개요

Nest CLI는 Nest 애플리케이션을 초기화, 개발 및 유지 관리하는 데 도움이 되는 명령줄 인터페이스 도구입니다. 프로젝트 스캐폴딩, 개발 모드에서 서비스, 프로덕션 배포를 위한 애플리케이션 빌드 및 번들링 등 다양한 방식으로 지원합니다. 모범 사례 아키텍처 패턴을 구현하여 잘 구조화된 앱을 장려합니다.

설치

참고: 이 가이드에서는 Nest CLI를 포함한 패키지를 설치하기 위해 npm을 사용하는 방법을 설명합니다. 재량에 따라 다른 패키지 관리자를 사용할 수도 있습니다. npm을 사용하면 OS 명령줄에서 `네스트` CLI 바이너리 파일의 위치를 확인하는 방법을 관리하는 데 사용할 수 있는 몇 가지 옵션이 있습니다. 여기서는 `-g` 옵션을 사용하여 `네스트` 바이너리를 전역적으로 설치하는 방법을 설명합니다. 이는 어느 정도 편의성을 제공하며 문서 전체에서 가정하는 접근 방식입니다. `npm` 패키지를 전역으로 설치하면 올바른 버전이 실행되고 있는지 확인할 책임이 사용자에게 있다는 점에 유의하세요. 또한 서로 다른 프로젝트가 있는 경우 각 프로젝트가 동일한 버전의 CLI를 실행한다는 의미이기도 합니다. 합리적인 대안은 `npm` cli에 내장된 npx 프로그램(또는 다른 패키지 관리자와 유사한 기능)을 사용하여 관리되는 버전의 Nest CLI를 실행하도록 하는 것입니다. 자세한 내용은 npx 설명서 및/또는 개발자 지원 담당자에게 문의하는 것이 좋습니다.

`npm install -g` 명령을 사용하여 CLI를 전역으로 설치합니다(전역 설치에 대한 자세한 내용은 위의 참고 사항 참조).

```
$ npm install -g @nestjs/cli
```

> 정보 힌트 또는 cli를 전역적으로 설치하지 않고 다음 명령을 사용할 수 있습니다.

기본 워크플로

설치가 완료되면 `네스트를` 통해 OS 명령줄에서 직접 CLI 명령을 호출할 수 있습니다.
실행 파일을 다운로드합니다. 다음을 입력하여 사용 가능한 `네스트` 명령을 확인하세요:

```
nest --help
```

다음 구문을 사용하여 개별 명령에 대한 도움말을 확인하세요. 아래 예제에서 생성이라고 표시된 곳에 새로 만들기, 추가 등의 명령을 대입하면 해당 명령에 대한 자세한 도움말을 볼 수 있습니다:

```
nest 생성 --help
```

개발 모드에서 새 기본 Nest 프로젝트를 생성, 빌드 및 실행하려면 새 프로젝트의 부모가 될 폴더로 이동하여 다음 명령을 실행합니다:

```
nest new my-nest-project
$ cd my-nest-project
$ npm 실행 시작:dev
```

브라우저에서 http://localhost:3000 을 열어 새 애플리케이션이 실행되는 것을 확인합니다. 소스 파일을 변경하면 앱이 자동으로 다시 컴파일되고 다시 로드됩니다.

> 정보 힌트 더 빠른 빌드를 위해 SWC 빌더를 사용하는 것이 좋습니다(기본 TypeScript 컴파일러보다 10배 이상 성능 향상).

프로젝트 구조

`nest new`를 실행하면 Nest는 새 폴더를 생성하고 초기 파일 세트를 채워 상용구 애플리케이션 구조를 생성합니다. 이 문서 전체에 설명된 대로 새 컴포넌트를 추가하여 이 기본 구조에서 계속 작업할 수 있습니다. `Nest에서` 생성된 프로젝트 구조를 표준 모드라고 합니다. Nest는 여러 프로젝트와 라이브러리를 관리하기 위한 대체 구조인 모노레포 모드도 지원합니다.

빌드 프로세스 작동 방식(기본적으로 모노레포 모드는 모노레포 스타일 프로젝트 구조에서 발생할 수 있는 빌드 복잡성을 간소화함) 및 기본 제공 라이브러리 지원과 관련된 몇 가지 특정 고려 사항을 제외하고 나머지 Nest 기능 및 이 문서는 표준 및 모노레포 모드 프로젝트 구조 모두에 동일하게 적용됩니다. 실제로 향후 언제든지 표준 모드에서 모노레포 모드로 쉽게 전환할 수 있으므로 Nest에 대해 배우는 동안 이 결정을 안전하게 연기할 수 있습니다.

두 모드 중 하나를 사용해 여러 프로젝트를 관리할 수 있습니다. 다음은 두 모드의 차이점을 간단히 요약한 것입니다:

기능	표준 모드	모노레포 모드
여러 프로젝트	별도의 파일 시스템 구조	단일 파일 시스템 구조
`노드 모듈` 및 `package.json`	인스턴스 분리	모노레포에서 공유
기본 컴파일러	`tsc`	웹팩
컴파일러 설정	별도로 지정	모노레포 기본값은 다음과 같습니다. 프로젝트별로 재정의
다음과 같은 구성 파일 `.eslintrc.js`,	`.prettierrc` 등	

	별도로 지정	모노레포에서 공유
네스트 빌드 및 네스트 시작 명령	Target은 컨텍스트의 (유일한) 프로젝트로 자동 기본 설정됩니다.	Target은 기본적으로 모노레포의 기본 프로젝트로 설정됩니다.
라이브러리	일반적으로 npm 패키징을 통해 수동으로 관리됩니다.	경로 관리 및 번들링을 포함한 기본 제공 지원

어떤 모드가 가장 적합한지 결정하는 데 도움이 되는 자세한 정보는 작업 공간 및 라이브러리 섹션을 참조하세요.

CLI 명령 구문

모든 네스트 명령은 동일한 형식을 따릅니다:

```
nest commandOr별칭 requiredArg [optionalArg] [옵션]
```

예를 들어

```
nest new my-nest-project --dry-run
```

여기서 new는 *commandOrAlias 입니다*. 새 명령의 별칭은 n이고, my-nest-project의 별칭은 *requiredArg 입니다*. 명령줄에 requiredArg가 제공되지 않으면 nest가 이를 묻는 메시지를 표시합니다. 또한 --dry-run에는 이에 상응하는 약식 -d가 있습니다. 이를 염두에 두고 다음 명령은 위 명령과 동일합니다:

```
nest n my-nest-project -d
```

대부분의 명령과 일부 옵션에는 별칭이 있습니다. nest new --help를 실행하여 이러한 옵션과 별칭을 확인하고 위의 구문을 이해했는지 확인하세요.

명령 개요

다음 명령 중 하나를 실행하여 명령별 옵션을 보려면 nest <command> --help를 입력합니다. 각 명령에 대한

자세한 설명은 사용법을 참조하세요.

