

소개

Nest(NestJS)는 효율적이고 확장 가능한 [Node.js](#) 서버 측 애플리케이션을 구축하기 위한 프레임워크입니다. 이 프레임워크는 프로그래시브 [자바스크립트](#)를 사용하며, [타입스크립트](#)를 기반으로 구축되어 완벽하게 지원하지만 개발자는 순수 자바스크립트로 코딩할 수 있으며, OOP(객체지향 프로그래밍), FP(함수형 프로그래밍), FRP(함수형 반응형 프로그래밍)의 요소를 결합합니다.

내부적으로 Nest는 [Express](#)(기본값)와 같은 강력한 HTTP 서버 프레임워크를 사용하며, 선택적으로 [Fastify](#)도 사용하도록 구성할 수 있습니다!

Nest는 이러한 일반적인 Node.js 프레임워크(Express/Fastify)보다 높은 수준의 추상화를 제공할 뿐만 아니라 개발자에게 API를 직접 노출합니다. 따라서 개발자는 기본 플랫폼에서 사용할 수 있는 무수히 많은 타사 모듈을 자유롭게 사용할 수 있습니다.

철학

최근 몇 년 동안 Node.js 덕분에 JavaScript는 프론트엔드 및 백엔드 애플리케이션 모두에서 웹의 '공용어'가 되었습니다. 그 결과 개발자의 생산성을 향상시키고 빠르고 테스트 가능하며 확장 가능한 프론트엔드 애플리케이션을 만들 수 있는 [Angular](#), [React](#), [Vue](#)와 같은 멋진 프로젝트가 탄생했습니다.

하지만 노드(및 서버 측 자바스크립트)를 위한 훌륭한 라이브러리, 헬퍼, 도구가 많이 존재하지만, 그 중 어느 것도 아키텍처라는 주요 문제를 효과적으로 해결하지 못합니다.

Nest는 개발자와 팀이 고도로 테스트 가능하고 확장 가능하며 느슨하게 결합되고 쉽게 유지 관리할 수 있는 애플리케이션을 만들 수 있는 기본 애플리케이션 아키텍처를 제공합니다. 이 아키텍처는 Angular에서 많은 영감을 받았습니다.

설치

시작하려면 [Nest CLI](#)를 사용하여 프로젝트를 스캐폴드하거나 스타터 프로젝트를 복제할 수 있습니다(둘 다 동일한 결과를 생성합니다).

Nest CLI로 프로젝트를 스캐폴드하려면 다음 명령을 실행합니다. 그러면 새 프로젝트 디렉터리가 생성되고 초기 핵심 Nest 파일과 지원 모듈로 디렉터리가 채워져 프로젝트의 일반적인 기본 구조가 만들어집니다. Nest

CLI로 새 프로젝트를 생성하는 것은 처음 사용하는 사용자에게 권장됩니다. 이 방법은 [첫걸음에서](#) 계속 설명하겠습니다.

```
$ npm i -g @nestjs/cli  
nest 새 프로젝트 이름
```

정보 힌트 더 엄격한 기능 세트를 가진 새 TypeScript 프로젝트를 만들려면 `--strict` 플래그를
중첩 새 명령.

대안

또는 Git을 사용하여 TypeScript 스타터 프로젝트를 설치합니다:

```
$ git clone https://github.com/nestjs/typescript-starter.git 프로젝트  
프로젝트  
npm 설치  
$ npm 실행 시작
```

정보 힌트 git 히스토리 없이 리포지토리를 복제하려면 [degit](#)을 사용하면 됩니다.

브라우저를 열고 <http://localhost:3000/> 으로 이동합니다.

스타터 프로젝트의 JavaScript 버전을 설치하려면 위의 명령 순서대로 [javascript-starter.git](#)을 사용하세요.

npm(또는 yarn)으로 코어 및 지원 파일을 설치하여 처음부터 새 프로젝트를 수동으로 생성할 수도 있습니다. 물론 이 경우 프로젝트 상용구 파일을 직접 만들어야 합니다.

```
npm i --save @nestjs/core @nestjs/common rxjs reflect-metadata
```

첫 번째 단계

이 글에서는 Nest의 핵심 기본 사항을 배웁니다. Nest 애플리케이션의 필수 구성 요소에 익숙해지기 위해 입문 수준에서 많은 부분을 다루는 기능을 갖춘 기본 CRUD 애플리케이션을 빌드해 보겠습니다.

