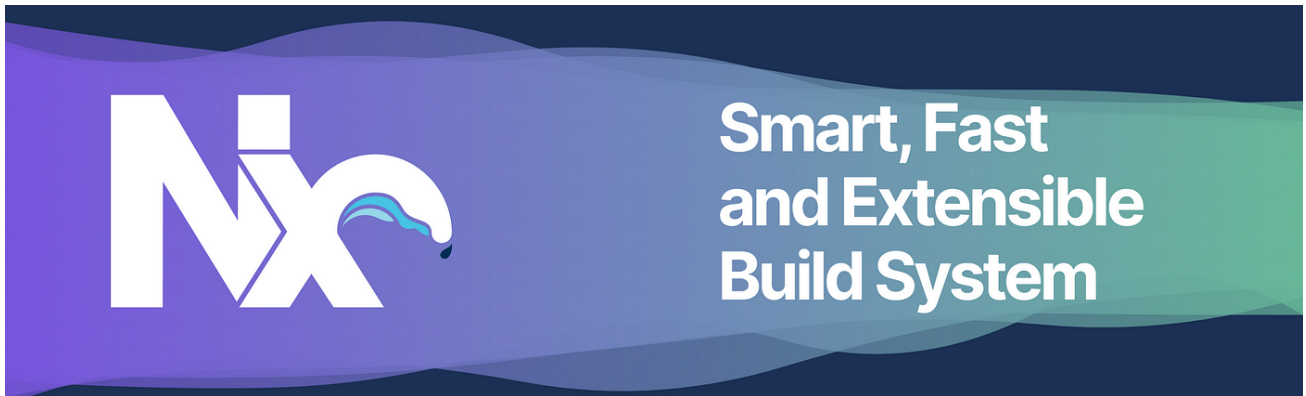
```
{
  "pipeline": {
    "build": {
      "dependsOn": ["^build"],
      "outputs": ["dist/**", ".next/**"]
    },
    "lint": {
      "outputs": []
    },
    "dev": {
      "cache": false
    }
  }
}
```

```
}
}
```

이 도구는 빌딩 프로세스를 심층적으로 구성하고자 하는 독립적인 모노레포를 원하는 경우 유용합니다.

유망하지만 Nx 와 같은 성숙한 툴링을 따라잡으려면 아직 갈 길이 멉니다. Vercel 과 함께라면 큰 발전을 이룰 것으로 보입니다. 이 도구는 미래를 위한 배팅일 것입니다.

4. NX



Nx는 Nrwl 팀에 의해 만들어졌습니다. 몇 년 동안이나 이용 가능했지만 최근에 매우 인기가 많아졌습니다. TypeScript를 사용하여 빌드됩니다만 이것에 속지 마세요. 성능이 정말 뛰어납니다. C++ 로 작성된 모듈과 코어 Node.js 에 의해 많은 계산이 수행됩니다.

이 도구의 철학은 프로젝트가 특정한 구조를 가져야 한다는 것입니다. 그들은 전체 구조를 건조하고, 다용도적이며, 플러그가 가능한 상태로 유지하기 위해 비트 플러그인 생태계를 만들었습니다. Nx는 확장 가능하고, 빠르고, 똑똑한 빌드 시스템입니다.

가장 좋은 특징은 무엇일까요?

- 스마트 리빌드. 변경 사항의 영향을 받는 내용만 다시 빌드하고 다시 테스트합니다.
- 프로젝트 그래프
- 분산 태스크 실행
- 컴퓨팅 및 원격 캐싱
- 개발자 경험
- 소유권 관리
- 발전기
- 플러그인 생태계
- CL/V 코드 확장

마이그레이션은 간단합니다. 먼저 명령을 실행하여 Nx를 추가합니다.

```
// 실행
npx add-nx-to-monorepo
```

위 내용은 무엇을 하는걸까요?

- package.json에 Nx를 추가합니다.
- 필요한 구성을 모두 포함하는 nx.json을 생성합니다.
- 무료이며 인증이 필요하지 않은 Nx Cloud를 설정합니다("예"를 선택한 경우).

다음 단계는 아래와 같습니다.

- 플러그인 사용
- 플러그인에 활용되는 중복 구성 삭제

Nx는 견고하고 성능이 뛰어납니다. 그것은 터보레포 와 러쉬 보다 더 성숙합니다. 훌륭한 커뮤니티 지원과 더 많은 리소스들이 있습니다. VSCode 플러그인을 사용하면 CLI를 매우 직관적으로 사용할 수 있습니다. 여러분은 이 도구를 선택함으로써 절대 잘못될 수 없습니다. 이 도구는 안전한 배팅 입니다.

정리

현재 Lerna 프로젝트는 Nx와 거의 같은 다운로드 수를 가지고 있으며 터보레포와 러쉬를 합친 것보다 더 많습니다. 유지보수가 부족하다는 인식이 확산됨에 따라 이 트렌드는 바뀔 것 입니다.

비록 꽤 새롭지만, 터보레포는 지금까지 내가 가장 좋아하는 것 중 하나가 되었습니다. 이 도구는 모든 체크박스를 충족하는 것처럼 보입니다. 추천을 드리자면 두 말할 것도 없이 Nx 입니다. Nx 는 성숙하고, 빠르고, 신뢰할 수 있습니다.

감사합니다.