명령	Alias	설명
new	n	실행에 필요한 모든 상용구 파일이 포함된 새로운 *표준 모드* 애플리케이션을 스캐폴드합니다.
생성	g	회로도를 기반으로 파일을 생성 및/또는 수정합니다.
빌드		애플리케이션 또는 작업 공간을 출력 폴더로 컴파일합니다.
시작		애플리케이션(또는 워크스페이스의 기본 프로젝트)을 컴파일하고 실행합니다.

추가		네스트 라이브러리로 패키징된 라이브러리를 임포트하여 설치 스키마를 실행합니다.
정보	i	설치된 네스트 패키지에 대한 정보 및 기타 유용한 시스템 정보를 표시합니다. 요구

사항

Nest CLI에는 공식 바이너리와 같은 국제화 지원(ICU)으로 빌드된 Node.js 바이너리가 필요합니다.
를 다운로드하세요. ICU와 관련된 오류가 발생하면 바이너리가 이 요구 사항을 충족하는지 확인하세요.

```
node -p process.versions.icu
```

명령이 정의되지 않은 상태로 출력되면 Node.js 바이너리가 국제화를 지원하지 않는 것입니다.

작업 공간

Nest에는 코드 정리를 위한 두 가지 모드가 있습니다:

- 표준 모드: 자체 종속성과 설정이 있고 모듈 공유 또는 복잡한 빌드 최적화를 위해 최적화할 필요가 없는 개별 프로젝트 중심 애플리케이션을 빌드하는 데 유용합니다. 기본 모드입니다.
- 모노레포 모드: 이 모드는 코드 아티팩트를 경량 모노레포의 일부로 취급하며, 개발자 팀 및/또는 다중 프로젝트 환경에 더 적합할 수 있습니다. 빌드 프로세스의 일부를 자동화하여 모듈식 컴포넌트를 쉽게 만들고 구성할 수 있으며, 코드 재사용을 촉진하고, 통합 테스트를 더 쉽게 수행할 수 있으며, 에슬린트 규칙 및 기타 구성 정책과 같은 프로젝트 전체 아티팩트를 쉽게 공유할 수 있고, github 서브모듈과 같은 대안보다 사용하기가 쉽습니다.

  모노레포 모드는 `nest-cli.json` 파일에 표시된 작업 공간 개념을 사용하여 모노레포의 구성 요소 간의 관계를 조정합니다.

Nest의 거의 모든 기능은 코드 정리 모드와 무관하다는 점에 유의하세요. 이 선택에 따른 유일한 영향은 프로젝트 구성 방식과 빌드 아티팩트 생성 방식뿐입니다. CLI부터 핵심 모듈, 애드온 모듈에 이르기까지 다른 모든 기능은 어느 모드에서나 동일하게 작동합니다.

또한 언제든지 표준 모드에서 모노레포 모드로 쉽게 전환할 수 있으므로 두 가지 방법의 이점이 더 명확해질 때까지 이 결정을 미룰 수 있습니다.

표준 모드

네스트 새로 만들기를 실행하면 기본 제공 회로도를 사용하여 새 프로젝트가 만들어집니다. Nest는 다음을 수행합니다:

. 새 폴더를 중첩할 때 제공한 이름 인수에 해당하는 새 폴더를 만듭니다.
. 해당 폴더를 최소한의 기본 수준 Nest 애플리케이션에 해당하는 기본 파일로 채웁니다. 이러한 파일은 타입스크립트 스타터 리포지토리에서 확인할 수 있습니다.
. 애플리케이션을 컴파일, 테스트 및 제공하기 위한 다양한 도구를 구성하고 활성화하는 `nest-cli.json`, `package.json` 및 `tsconfig.json`과 같은 추가 파일을 제공합니다.

여기에서 시작 파일을 수정하고, 새 컴포넌트를 추가하고, 종속성(예: npm 설치)을 추가하고, 이 문서의 나머지

부분에서 설명하는 대로 애플리케이션을 개발할 수 있습니다.

모노레포 모드

모노레포 모드를 활성화하려면 *표준 모드* 구조로 시작하여 프로젝트를 추가합니다. 프로젝트는 전체 애플리케이션(명령 `네스트 생성 앱을` 사용하여 작업 공간에 추가) 또는 라이브러리(명령 `네스트 생성 라이브러리를` 사용하여 작업 공간에 추가)일 수 있습니다. 아래에서 이러한 특정 유형의 프로젝트 구성 요소에 대해 자세히 설명하겠습니다. 여기서 주목해야 할 핵심 사항은 기존 표준 모드 구조에 프로젝트를 추가하여 모노레포 모드로 변환하는 작업이라는 점입니다. 한 가지 예를 살펴보겠습니다.

우리가 달리면:

```
nest new my-project
```

표준 모드 구조는 다음과 같은 폴더 구조로 구성되었습니다: node_modules
src
app.controller.ts
app.module.ts
app.service.ts
main.ts
nest-cli.json
package.json
tsconfig.json
.eslintrc.js

이를 다음과 같이 모노레포 모드 구조로 변환할 수 있습니다:

```
$ cd my-project
nest 생성 앱 내 앱
```

이 시점에서 nest는 기존 구조를 모노레포 모드 구조로 변환합니다. 이로 인해 몇 가지 중요한 변경 사항이 발생합니다. 이제 폴더 구조는 다음과 같습니다:

앱 내-

앱 src

app.controller.ts
app.module.ts
app.service.ts
main.ts
tsconfig.app.json

내 프로젝트

src
app.controller.ts
app.module.ts
app.service.ts
main.ts
tsconfig.app.json
nest-cli.json
package.json
tsconfig.json
.eslintrc.js

`앱 생성` 스키마는 각 애플리케이션 프로젝트를 `앱` 폴더 아래로 이동하고 각 프로젝트의 루트 폴더에 프로젝트별 `tsconfig.app.json` 파일을 추가하는 등 코드를 재구성했습니다. 원래 `내 프로젝트` 앱이 모노

레포의 기본 프로젝트가 되었으며, 이제 방금 추가한 내 앱과 피어(apps 폴더 아래에 위치)가 되었습니다.

아래에서 기본 프로젝트에 대해 설명하겠습니다.

> 오류 경고 표준 모드 구조를 모노레포로 변환하는 것은 표준 Nest 프로젝트 구조를 따르는 프로젝트에서만 작동합니다. 특히, 변환하는 동안 스키마는 루트의 앱 폴더 아래에 있는 프로젝트 폴더의 src 및 테스트 폴더를 재배치하려고 시도합니다. 프로젝트가 이 구조를 사용하지 않는 경우 변환이 실패하거나 신뢰할 수 없는 결과가 생성됩니다.