언어

저희는 [TypeScript](#)를 사랑하지만 무엇보다도 [Node.js](#)를 사랑합니다. 그렇기 때문에 Nest는 TypeScript와 순수 JavaScript 모두와 호환됩니다. Nest는 최신 언어 기능을 활용하므로 바닐라 자바스크립트와 함께 사용하려면 [바벨](#) 컴파일러가 필요합니다.

제공하는 예제에서는 대부분 TypeScript를 사용하지만, 코드 스니펫을 언제든지 바닐라 JavaScript 구문으로 전환할 수 있습니다(각 스니펫의 오른쪽 상단에 있는 언어 전환 버튼을 클릭하기만 하면 됩니다).

전제 조건

운영 체제에 [Node.js](#)(버전 ≥ 16)가 설치되어 있는지 확인하세요. 설정

Nest CLI를 사용하면 새 프로젝트를 설정하는 것이 매우 간단합니다. [npm](#)이 설치되어 있으면 새 Nest를 만들 수 있습니다.

프로젝트에 다음 명령을 실행합니다:

```
$ npm i -g @nestjs/cli  
nest 새 프로젝트 이름
```

정보 힌트 TypeScript의 더 엄격한 기능 세트를 사용하여 새 프로젝트를 만들려면 [nest new](#) 명령에 [-strict](#) 플래그를 전달하세요.

[프로젝트 이름](#) 디렉터리가 생성되고, 노드 모듈과 몇 가지 상용구 파일이 설치되며, [src/](#) 디렉터리가 생성되어 몇 가지 핵심 파일로 채워집니다.

src
app.controller.spec.ts
app.controller.ts
app.module.ts
app.service.ts
main.ts

다음은 이러한 핵심 파일에 대한 간략한 개요입니다:

`app.controller.ts` 단일 경로를 가진 기본 컨트롤러입니다.

`app.controller.spec.ts` 컨트롤러에 대한 단위 테스트입니다.

app.module.ts	애플리케이션의 루트 모듈입니다.
app.service.ts	단일 메서드를 사용하는 기본 서비스입니다.
main.ts	핵심 기능을 사용하는 애플리케이션의 엔트리 파일입니다.
	NestFactory를 사용하여 Nest 애플리케이션 인스턴스를 생성합니다.

main.ts에는 애플리케이션을 부트스트랩하는 비동기 함수가 포함되어 있습니다:

@@파일명(메인)

```
'@nestjs/core'에서 { NestFactory }를 임포트하고,  
'./app.module'에서 { AppModule }을 임포트합니다;
```

비동기 함수 부트스트랩()

```
const app = await NestFactory.create(AppModule);  
await app.listen(3000);  
}
```

부트스트랩(); @@스위

치

```
'@nestjs/core'에서 { NestFactory }를 임포트하고,  
'./app.module'에서 { AppModule }을 임포트합니다;
```

비동기 함수 부트스트랩()

```
const app = await NestFactory.create(AppModule);  
await app.listen(3000);  
}
```

부트스트랩();

Nest 애플리케이션 인스턴스를 생성하기 위해 핵심 `NestFactory` 클래스를 사용합니다. `NestFactory`는 애플리케이션 인스턴스를 생성할 수 있는 몇 가지 정적 메서드를 노출합니다. `create()` 메서드는 `INestApplication` 인터페이스를 충족하는 애플리케이션 객체를 반환합니다. 이 객체는 다음 장에서 설명하는 메서드 집합을 제공합니다. 위의 `main.ts` 예시에서는 애플리케이션이 인바운드 HTTP 요청을 기다릴 수 있도록 HTTP 리스너를 시작하기만 하면 됩니다.

Nest CLI로 스캐폴드된 프로젝트는 개발자가 각 모듈을 전용 디렉터리에 보관하는 관례를 따르도록 권장하는 초기 프로젝트 구조를 생성합니다.

정보 힌트 기본적으로 애플리케이션을 생성하는 동안 오류가 발생하면 앱은 코드 1과 함께 종료됩니다. 대신 오류를 발생시키려면 `abortOnError` 옵션을 비활성화하세요(예:

```
NestFactory.create(AppModule, {{ '{' }} abortOnError: false {{ '}' }})).
```

플랫폼

Nest는 플랫폼에 구애받지 않는 프레임워크를 지향합니다. 플랫폼 독립성은 개발자가 여러 유형의 애플리케이션에서 활용할 수 있는 재사용 가능한 논리적 부분을 생성할 수 있게 해줍니다. 기술적으로 Nest는 어댑터를 생성하면 모든 Node HTTP 프레임워크에서 작동할 수 있습니다. 기본적으로 지원되는 HTTP 플랫폼은 익스프레스와 [패스트파이의](#) 두 가지입니다. 필요에 가장 적합한 것을 선택할 수 있습니다.