작업 공간 프로젝트

모노레포는 워크스페이스라는 개념을 사용하여 멤버 엔티티를 관리합니다. 워크스페이스는 프로젝트로 구성됩니다. 프로젝트는 다음 중 하나일 수 있습니다:

- 애플리케이션: 애플리케이션을 부트스트랩하기 위한 main.ts 파일을 포함한 전체 Nest 애플리케이션. 컴파일 및 빌드 고려 사항을 제외하면 워크스페이스 내의 애플리케이션 유형 프로젝트는 *표준 모드* 구조 내의 애플리케이션과 기능적으로 동일합니다.
- 라이브러리: 라이브러리는 다른 프로젝트에서 사용할 수 있는 범용 기능 세트(모듈, 공급자, 컨트롤러 등)를 패키징하는 방식입니다. 라이브러리는 자체적으로 실행할 수 없으며 main.ts 파일이 없습니다. 라이브러리에 대한 자세한 내용은 여기에서 확인하세요.

모든 워크스페이스에는 기본 프로젝트(애플리케이션 유형 프로젝트여야 함)가 있습니다. 기본 프로젝트의 루트를 가리키는 nest-cli.json 파일의 최상위 "root" 속성에 의해 정의됩니다(자세한 내용은 아래 CLI 속성 참조). 일반적으로 이것은 표준 모드 애플리케이션으로 시작하여 나중에 네스트 생성 앱을 사용하여 모노레포 애플리케이션으로 변환한 것입니다. 다음 단계를 수행하면 이 속성이 자동으로 채워집니다.

기본 프로젝트는 프로젝트 이름이 제공되지 않은 경우 네스트 빌드 및 네스트 시작과 같은 네스트 명령에 사용됩니다.

예를 들어, 위의 모노레포 구조에서 실행 중인

```
nest start
```

를 누르면 내 프로젝트 앱이 시작됩니다. 내 앱을 시작하려면 다음을 사용합니다:

```
nest start my-app
```

애플리케이션

애플리케이션 유형 프로젝트 또는 비공식적으로 "애플리케이션"이라고도 하는 것은 실행 및 배포할 수 있는 완전한 Nest 애플리케이션입니다. `네스트 생성 앱으로` 애플리케이션 유형 프로젝트를 생성합니다.

이 명령은 타입스크립트 스타터에서 표준 `src` 및 `테스트` 폴더를 포함한 프로젝트 스켈레톤을 자동으로 생성합니다. 표준 모드와 달리 모노레포의 애플리케이션 프로젝트에는 패키지 종속성(`package.json`)이나 `.prettierrc` 및 `.eslintrc.js`와 같은 기타 프로젝트 구성 아티팩트가 없습니다. 대신 모노레포 전체 종속성 및 구성 파일이 사용됩니다.

그러나 스키마는 프로젝트의 루트 폴더에 프로젝트별 `tsconfig.app.json` 파일을 생성합니다. 이 구성 파일은 컴파일 출력 폴더를 올바르게 설정하는 등 적절한 빌드 옵션을 자동으로 설정합니다. 이 파일은 최상위(모노레포) `tsconfig.json` 파일을 확장하므로 모노레포 전체에서 전역 설정을 관리할 수 있지만 필요한 경우 프로젝트 수준에서 이를 재정의할 수 있습니다.

라이브러리

앞서 언급했듯이 라이브러리형 프로젝트 또는 간단히 "라이브러리"는 실행하기 위해 애플리케이션으로 구성해야 하는 Nest 구성 요소의 패키지입니다. `네스트 생성 라이브러리를` 사용하여 라이브러리형 프로젝트를 생성합니다. 라이브러리에 무엇이 포함될지 결정하는 것은 아키텍처 설계 결정입니다. 라이브러리에 대해서는 라이브러리 챕터에서 자세히 설명합니다.

CLI 속성

Nest는 표준 및 모노레포 구조의 프로젝트를 구성, 빌드 및 배포하는 데 필요한 메타데이터를 `nest-cli.json` 파일에 보관합니다. Nest는 프로젝트를 추가할 때 이 파일을 자동으로 추가하고 업데이트하므로 일반적으로 이 파일에 대해 생각하거나 내용을 편집할 필요가 없습니다. 그러나 수동으로 변경해야 하는 설정이 몇 가지 있으므로 파일에 대한 개요를 파악하고 있으면 도움이 됩니다.