플랫폼 익스프레스	Express는 잘 알려진 미니멀리즘 노드용 웹 프레임워크입니다. 커뮤니티에서 구현한 많은 리소스가 포함된 실전 테스트를 거친, 프로덕션에 바로 사용할 수 있는 라이브러리입니다.
플랫폼 - 패스트파이브	기본적으로 @nestjs/platform-express 패키지가 사용됩니다. 많은 사용자가 Express를 잘 사용하고 있으며, 이를 활성화하기 위해 별도의 조치를 취할 필요가 없습니다.
스트파이브	Fastify는 최대한의 효율성과 속도를 제공하는 데 중점을 둔 고성능, 낮은 오버헤드 프레임워크입니다. 여기에서 사용 방법을 읽어보세요.

어떤 플랫폼을 사용하든 자체 애플리케이션 인터페이스를 노출합니다. 이는 각각 다음과 같이 표시됩니다.

`NestExpressApplication` 및 `NestFastifyApplication`.

아래 예시처럼 `NestFactory.create()` 메서드에 유형을 전달하면 `앱` 객체에 해당 특정 플랫폼에서만 사용할 수 있는 메서드가 생깁니다. 그러나 실제로 기본 플랫폼 API에 액세스하려는 경우가 아니라면 유형을 지정할 필요가 없습니다.

```
const app = await NestFactory.create<NestExpressApplication>(AppModule);
```

애플리케이션 실행

설치 프로세스가 완료되면 OS 명령 프롬프트에서 다음 명령을 실행하여 인바운드 HTTP 요청을 수신하는 애플리케이션을 시작할 수 있습니다:

```
$ npm 실행 시작
```

정보 힌트 개발 프로세스 속도를 높이려면(빌드 속도가 20배 빨라짐) 다음과 같이 `시작` 스크립트에 `-b swc` 플래그를 전달하여 `SWC` 빌더를 사용할 수 있습니다.

이 명령은 `src/main.ts` 파일에 정의된 포트에서 수신 대기 중인 HTTP 서버로 앱을 시작합니다. 애플리케이션이 실행되면 브라우저를 열고 `http://localhost:3000/`로 이동합니다. `Hello World!` 메시지가 표시됩니다.

파일의 변경 사항을 확인하려면 다음 명령을 실행하여 애플리케이션을 시작하면 됩니다:

```
$ npm 실행 시작:dev
```

이 명령은 파일을 감시하여 자동으로 서버를 다시 컴파일하고 다시 로드합니다. 린팅 및 서

식 지정

CLI는 대규모로 안정적인 개발 워크플로우를 구축하기 위해 최선의 노력을 기울입니다. 따라서 생성된 네스트 프로젝트에는 코드 린터와 포매터가 모두 사전 설치되어 있습니다(각각 eslint와 prettier).

정보 힌트 포매터와 린터의 역할에 대해 잘 모르시나요? [여기에서](#) 차이점을 알아보세요.

안정성과 확장성을 극대화하기 위해 기본 eslint와 더 예쁜 cli 패키지를 사용합니다. 이 설정은 설계상 공식 확장 프로그램과 깔끔한 IDE 통합을 가능하게 합니다.

IDE가 적합하지 않은 헤드리스 환경(지속적 통합, Git 후크 등)의 경우 Nest 프로젝트에는 바로 사용할 수 있는 npm 스크립트가 함께 제공됩니다.

```
# 보푸라기 및 보푸라기 자동 수정 기능(에슬린트)
$ npm 실행 린트

# 더 예쁘게 포맷
$ npm 실행 형식
```

컨트롤러

컨트롤러는 들어오는 요청을 처리하고 클라이언트에 응답을 반환할 책임이 있습니다.



컨트롤러의 목적은 애플리케이션에 대한 특정 요청을 수신하는 것입니다. 라우팅 메커니즘은 어떤 컨트롤러가 어떤 요청을 수신할지 제어합니다. 각 컨트롤러에는 둘 이상의 경로가 있는 경우가 많으며, 경로마다 다른 작업을 수행할 수 있습니다.

기본 컨트롤러를 생성하기 위해 클래스와 데코레이터를 사용합니다. 데코레이터는 클래스를 필수 메타데이터와 연결하고 Nest가 라우팅 맵을 생성(요청을 해당 컨트롤러에 연결)할 수 있도록 합니다.

정보 힌트 [유효성 검사](#) 기능이 내장된 CRUD 컨트롤러를 빠르게 생성하려면 CLI의 [CRUD 생성기](#)(`nest g resource [name]`)를 사용할 수 있습니다.

라우팅

다음 예제에서는 기본 컨트롤러를 정의하는 데 필요한 `@Controller()` 데코레이터를 사용하겠습니다. 선택적 경로 경로 접두사 `cats`를 지정하겠습니다. `컨트롤러()` 데코레이터에 경로 접두사를 사용하면 관련 경로 집합을 쉽게 그룹화하고 반복되는 코드를 최소화할 수 있습니다. 예를 들어 고양이 엔티티와의 상호 작용을 관리하는 일련의 경로를 `/cats` 경로 아래에 그룹화할 수 있습니다. 이 경우 `@Controller()` 데코레이터에 경로 접두사 `cats`를 지정하면 파일의 각 경로에 대해 해당 경로 부분을 반복할 필요가 없습니다.

```
@@파일명(cats.controller)
'@nestjs/common'에서 { Controller, Get }을 가져옵니다;

@Controller('cats')
내보내기 클래스 CatsController {
  @Get()
  findAll(): 문자열 {
    반환 '이 액션은 모든 고양이를 반환합니다';
  }
}
@@switch
'@nestjs/common'에서 { Controller, Get }을 가져옵니다;

@Controller('cats')
내보내기 클래스 CatsController {
  @Get()
  findAll() {
    반환 '이 액션은 모든 고양이를 반환합니다';
```

정보 힌트 CLI를 사용하여 컨트롤러를 만들려면 `$ nest g 컨트롤러 [이름]`을 실행하기만 하면 됩니다.

명령을 사용합니다.

`findAll()` 메서드 앞에 `@Get()` HTTP 요청 메서드 데코레이터를 사용하면 Nest가 HTTP 요청의 특정 엔드포인트에 대한 핸들러를 생성하도록 지시합니다. 엔드포인트는 HTTP 요청 메서드(이 경우 GET)와 경로 경로에 해당합니다. 경로 경로란 무엇인가요? 핸들러의 경로 경로는 컨트롤러에 대해 선언된 (선택 사항) 접두사와 메서드의 데코레이터에 지정된 경로를 연결하여 결정됩니다. 모든 경로에 접두사('cats')를 선언했고 데코레이터에 경로 정보를 추가하지 않았으므로 Nest는 `GET /cats` 요청을 이 핸들러에 매핑합니다. 앞서 언급했듯이 경로에는 선택적 컨트롤러 경로 접두사와 요청 메서드 데코레이터에 선언된 경로 문자열이 모두 포함됩니다. 예를 들어, 경로 접두사 `cats`와 데코레이터 `@Get('breed')`를 결합하면 `GET /cats/breed`와 같은 요청에 대한 경로 매핑이 생성됩니다.

위의 예시에서 이 엔드포인트로 GET 요청이 이루어지면 Nest는 요청을 사용자 정의 `findAll()` 메서드로 라우팅합니다. 여기서 선택한 메서드 이름은 완전히 임의적이라는 점에 유의하세요. 경로를 바인딩할 메서드를 선언해야 하지만 Nest는 선택한 메서드 이름에 어떤 의미도 부여하지 않습니다.

이 메서드는 200 상태 코드와 관련 응답(이 경우 문자열)을 반환합니다. 왜 이런 일이 발생할까요? 이를 설명하기 위해 먼저 Nest가 응답을 조작하기 위해 두 가지 다른 옵션을 사용한다는 개념을 소개하겠습니다:

표준(권장) 이 기본 제공 메서드를 사용하면 요청 핸들러가 JavaScript 객체 또는 배열을 반환할 때 자동으로 JSON으로 직렬화됩니다. 그러나 JavaScript 기본 유형(예: 문자열, 숫자, 부울)을 반환하는 경우 Nest는 직렬화를 시도하지 않고 값만 전송합니다. 따라서 응답 처리가 간단해집니다. 값만 반환하면 나머지는 Nest가 알아서 처리합니다.

라이브러리별 또한 응답의 상태 코드는 201을 사용하는 POST 요청을 제외하고는 기본적으로 항상 200입니다. 핸들러 수준에서 `@HttpCode(...)` 데코레이터를 추가하여 이 동작을 쉽게 변경할 수 있습니다([상태 코드 참조](#)).