위의 단계를 실행하여 모노레포를 생성한 후 `nest-cli.json` 파일은 다음과 같이 표시됩니다:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "apps/my-project/src",
  "monorepo": true,
  "root": "apps/my-project", "컴파일러옵션": {
    "webpack": true,
    "tsConfigPath": "apps/my-project/tsconfig.app.json"
  },
  "프로젝트": {
    "my-project": {
      "유형": "애플리케이션", "루트":
      "apps/my-project",
      "entryFile": "main",
      "sourceRoot": "apps/my-project/src",
      "컴파일러옵션": {
        "tsConfigPath": "apps/my-project/tsconfig.app.json"
      }
    },
    "my-app": {
      "유형": "애플리케이션", "루트":
      "apps/my-app",
      "entryFile": "main",
      "sourceRoot": "apps/my-app/src",
      "컴파일러옵션": {
        "tsConfigPath": "apps/my-app/tsconfig.app.json"
      }
    }
  }
}
```

파일은 섹션으로 나뉩니다:

- 표준 및 모노레포 전체 설정을 제어하는 최상위 속성이 있는 글로벌 섹션
- 각 프로젝트에 대한 메타데이터가 있는 최상위 속성("프로젝트")입니다. 이 섹션은 모노레포 모드 구조에만 존재합니다.

최상위 속성은 다음과 같습니다:

- "collection": 컴포넌트를 생성하는 데 사용되는 회로도 모음을 가리키며, 일반적으로 이 값을 변경해서는 안 됩니다.
- "sourceRoot": 표준 모드 구조에서는 단일 프로젝트의 소스 코드 루트를 가리키고, 모노레포 모드 구조에서는 *기본 프로젝트를* 가리킵니다.
- "compilerOptions": 컴파일러 옵션을 지정하는 키와 옵션 설정을 지정하는 값이 있는 맵; 자세한 내용은 아래를 참조하세요.
- "generateOptions": 전역 생성 옵션을 지정하는 키와 옵션 설정을 지정하는 값이 있는 맵(아래 세부 정보 참조).
- "monorepo": (모노레포만 해당) 모노레포 모드 구조의 경우, 이 값은 항상 참입니다• "root": (모노레포만 해당) *기본 프로젝트의* 프로젝트 루트를 가리킵니다.

글로벌 컴파일러 옵션

이러한 속성은 사용할 컴파일러를 지정하고, 네스트 빌드 또는 네스트 시작의 일부로 컴파일 단계에 영향을 주는 다양한 옵션을 지정하며, 컴파일러에 관계없이 tsc 또는 웹팩 등 컴파일러를 지정할 수 있습니다.

속성 이름속성	값 유형
웹팩 구성 경로	문자열
웹팩	부울
deleteOutDir	부울
tsConfigPath	문자열
자산	배열

설명	
true이면 웹팩 컴파일러를 사용합니다. false이거나 존재하지 않으면 `tsc`를 사용합니다. 모노레포 모드에서는 기본값이 참(웹팩 사용)이고, 표준 모드에서는 기본값이 거짓(`tsc` 사용)입니다. 자세한 내용은 아래를 참조하세요. (더 이상 사용되지 않음: 대신 `빌더` 사용)	(모노레포만 해당) `프로젝트` 옵션 없이 `네스트 빌드` 또는 `네스트` 시작이 호출될 때(예: 기본 프로젝트가 빌드 또는 시작될 때) 사용되는 `tsconfig.json` 설정이 포함된 파일을 가리킵니다.
	웹팩 옵션 파일을 가리킵니다. 지정하지 않으면 Nest는 `webpack.config.js` 파일을 찾습니다. 자세한 내용은 아래를 참조하세요.
	true이면 컴파일러가 호출될 때마다 먼저 컴파일 출력 디렉터리를 제거합니다(기본값은 `./dist`이며 `tsconfig.json`에 구성된 대로).
	컴파일 단계가 시작될 때마다 타입스크립트가 아닌 에셋을 자동으로 배포하도록 설정합니다(`--watch` 모드의 증분 컴파일에서는 에셋 배포가 발생하지 않음). 자세한 내용은 아래를 참조하세요.

속성 이름	속성 값 유형	설명
`watchAssets`	부울	true이면 감시 모드로 실행하여 모든 비타입스크립트 에셋을 감시합니다. (감시 대상 에셋을 보다 세밀하게 제어하려면 아래 에셋 섹션을 참조하세요.)
`manualRestart`	부울	true이면 바로 가기 `rs`를 사용하여 서버를 수동으로 다시 시작할 수 있습니다. 기본값은 `false`입니다.
`빌더`	문자열/객체	프로젝트를 컴파일하는 데 사용할 `빌더`(`tsc`, `swc` 또는 `웹팩`)를 CLI에 지시합니다. 빌더의 동작을 사용자 지정하려면 `유형`(`tsc`, `swc` 또는 `웹팩`)과 `옵션`이라는 두 가지 속성이 포함된 객체를 전달하면 됩니다.
`typeCheck`	부울	`true`이면 SWC 기반 프로젝트에 대해 유형 검사를 활성화합니다(`빌더`는 `SWC`입니다). 기본값은 `false`입니다.

글로벌 생성 옵션

이러한 속성은 `둥지 생성` 명령에서 사용할 기본 생성 옵션을 지정합니다.

속성 이름	속성 값 유형	설명
`사양`	부울 *또는* 객체	값이 부울인 경우 값이 `참`이면 기본적으로 `사양` 생성이 활성화되고 값이 `거짓`이면 비활성화됩니다. CLI 명령줄에서 전달된 플래그는 프로젝트별 `generateOptions` 설정과 마찬가지로 이 설정을 재정의합니다(자세한 내용은 아래 참조). 값이 오브젝트인 경우 각 키는 스키마 이름을 나타내며, 부울 값은 해당 특정 스키마에 대한 기본 사양 생성의 활성화/비활성화 여부를 결정합니다.
`flat`	boolean	true이면 모든 생성 명령이 플랫 구조를 생성합니다.

다음 예제에서는 부울 값을 사용하여 모든 프로젝트에서 기본적으로 스펙 파일 생성을 비활성화하도록 지정합니다:

```
{
  "생성옵션": { "spec":
    false
  },
  ...
}
```

다음 예제에서는 부울 값을 사용하여 모든 프로젝트에서 플랫 파일 생성을 기본값으로 지정합니다:

```
{
  "생성옵션": { "flat": true
  },
  ...
}
```

다음 예제에서는 `서비스` 도식(예: `둥지 생성 서비스...`)에 대해서만 `사양` 파일 생성이 비활성화됩니다:

```
{
  "generateOptions": {
    "spec": {
      "서비스": false
    }
  },
  ...
}
```

> 경고 사양을 객체로 지정할 때 생성 스키마의 키는 현재 자동 별칭 처리를 지원하지 않습니다. 즉, 예를 들어 `service: false로` 키를 지정하고 별칭 `s를` 통해 서비스를 생성하려고 해도 여전히 사양이 생성됩니다. 일반 스키마 이름과 별칭이 모두 의도한 대로 작동하는지 확인하려면 아래와 같이 일반 명령 이름과 별칭을 모두 지정하세요.
>
> ```
> {
>   "generateOptions": {
>     "spec": {
>       "서비스": 거짓, "S": 거
>       짓
>     }
>   },
>   ...
> }
> ```

프로젝트별 생성 옵션

전역 생성 옵션을 제공하는 것 외에도 프로젝트별 생성 옵션을 지정할 수도 있습니다. 프로젝트별 생성 옵션은 전역 생성 옵션과 완전히 동일한 형식을 따르지만 각 프로젝트에 직접 지정됩니다.

프로젝트별 생성 옵션이 전역 생성 옵션보다 우선합니다.

```
{
  "프로젝트": {
```

```
    "cats-project": {
      "generateOptions": {
        "spec": {
          "service": false
        }
      },
      ...
    }
  },
  ...
}
```

> 경고 경고 생성 옵션의 우선순위는 다음과 같습니다. CLI 명령줄에 지정된 옵션이 프로젝트별 옵션보다 우선합니다. 프로젝트별 옵션은 전역 옵션보다 우선합니다.

지정된 컴파일러

기본 컴파일러가 다른 이유는 대규모 프로젝트(예: 모노레포에서 더 일반적)의 경우 웹팩이 빌드 시간과 모든 프로젝트 구성 요소를 함께 묶은 단일 파일을 생성하는 데 상당한 이점을 가질 수 있기 때문입니다. 개별 파일을 생성하려면 `"webpack"을 false로` 설정하면 빌드 프로세스에서 `tsc`(또는 `swc)를` 사용하게 됩니다.

웹팩 옵션

웹팩 옵션 파일에는 표준 웹팩 구성 옵션이 포함될 수 있습니다. 예를 들어, 기본적으로 제외되는 node_modules를 번들링하도록 웹팩에 지시하려면 `webpack.config.js에` 다음을 추가합니다:

```
module.exports = { 외부:
  [],
};
```

웹팩 구성 파일은 자바스크립트 파일이므로 기본 옵션을 취하고 수정된 객체를 반환하는 함수를 노출할 수도 있습니다:

```
module.exports = function (options) {
  return {
    ...옵션,
    외부: [],
  };
};
```

자산

TypeScript 컴파일은 컴파일러 출력(`.js` 및 `.d.ts` 파일)을 지정된 출력 디렉터리에 자동으로 배포합니다. 또한 `.graphql` 파일, 이미지, `.html` 파일 및 기타 에셋과 같은 TypeScript가 아닌 파일을 배포하는 데 편리할 수 있습니다. 이렇게 하면 네스트 빌드(및 모든 초기 컴파일 단계)를 경량 개발 빌드 단계로 취급하여 TypeScript 이외의 파일을 편집하고 반복적으로 컴파일 및 테스트할 수 있습니다. 에셋은 `src` 폴더에 있어야 하며 그렇지 않으면 복사되지 않습니다.

`자산` 키의 값은 배포할 파일을 지정하는 요소 배열이어야 합니다. 예를 들어 요소는 글로브와 `같은` 파일 사양이 포함된 간단한 문자열일 수 있습니다:

```
"assets": ["**/*.graphql"],
"watchAssets": true,
```

세밀한 제어를 위해 요소는 다음 키를 가진 객체가 될 수 있습니다:

- `"include"`: 배포할 에셋에 대한 `글로브형` 파일 사양
- `"제외"`: `포함` 목록에서 제외할 에셋에 대한 `글로브 형식의` 파일 사양
- `"outDir"`: 에셋을 배포할 경로(루트 폴더 기준)를 지정하는 문자열입니다. 기본값은 컴파일러 출력에 구성된 것과 동일한 출력 디렉터리입니다.
- `"watchAssets"`: boolean; `true이면` 지정된 에셋을 감시하는 감시 모드로 실행합니다

예를 들어:

```
"자산": [
  { "include": "**/*.graphql", "exclude": "**/omitted.graphql",
"watchAssets": true },
]
```

> 경고 경고 최상위 `컴파일러 옵션` 프로퍼티에서 watchAssets를 설정하면 모든 `에셋` 프로퍼티 내의 `watchAssets` 설정을 변경합니다.

프로젝트 속성

이 요소는 모노레포 모드 구조에만 존재합니다. 이러한 속성은 Nest에서 모노레포 내에서 프로젝트와 해당 구성 옵션을 찾는 데 사용되므로 일반적으로 편집해서는 안 됩니다.

라이브러리

많은 애플리케이션이 동일한 일반적인 문제를 해결하거나 모듈식 구성 요소를 여러 다른 상황에서 재사용해야 합니다. Nest에는 이를 해결하는 몇 가지 방법이 있지만, 각기 다른 아키텍처 및 조직 목표를 달성하는 데 도움이 되는 방식으로 문제를 해결하기 위해 서로 다른 수준에서 작동합니다.

네스트 모듈은 단일 애플리케이션 내에서 컴포넌트를 공유할 수 있는 실행 컨텍스트를 제공하는 데 유용합니다. 모듈을 npm으로 패키징하여 다른 프로젝트에 설치할 수 있는 재사용 가능한 라이브러리를 만들 수도 있습니다. 이는 서로 느슨하게 연결되어 있거나 관련이 없는 여러 조직에서 사용할 수 있는 구성 가능하고 재사용 가능한 라이브러리를 배포하는 효과적인 방법이 될 수 있습니다(예: 타사 라이브러리를 배포/설치하는 방식).

긴밀하게 조직된 그룹(예: 회사/프로젝트 경계 내)에서 코드를 공유할 때는 컴포넌트 공유에 보다 가벼운 접근 방식을 사용하는 것이 유용할 수 있습니다. 이를 가능하게 하는 구조로 모노레포가 생겨났으며, 모노레포 내에서 라이브러리는 쉽고 가벼운 방식으로 코드를 공유할 수 있는 방법을 제공합니다. Nest 모노레포에서 라이브러리를 사용하면 컴포넌트를 공유하는 애플리케이션을 쉽게 어셈블할 수 있습니다. 이를 통해 모놀리식 애플리케이션과 개발 프로세스를 분해하여 모듈식 구성 요소를 빌드하고 구성하는 데 집중할 수 있습니다.

네스트 라이브러리

Nest 라이브러리는 자체적으로 실행할 수 없다는 점에서 애플리케이션과 다른 Nest 프로젝트입니다. 라이브러리 코드를 실행하려면 라이브러리를 포함하는 애플리케이션으로 가져와야 합니다. 이 섹션에서 설명하는 라이브러리에 대한 기본 지원은 모노레포에서만 사용할 수 있습니다(표준 모드 프로젝트는 npm 패키지를 사용하여 유사한 기능을 구현할 수 있음).

예를 들어, 조직에서 모든 내부 애플리케이션에 적용되는 회사 정책을 구현하여 인증을 관리하는 AuthModule을 개발할 수 있습니다. 각 애플리케이션에 대해 해당 모듈을 별도로 빌드하거나 npm으로 코드를 물리적으로 패키징하고 각 프로젝트에 설치하도록 하는 대신 모노레포에서 이 모듈을 라이브러리로 정의할 수 있습니다. 이렇게 구성하면 라이브러리 모듈의 모든 소비자는 커밋될 때 최신 버전의 AuthModule을 볼 수 있습니다. 이는 컴포넌트 개발 및 어셈블리를 조정하고 엔드투엔드 테스트를 간소화하는 데 상당한 이점을 가져다줄 수 있습니다.

라이브러리 만들기

재사용에 적합한 모든 기능은 라이브러리로 관리할 수 있는 후보입니다. 무엇이 라이브러리이고 무엇이 애플리케이션의 일부가 되어야 하는지를 결정하는 것은 아키텍처 설계의 결정입니다. 라이브러리 생성에는 단순히 기존 애플리케이션에서 새 라이브러리로 코드를 복사하는 것 이상의 작업이 포함됩니다. 라이브러리로 패키징할 때는 라이브러리 코드를 애플리케이션에서 분리해야 합니다. 따라서 사전에 더 많은 시간이 소요될 수 있으며, 더 긴밀하게 결합된 코드에서는 직면하지 않을 수 있는 몇 가지 설계 결정을 내려야 할 수도 있습니다. 하지만 라이브러리를 사용하여 여러 애플리케이션에서 애플리케이션을 더 빠르게 어셈블할 수 있다면 이러한 추가 노력은 보상이 될 수 있습니다.

라이브러리 만들기를 시작하려면 다음 명령을 실행합니다:

```
nest g library my-library
```

명령을 실행하면 `라이브러리` 회로도에서 라이브러리의 접두사(일명 별칭)를 입력하라는 메시지가 표시됩니다:

```
라이브러리에 어떤 접두사를 사용하시겠습니까(기본값: @app)?
```

그러면 작업 공간에 `내 라이브러리라는` 새 프로젝트가 생성됩니다. 라이브러리 유형 프로젝트는 애플리케이션 유형 프로젝트와 마찬가지로 회로도를 사용하여 명명된 폴더에 생성됩니다. 라이브러리는 모노레포 루트의 `libs` 폴더에서 관리됩니다. Nest는 라이브러리를 처음 만들 때 `libs` 폴더를 만듭니다.

라이브러리용으로 생성된 파일은 애플리케이션용으로 생성된 파일과 약간 다릅니다. 위 명령을 실행한 후 `libs` 폴더의 내용은 다음과 같습니다:

libs
내 라이브

러리 src

index.ts
내 라이브러리.모듈.ts

내 라이브러리.서비스

.ts
tsconfig.lib.json

`nest-cli.json` 파일의 `"projects"` 키 아래에 라이브러리에 대한 새 항목이 생깁니다:

```
...
{
    "내 라이브러리": {
      "유형": "라이브러리",
      "root": "libs/my-library",
      "entryFile": "index",
      "sourceRoot": "libs/my-library/src",
      "컴파일러옵션": {
        "tsConfigPath": "libs/my-library/tsconfig.lib.json"
      }
}
...
```

라이브러리와 애플리케이션 간에 `nest-cli.json` 메타데이터에는 두 가지 차이점이 있습니다:

- `"유형"` 속성이 `"애플리케이션"` 대신 `"라이브러리"`로 설정됩니다.

- "entryFile" 속성이 "main" 대신 "index"로 설정됩니다.

이러한 차이점은 라이브러리를 적절하게 처리하기 위한 빌드 프로세스의 핵심입니다. 예를 들어 라이브러리는 index.js 파일을 통해 함수를 내보냅니다.

애플리케이션 유형 프로젝트와 마찬가지로 라이브러리에는 각각 루트(모노레포 전체) tsconfig.json 파일을 확장하는 자체 tsconfig.lib.json 파일이 있습니다. 필요한 경우 이 파일을 수정하여 라이브러리별 컴파일러 옵션을 제공할 수 있습니다.

CLI 명령으로 라이브러리를 빌드할 수 있습니다:

```
nest build my-library
```

라이브러리 사용

자동으로 생성된 구성 파일을 사용하면 라이브러리를 사용하는 것이 간단합니다. 내 라이브러리 라이브러리에서 내 프로젝트 애플리케이션으로 MyLibraryService를 가져오려면 어떻게 해야 할까요?

먼저, 라이브러리 모듈을 사용하는 것은 다른 Nest 모듈을 사용하는 것과 동일하다는 점에 유의하세요. 모노레포가 하는 일은 라이브러리를 임포트하고 빌드를 생성하는 경로를 투명하게 관리하는 것입니다.

MyLibraryService를 사용하려면 선언 모듈을 임포트해야 합니다. 내 프로젝트/src/app.module.ts를 다음과 같이 수정하여 MyLibraryModule을 임포트할 수 있습니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./app.controller'에서 { AppController }를 임포트하고,
'./app.service'에서 { AppService }를 임포트합니다;
'@app/my-library'에서 { MyLibraryModule }을 가져옵니다;

모듈({
  임포트: [MyLibraryModule], 컨트롤러
  : [AppController], 제공자:
  [AppService],
})
내보내기 클래스 AppModule {}
```

위에서 ES 모듈 가져오기 줄에 @app이라는 경로 별칭을 사용했는데, 이 별칭은 위의 nest g 라이브러리 명령에 제공한 접두사입니다. 내부적으로 Nest는 tsconfig 경로 매핑을 통해 이 작업을 처리합니다. 라이브러리를 추가할 때 Nest는 다음과 같이 글로벌(모노레포) tsconfig.json 파일의 "경로" 키를 업데이트합니다:

```
"경로": {
    "@app/my-library": [
        "libs/my-library/src"
    ],
    "@app/my-library/*": [
        "libs/my-library/src/*"
    ]
}
```

간단히 말해, 모노레포와 라이브러리 기능의 결합으로 라이브러리 모듈을 애플리케이션에 쉽고 직관적으로 포함할 수 있게 되었습니다.

이와 동일한 메커니즘으로 라이브러리를 구성하는 애플리케이션을 빌드하고 배포할 수 있습니다. `MyLibraryModule을` 임포트한 후 `네스트 빌드를` 실행하면 모든 모듈 확인이 자동으로 처리되고 배포를 위해 모든 라이브러리 종속성과 함께 앱이 번들로 제공됩니다. 기본 컴파일러는

모노레포는 웹팩이므로 결과 배포 파일은 트랜스파일된 모든 JavaScript 파일을 단일 파일로 묶은 단일 파일입니다. 여기에 설명된 대로 `tsc로` 전환할 수도 있습니다.

CLI 명령 참조

둥지 새로 만들기

새 (표준 모드) Nest 프로젝트를 만듭니다.

```
nest new <이름> [옵션]
nest n <이름> [옵션]
```

설명

새 Nest 프로젝트를 생성하고 초기화합니다. 패키지 관리자를 위한 프롬프트입니다.

- 지정된 <이름>으로 폴더를 만듭니다.
- 구성 파일로 폴더를 채웁니다.
- 소스 코드(/src) 및 엔드투엔드 테스트(/test)를 위한 하위 폴더 생성• 앱 구성 요소 및 테스트를 위한 기본 파일로 하위 폴더를 채웁니다.

인수

인수	설명
<이름>	새 프로젝트의 이름입니다.

옵션

옵션	설명
--드라이런	변경 사항을 보고하지만 파일 시스템을 변경하지는 않습니다.
--skip-git	리자 [패키지- 관리자] --언어 [언어]
--스킵 인스톨	--컬렉션 [컬렉션 이름]
--패키지- 관	

별칭: `-d`	기화 건너뛰기. 별칭: `-g`
g i t	패키지 설치 건너뛰기. 별칭: `-s`
리	패키지 관리자를 지정합니다. `npm`, `yarn` 또는 `pnpm`을 사용합니다. 패키지 관리자는 전역적으로 설치해야 합니다.
포	별칭: `-p`
지	
토	프로그래밍 언어(`TS` 또는 `JS`)를 지정합니다. 별칭 : `-l`
리	회로도 컬렉션을 지정합니다. 회로도가 포함된 설치된 npm 패키지의 패키지 이름을 사용합니다.
초	별칭: `-c`

옵션	설명
`--엄격한`	다음 TypeScript 컴파일러 플래그를 활성화하여 프로젝트를 시작하세요: `strictNullChecks`, `noImplicitAny`, `strictBindCallApply`, `forceConsistentCasingInFileNames`, `noFallthroughCasesInSwitch`.

둥지 생성

회로도를 기반으로 파일을 생성 및/또는 수정합니다.

```
nest 생성 <도식> <이름> [옵션]
$ nest g <도식> <이름> [옵션]
```

인수

인수	설명
`<도식적>`	생성할 `회로도` 또는 `컬렉션:회로도입니다.` 아래 표에서 사용 가능한 회로도.
`<이름>`	생성된 컴포넌트의 이름입니다.

회로도

이름	별칭	설명
`앱`		모노레포 내에서 새 애플리케이션 생성(모노레포인 경우 모노레포로 변환) 표준 구조).
`라이브러리`	`lib`	모노레포 내에서 새 라이브러리를 생성합니다(표준 구조인 경우 모노레포로 변환).
`클래스`	`cl`	새 클래스를 생성합니다.
`컨트롤러`	`co`	컨트롤러 선언을 생성합니다.
`데코레이터`	`d`	사용자 지정 데코레이터를 생성합니다.
`필터`	`f`	필터 선언을 생성합니다.
`게이트웨이`	`ga`	게이트웨이 선언을 생성합니다.
`guard`	`gu`	가드 선언을 생성합니다.

인터페이스	itf	인터페이스를 생성합니다.
인터셉터	itc	인터셉터 선언을 생성합니다.
미들웨어	mi	미들웨어 선언을 생성합니다.
모듈	mo	모듈 선언을 생성합니다.

이름	Alias	설명
파이프	파이	파이프 선언을 생성합니다.
공급자	홍보	공급자 선언을 생성합니다.
해결자	r	리졸버 선언을 생성합니다.
리소스	res	새 CRUD 리소스를 생성합니다. 자세한 내용은 CRUD(리소스) 생성기를 참조하세요.
서비스	s	서비스 선언을 생성합니다.

옵션

옵션	설명
--드라이런	변경된 내용은 보고하지만 파일 시스템. 별칭: -d
--프로젝트 [프로젝트]	요소를 추가해야 하는 프로젝트입니다. 별칭: -p
--flat	요소에 대한 폴더를 생성하지 않습니다.
--컬렉션 [컬렉션 이름]	회로도 컬렉션을 지정합니다. 설치된 npm의 패키지 이름 사용 도식이 포함된 패키지입니다. 별칭: -c
--spec	스펙 파일 생성 적용(기본값)
--no-spec	사양 파일 생성 네스트 빌드

비활성화

애플리케이션 또는 작업 공간을 출력 폴더로 컴파일합니다.

또한 빌드 명령은 다음을 담당합니다:

- tsconfig-paths를 통한 매핑 경로(경로 별칭을 사용하는 경우)
- OpenAPI 데코레이터로 DTO에 주석 달기(@nestjs/swagger CLI 플러그인이 활성화된 경우)

GraphQL 데코레이터로 DTO에 주석 달기(`@nestjs/graphql` CLI 플러그인이 활성화된 경우)

인수

인수	설명
<이름>	빌드할 프로젝트의 이름입니다.

옵션

옵션	설명
`--경로 [경로]`	`tsconfig` 파일의 경로입니다. 별칭 `-p`
`--config [경로]`	`nest-cli` 구성 파일의 경로입니다. 별칭 `-c`
`--시계`	시계 모드에서 실행(실시간 새로 고침)합니다. 컴파일에 `tsc를` 사용하는 경우 rs를 입력하여 애플리케이션을 다시 시작할 수 있습니다(`수동 재시작` 옵션이 `true로` 설정됨). 별칭 `-w`
`--빌더 [이름]`	컴파일에 사용할 빌더를 지정합니다(`tsc`, `swc` 또는 `웹팩`). 별칭 `-b`
`--webpack`	컴파일에 웹팩을 사용합니다(더 이상 사용되지 않음: 대신 `--builder 웹팩` 사용).
`--웹팩 경로`	웹팩 구성 경로입니다.
`--tsc`	컴파일에 `tsc를` 강제로 사용합

니다. 중첩 시작

애플리케이션(또는 워크스페이스의 기본 프로젝트)을 컴파일하고 실행합니다.

```
nest 시작 <이름> [옵션]
```

인수

인수	설명
`<이름>`	실행할 프로젝트의 이름입니다.

옵션

옵션	설명
`--경로 [경로]`	`tsconfig` 파일의 경로입니다. 별칭 `-p`
`--config [경로]`	`nest-cli` 구성 파일의 경로입니다. 별칭 `-c`
`--시계`	시계 모드에서 실행(실시간 다시 로드) 별칭 `-w`

옵션	설명
`--빌더 [이름]`	컴파일에 사용할 빌더를 지정합니다(`tsc`, `swc` 또는 `웹팩`). 별칭 `-b`
`--보존워치 출력`	화면을 지우는 대신 오래된 콘솔 출력을 감시 모드로 유지합니다. (`TSC` 감시 모드만 해당)
`--watchAssets`	보기 모드(실시간 다시 로드)에서 실행하여 비TS 파일(에셋)을 시청합니다. 자세한 내용은 에셋을 참조하세요.
`--디버그 [호스트포트]`	디버그 모드에서 실행(--inspect 플래그 사용) 별칭 `-d`
`--웹팩`	컴파일에 웹팩을 사용합니다. (더 이상 사용되지 않음: `--builder 웹팩 사용` 대신)
`--webpackPath`	웹팩 구성 경로입니다.
`--tsc`	컴파일 시 `tsc`를 강제로 사용합니다. 실행할 바이너리(기본 값: `노드`).
`--exec [바이너리]`	별칭 `-e`

둥지 추가

네스트 라이브러리로 패키징된 라이브러리를 임포트하여 설치 스키마를 실행합니다.

```
nest 추가 <이름> [옵션]
```

인수

인수	설명
`<이름>`	가져올 라이브러리의 이름입니다.

네스트 정보

설치된 네스트 패키지에 대한 정보 및 기타 유용한 시스템 정보를 표시합니다. 예를 들어

```
둥지 정보
```

```
 _   _              _       ___  ______ ______ _      ______
| \ | |            | |     |_  |/  ___|/  __ \|     ||_  _|
|  \| |   ___  ___ | |       | |\ `--. |    /\/| || |
| . ` | / _ \/ __|| __|      | | `--. \| || || |
| |\  ||  __/\__ \| |_ /\__/ //\__/ /| \__/\| |_____| |_
```

```
\_| \_/ \___||___/ \__|\____/ \____/  \____/\_____/\___/

[시스템 정보]
OS 버전 : macOS High Sierra
NodeJS 버전 : v16.18.0 [네스트
정보] 마이크로서비스 버전 : 10.0.0
웹소켓 버전 : 10.0.0 테스트 버전 :
10.0.0 일반 버전 : 10.0.0
핵심 버전 : 10.0.0
```

Nest CLI 및 스크립트

이 섹션에서는 `nest` 명령이 컴파일러 및 스크립트와 상호 작용하는 방식에 대한 추가 배경 지식을 제공하여 DevOps 담당자가 개발 환경을 관리하는 데 도움을 줍니다.

Nest 애플리케이션은 실행하기 전에 JavaScript로 컴파일해야 하는 표준 TypeScript 애플리케이션입니다. 컴파일 단계를 수행하는 방법에는 여러 가지가 있으며, 개발자/팀은 자신에게 가장 적합한 방법을 자유롭게 선택할 수 있습니다. 이를 염두에 두고 Nest는 다음과 같은 작업을 수행하는 도구 세트를 기본으로 제공합니다:

- 명령줄에서 사용할 수 있는 표준 빌드/실행 프로세스를 제공하여 합리적인 기본값으로 '바로 작동'하도록 합니다.
- 빌드/실행 프로세스가 열려 있는지 확인하여 개발자가 기본 도구에 직접 액세스하여 기본 기능 및 옵션을 사용하여 사용자 지정할 수 있도록 합니다.
- 전체 컴파일/배포/실행 파이프라인을 개발팀이 사용하는 모든 외부 도구로 관리할 수 있도록 완전히 표준화된 TypeScript/Node.js 프레임워크를 유지합니다.