라이브러리별(예: Express) 응답 객체를 사용할 수 있으며, 메서드 핸들러 시그니처에 `@Res()` 데코레이터를 사용하여 삽입할 수 있습니다(예: `findAll(@Res() 응답)`). 이 접근 방식을 사용하면 해당 객체에 의해 노출된 기본 응답 처리 메서드를 사용할 수 있습니다. 예를 들어 Express를 사용하면 `response.status(200).send()` 같은 코드를 사용하여 응답을 구성할 수 있습니다.

경고 네스트는 핸들러가 `@Res()` 또는 `@Next()`를 사용하는 경우 라이브러리별 옵션을 선택했음을 나타내는 경고를 감지합니다. 두 접근 방식을 동시에 사용하면 이 단일 경로에 대해 표준 접근 방식이 자동으로 비활성화되고 더 이상 예상대로 작동하지 않습니다. 두 가지 접근 방식을 동시에 사용하려면(예: 응답 객체를 삽입하여 쿠키/헤더만 설정하고 나머지는 프레임워크에 맡기는 경우) `@Res({{ ' ' }} 패스스루: true {{ ' ' }})` 데코레이터에서 `패스스루` 옵션을 `true`로 설정해야 합니다.

요청 개체

핸들러는 종종 클라이언트 요청 세부 정보에 액세스해야 합니다. Nest는 기본 플랫폼(기본적으로 Express)의 [요청 객체에](#) 대한 액세스를 제공합니다. 핸들러의 서명에 `@Req()` 데코레이터를 추가하여 Nest에 요청 객체를 주입하도록 지시하면 요청 객체에 액세스할 수 있습니다.

```

@@파일명(cats.controller)

'@nestjs/common'에서 { Controller, Get, Req }를 가져오고,
'express'에서 { Request }를 가져옵니다;

@Controller('cats')
내보내기 클래스 CatsController {
  @Get()
  findAll(@Req() request: 요청): 문자열 { 반환 '이
    액션은 모든 고양이를 반환합니다';
  }
}
@@switch
'@nestjs/common'에서 { Controller, Bind, Get, Req }를 가져옵니다;

@Controller('cats')
내보내기 클래스 CatsController {
  @Get()
  @Bind(Req())
  findAll(request) {
    반환 '이 액션은 모든 고양이를 반환합니다';
  }
}

```

정보 힌트 위의 `요청: 요청` 매개변수 예제에서와 같이 [익스프레스](#) 타이핑을 활용하려면 `@types/express` 패키지를 설치하세요.

요청 객체는 HTTP 요청을 나타내며 요청 쿼리 문자열, 매개변수, HTTP 헤더 및 본문에 대한 속성을 가지고 있습니다(자세한 내용은 [여기를 참조하세요](#)). 대부분의 경우 이러한 속성을 수동으로 가져올 필요는 없습니다. 대신 바로 사용할 수 있는 `@Body()` 또는 `@Query()`와 같은 전용 데코레이터를 사용할 수 있습니다. 아래는 제공되는 데코레이터와 이들이 나타내는 일반 플랫폼별 객체 목록입니다.

`요청()`, `@요청()` 요청 `@응답()`, `@응`

`답()`, `@Res()`* `res` `@다음()` 다음

`세션()` `req.session`

`Param(키?: 문자열)` `req.params`/`req.params[키]`

`@Body(키?: 문자열)` `req.body`/`req.body[키]` `@Query(`

`키?: 문자열)` `req.query`/`req.query[키]`

@Headers(이름?: 문자열) req.headers / req.headers[이름]

@Ip() req.ip

호스트 파라미터() req.hosts

* 기본 HTTP 플랫폼(예: Express 및 Fastify)에서의 타이핑과의 호환성을 위해 Nest는 `@Res()` 및 `@Response()` 데코레이터를 제공합니다. `Res()`는 `@Response()`의 별칭일 뿐입니다. 둘 다 기본 네이티브 플랫폼 응답 객체 인터페이스를 직접 노출합니다. 이 두 데코레이터를 사용할 때는 기본 라이브러리의 타이핑(예: `@types/express`)도 가져와야 최대한 활용할 수 있습니다. 메서드 핸들러에 `@Res()` 또는 `@Response()`를 삽입하면 해당 핸들러에 대해 Nest를 라이브러리 전용 모드로 전환하고 응답을 관리할 책임이 있다는 점에 유의하세요. 이 경우 응답 객체(예: `res.json(...)` 또는 `res.send(...)`)를 호출하여 어떤 종류의 응답을 발행해야 하며, 그렇지 않으면 HTTP 서버가 중단됩니다.

정보 힌트 나만의 맞춤 데코레이터를 만드는 방법을 알아보려면 [이](#) 장을 참조하세요.

리소스

앞서 고양이 리소스를 가져오는 엔드포인트(GET 경로)를 정의했습니다. 일반적으로 새 레코드를 생성하는 엔드포인트도 제공해야 합니다. 이를 위해 POST 핸들러를 만들어 보겠습니다:

```
@@파일명(cats.controller)
'@nestjs/common'에서 { Controller, Get, Post }를 가져옵니다;

@Controller('cats')
내보내기 클래스 CatsController {
  @Post()
  create(): 문자열 {
    '이 동작은 새 고양이를 추가합니다'를 반환합니다;
  }

  @Get()
  findAll(): 문자열 {
    반환 '이 액션은 모든 고양이를 반환합니다';
  }
}

@@switch
'@nestjs/common'에서 { Controller, Get, Post }를 가져옵니다;

@Controller('cats')
내보내기 클래스 CatsController {
  @Post()
  create() {
    '이 동작은 새 고양이를 추가합니다'를 반환합니다;
  }

  @Get()
  findAll() {
    반환 '이 액션은 모든 고양이를 반환합니다';
  }
}
```

아주 간단합니다. Nest는 모든 표준 HTTP 메서드에 대한 데코레이터를 제공합니다: `Get()`, `@Post()`, `@Put()`, `@Delete()`, `@Patch()`, `@Options()`, `@Head()` 등입니다. 또한 `@All()`은 엔드포인트를 정의합니다.

를 사용하여 모든 것을 처리합니

다. 경로 와일드카드

패턴 기반 경로도 지원됩니다. 예를 들어 별표는 와일드카드로 사용되며 다음과 일치합니다.
문자를 조합할 수 있습니다.

```
@Get('ab*cd')
findAll() {
  '이 경로는 와일드카드를 사용합니다'를 반환합니다;
}
```

'ab*cd' 경로 경로는 abcd, ab_cd, abecd 등과 일치합니다. 문자 ?, +, * 및 ()는 경로 경로에 사용할 수 있으며 정규식 대응 문자의 하위 집합입니다. 하이픈(-)과 점(.)은 문자열 기반 경로에서 문자 그대로 해석됩니다.

경고 경고 경로 중간에 와일드카드는 익스프레스에서만 지원됩니다.

상태 코드

앞서 언급했듯이 응답 상태 코드는 기본적으로 항상 200이며, 다음과 같은 POST 요청은 제외됩니다.

201. 핸들러 수준에서 @HttpCode(...) 데코레이터를 추가하여 이 동작을 쉽게 변경할 수 있습니다.

```
Post()
@HttpCode(204)
create() {
  '이 동작은 새 고양이를 추가합니다'를 반환합니다;
}
```

정보 힌트 @nestjs/common 패키지에서 HttpStatusCode를 가져옵니다.

상태 코드는 정적이지 않고 다양한 요인에 따라 달라지는 경우가 많습니다. 이 경우 라이브러리별 응답(@Res())
를 사용하여 삽입) 객체를 사용할 수 있습니다(또는 오류 발생 시 예외를 던지세요).

헤더

사용자 지정 응답 헤더를 지정하려면 @Header() 데코레이터 또는 라이브러리별 응답 객체를 사용하거나
나 res.header()를 직접 호출하면 됩니다.

```
@Post()
@Header('Cache-Control', 'none')
create() {
    '이 동작은 새 고양이를 추가합니다'를 반환합니다;
}
```

정보 힌트 [@nestjs/common](#) 패키지에서 헤더를 가져옵니다.

리디렉션

응답을 특정 URL로 리디렉션하려면 [@Redirect\(\)](#) 데코레이터 또는 라이브러리별 응답 객체를 사용하거나 [res.redirect\(\)](#)를 직접 호출할 수 있습니다.