이 목표는 `nest` 명령, 로컬에 설치된 TypeScript 컴파일러, `package.json` 스크립트의 조합을 통해 달성할 수 있습니다. 아래에서 이러한 기술이 어떻게 함께 작동하는지 설명합니다. 이를 통해 빌드/실행 프로세스의 각 단계에서 어떤 일이 일어나고 있는지, 필요한 경우 해당 동작을 사용자 지정하는 방법을 이해하는 데 도움이 될 것입니다.

네스트 바이너리

`nest` 명령은 OS 레벨 바이너리입니다(즉, OS 명령줄에서 실행됨). 이 명령은 실제로 아래에 설명된 세 가지 영역을 포함합니다. 프로젝트가 스캐폴드될 때 자동으로 제공되는 `package.json` 스크립트를 통해 빌드(`nest 빌드`) 및 실행(`nest 시작`) 하위 명령을 실행하는 것이 좋습니다(`nest new`를 실행하는 대신 리포지토리를 복제하여 시작하려는 경우 타입스크립트 스타터를 참조하세요).

빌드

`nest 빌드는` 표준 `tsc` 컴파일러 또는 `swc` 컴파일러(표준 프로젝트용) 또는 `ts-loader`를 사용하는 웹팩 번들러(모노포스의 경우) 위에 래퍼를 추가하는 것입니다. 이 래퍼는 기본적으로 `tsconfig-path`를 처리하는 것

외에는 다른 컴파일 기능이나 단계를 추가하지 않습니다. 이 옵션이 존재하는 이유는 대부분의 개발자, 특히 Nest를 처음 시작하는 개발자는 때때로 까다로울 수 있는 컴파일러 옵션(예: `tsconfig.json` 파일)을 조정할 필요가 없기 때문입니다.

자세한 내용은 네스트 빌드 문서를 참조하세요.

실행

`nest start`는 프로젝트가 빌드되었는지 확인한 다음(`nest build`와 동일), 컴파일된 애플리케이션을 실행하기 위해 이식 가능한 쉬운 방법으로 `node` 명령을 호출합니다. 빌드와 마찬가지로 필요에 따라 이 프로세스를 사용자 지정할 수 있으며, `중첩 시작` 명령과 해당 옵션을 사용하거나 완전히 대체할 수 있습니다. 전체 프로세스는 표준 TypeScript 애플리케이션 빌드 및 실행 파이프라인이며, 사용자는 이 프로세스를 자유롭게 관리할 수 있습니다.

자세한 내용은 둥지 시작 문서를 참조하세요. 세대

네스트 생성 명령은 이름에서 알 수 있듯이 새 네스트 프로젝트 또는 네스트 내의 컴포넌트를 생성합니다.
그들을.

패키지 스크립트

OS 명령 수준에서 네스트 명령을 실행하려면 네스트 바이너리를 전역적으로 설치해야 합니다. 이는 npm의 표준 기능이며 Nest가 직접 제어할 수 없습니다. 이로 인한 한 가지 결과는 전역으로 설치된 네스트 바이너리가 package.json에서 프로젝트 종속성으로 관리되지 않는다는 것입니다. 예를 들어 두 명의 개발자가 서로 다른 두 가지 버전의 네스트 바이너리를 실행할 수 있습니다. 이에 대한 표준 솔루션은 패키지 스크립트를 사용하여 빌드 및 실행 단계에서 사용되는 도구를 개발 종속성으로 취급할 수 있도록 하는 것입니다.

nest new를 실행하거나 타입스크립트 스타터를 복제하면 Nest는 빌드 및 시작과 같은 명령으로 새 프로젝트의 package.json 스크립트를 채웁니다. 또한 기본 컴파일러 도구(예: 타입스크립트)를 개발 종속 요소로 설치합니다.

빌드를 실행하고 다음과 같은 명령으로 스크립트를 실행합니다:

```
$ npm 실행 빌드
```

그리고

```
$ npm 실행 시작
```

이러한 명령은 npm의 스크립트 실행 기능을 사용하여 로컬에 설치된 네스트 바이너리를 사용하여 네스트 빌드 또는 네스트 시작을 실행합니다. 이러한 기본 제공 패키지 스크립트를 사용하면 Nest CLI 명령*에 대한 종속성을 완벽하게 관리할 수 있습니다. 즉, 이 권장 사용법을 따르면 조직의 모든 구성원이 동일한 버전의 명령을 실행하도록 보장할 수 있습니다.

*이는 빌드 및 시작 명령에 적용됩니다. nest new 및 nest generate 명령은 빌드/실행 파이프라인의 일부가 아니므로 다른 컨텍스트에서 작동하며 패키지.json 스크립트가 기본으로 제공되지 않습니다.

대부분의 개발자/팀은 Nest 프로젝트를 빌드하고 실행할 때 패키지 스크립트를 활용하는 것이 좋습니다. 옵션(`--path`, `-- webpack`, `--webpackPath`)을 통해 이러한 스크립트의 동작을 완전히 사용자 정의하거나 필요에 따라 `tsc` 또는 웹팩 컴파일러 옵션 파일(예: `tsconfig.json`)을 사용자 정의할 수 있습니다. 또한 완전히 사용자 정의된 빌드 프로세스를 실행하여 TypeScript를 컴파일할 수도 있습니다(또는 `ts-node`를 사용하여 직접 TypeScript를 실행할 수도 있습니다).

이전 버전과의 호환성

Nest 애플리케이션은 순수 TypeScript 애플리케이션이므로 이전 버전의 Nest 빌드/실행 스크립트는 계속 작동합니다. 업그레이드할 필요는 없습니다. 준비가 되면 새로운 `네스트 빌드` 및 `네스트 시작` 명령을 활용하거나 이전 또는 사용자 정의 스크립트를 계속 실행하도록 선택할 수 있습니다.

마이그레이션

변경할 필요는 없지만, `tsc-watch` 또는 `ts-node와` 같은 도구를 사용하는 대신 새 CLI 명령을 사용하도록 마이그레이션할 수 있습니다. 이 경우, 글로벌 및 로컬 모두에서 최신 버전의 `@nestjs/cli를` 설치하기만 하면 됩니다 :

```
$ npm install -g @nestjs/cli
cd /일부/프로젝트/루트/폴더
$ npm install -D @nestjs/cli
```

그런 다음 `package.json에` 정의된 `스크립트를` 다음 스크립트로 바꿀 수 있습니다:

```
"build": "둥지 빌드", "
시작": "둥지 시작",
"start:dev": "nest start --watch",
"start:debug": "nest start --debug --watch",
```