[리디렉션\(\)](#)은 두 개의 인자, [url](#)과 [statusCode](#)를 사용하며 둘 다 선택 사항입니다. 기본값은 상태 코드는 생략된 경우 [302\(발견됨\)](#)입니다.

```
Get()
@Redirect('https://nestjs.com', 301)
```

때때로 HTTP 상태 코드 또는 리디렉션 URL을 동적으로 확인하고 싶을 수 있습니다. 이를 위해서는 라우트 핸들러 메서드에서 셰이프와 함께 객체를 반환하면 됩니다:

```
{
  "url": 문자열,
  "statusCode": 숫자
}
```

반환된 값은 [@Redirect\(\)](#) 데코레이터에 전달된 모든 인수를 재정의합니다. 예를 들어

```
Get('docs')
@Redirect('https://docs.nestjs.com', 302)
getDocs(@Query('version') version) {
  if (version && version === '5') {
    반환 { url: 'https://docs.nestjs.com/v5/' };
  }
}
```

경로 매개변수

정적 경로가 있는 경로는 요청의 일부로 동적 데이터를 받아들여야 할 때 작동하지 않습니다(예: ID가 [1인](#) 고양 이를 가져오기 위한 [GET /cats/1](#)). 매개변수가 있는 경로를 정의하기 위해 경로 경로에 경로 매개변수 토큰을 추가하여 요청 URL의 해당 위치에서 동적 값을 캡처할 수 있습니다. 아래 [@Get\(\)](#) 데코레이터 예시의 경로 매개

변수 토큰은 이 사용법을 보여줍니다. 이러한 방식으로 선언된 경로 매개변수는 메서드 서명에 추가해야 하는

`@Param()` 데코레이터를 사용하여 액세스할 수 있습니다.

정보 힌트 매개변수가 있는 경로는 모든 정적 경로 뒤에 선언해야 합니다. 이렇게 하면 매개변수화된 경로가

정적 경로로 향하는 트래픽을 가로채는 것을 방지할 수 있습니다.

```

@@파일명()
@Get(':id')
findOne(@Param() params: any): string {
  console.log(params.id);
  반환 `이 함수는 #${params.id} cat을 반환합니다`;
}

@@switch
@Get(':id')
@Bind(Param())
findOne(params) {
  console.log(params.id);
  반환 `이 함수는 #${params.id} cat을 반환합니다`;
}

```

`Param()`은 메소드 매개변수(위 예시의 `매개변수`)를 장식하는 데 사용되며, 경로 매개변수를 메소드 본문 내에서 장식된 메소드 매개변수의 속성으로 사용할 수 있게 합니다. 위 코드에서 볼 수 있듯이 `params.id`를 참조하여 `id` 매개변수에 액세스할 수 있습니다. 특정 매개변수 토큰을 데코레이터에 전달한 다음 메서드 본문에서 이름으로 직접 경로 매개변수를 참조할 수도 있습니다.

정보 힌트 `@nestjs/common` 패키지에서 `매개변수` 가져오기.

```

@@파일명()
@Get(':id')
findOne(@Param('id') id: 문자열): 문자열 { return
  `이 작업은 #${id} cat`을 반환합니다;
}

@@switch
@Get(':id')
@Bind(Param('id'))
findOne(id) {
  반환 `이 함수는 #${id} 고양이를 반환합니다`;
}

```

하위 도메인 라우팅

컨트롤러 데코레이터는 `호스트` 옵션을 사용하여 들어오는 요청의 HTTP 호스트가 특정 값과 일치하도록 요구할 수 있습니다.

```
컨트롤러({ 호스트: 'admin.example.com' }) 내보내기

클래스 AdminController {
    @Get()
    index(): 문자열 { '관리자
        페이지'를 반환합니다;
    }
}
```

경고 Fastify는 중첩 라우터를 지원하지 않으므로 하위 도메인 라우팅을 사용할 때는 (기본값) Express 어댑터를 대신 사용해야 합니다.

경로 경로와 마찬가지로 호스트 옵션은 토큰을 사용하여 호스트 이름에서 해당 위치의 동적 값을 캡처할 수 있습니다. 아래 `@Controller()` 데코레이터 예제의 호스트 매개변수 토큰은 이 사용법을 보여줍니다. 이러한 방식으로 선언된 호스트 매개변수는 메서드 시그니처에 추가해야 하는 `@HostParam()` 데코레이터를 사용하여 액세스할 수 있습니다.

```
컨트롤러({ 호스트: ':account.example.com' }) 내보내  
기 클래스 AccountController {  
  @Get()  
  getInfo(@HostParam('account') account: string) { 반환  
    계정;  
  }  
}
```

범위

다른 프로그래밍 언어 배경을 가진 사람들에게는 Nest에서 거의 모든 것이 들어오는 요청에서 공유된다는 사실이 의외로 느껴질 수 있습니다. 데이터베이스에 대한 연결 풀, 전역 상태를 가진 싱글톤 서비스 등이 있습니다. Node.js는 모든 요청이 별도의 스레드에서 처리되는 요청/응답 다중 스레드 상태 비저장 모델을 따르지 않는다는 점을 기억하세요. 따라서 싱글톤 인스턴스를 사용하는 것은 애플리케이션에 완전히 안전합니다.

그러나 GraphQL 애플리케이션의 요청별 캐싱, 요청 추적 또는 멀티테넌시와 같이 컨트롤러의 요청 기반 수명이 원하는 동작일 수 있는 예외적인 경우가 있습니다. [여기에서](#) 범위를 제어하는 방법을 알아보세요.

비동기성

우리는 최신 JavaScript를 사랑하며 데이터 추출이 대부분 비동기식이라는 것을 알고 있습니다. 그렇기 때문에 Nest는 비동기 함수를 지원하고 잘 작동합니다.

정보 힌트 비동기/대기 기능에 대한 자세한 내용은 [여기를 참조하세요](#).

모든 비동기 함수는 [프로미스를](#) 반환해야 합니다. 즉, Nest가 자체적으로 해결할 수 있는 지연된 값을 반환할 수 있습니다. 이에 대한 예를 살펴보겠습니다:

```
@@파일명(cats.controller)
@Get()
비동기 findAll(): Promise<any[]> {
    return [];
}

@@스위치
@Get()
async findAll() {
    return [];
}
```

위의 코드는 완전히 유효합니다. 또한 Nest 경로 핸들러는 RxJS 관찰 가능한 스트림을 반환할 수 있어 훨씬 더 강력합니다. Nest는 자동으로 아래 소스를 구독하고 (스트림이 완료되면) 마지막으로 방출된 값을 가져옵니다.

```
@@파일명(cats.controller)
@Get()
findAll(): Observable<any>[] {
  return of([]);
}
@@switch
@Get()
findAll() {
  의 반환([]);
}
```

위의 두 가지 방법 모두 작동하며 요구 사항에 맞는 방법을 사용할 수 있습니다. 페이로드

요청

이전 예제에서 POST 라우트 핸들러는 클라이언트 매개변수를 허용하지 않았습니다. 여기에 `@Body()` 데코레이터를 추가하여 이 문제를 해결해 보겠습니다.

하지만 먼저 (TypeScript를 사용하는 경우) DTO(데이터 전송 객체) 스키마를 결정해야 합니다. DTO는 데이터가 네트워크를 통해 전송되는 방식을 정의하는 객체입니다. 타입스크립트 인터페이스를 사용하거나 간단한 클래스를 사용하여 DTO 스키마를 결정할 수 있습니다. 흥미롭게도 여기서는 클래스를 사용하는 것이 좋습니다. 그 이유는 무엇일까요? 클래스는 JavaScript ES6 표준의 일부이므로 컴파일된 JavaScript에서 실제 엔티티로 보존됩니다. 반면에 TypeScript 인터페이스는 트랜스파일링 중에 제거되므로 Nest는 런타임에 이를 참조할 수 없습니다. 이는 파이프와 같은 기능이 런타임에 변수의 메타타입에 액세스할 수 있을 때 추가적인 가능성을 가능하게 하기 때문에 중요합니다.

`CreateCatDto` 클래스를 만들어 보겠습니다:

```
@@파일명(create-cat.dto) 내보내기
```

```
클래스 CreateCatDto {  
    이름: 문자열; 나이  
    : 숫자; 품종: 문  
    자열;  
}
```

기본 속성은 세 가지뿐입니다. 그런 다음 새로 생성된 DTO를

CatsController:

```
@@파일명(cats.controller)  
@Post()  
async create(@Body() createCatDto: CreateCatDto) {
```

```
'이 동작은 새 고양이를 추가합니다'를 반환합니다;  
}  
스위치 @포스트()  
@바인드(본문())  
async create(createCatDto) {  
    '이 동작은 새 고양이를 추가합니다'를 반환합니다;  
}
```

정보 힌트 `ValidationPipe`는 메서드 핸들러가 수신해서는 안 되는 프로퍼티를 필터링할 수 있습니다. 이 경우 허용 가능한 속성을 화이트리스트에 추가할 수 있으며, 화이트리스트에 포함되지 않은 속성은 결과 객체에서 자동으로 제거됩니다. `CreateCatDto` 예제에서 화이트리스트는 `이름`, `나이`, `품종` 속성입니다. [여기에서](#) 자세히 알아보세요.

오류 처리

여기에는 오류 처리(즉, 예외 작업)에 대한 별도의 장이 있습니다. 전체 리소스 샘플

아래는 사용 가능한 여러 데코레이터를 사용하여 기본 컨트롤러를 만드는 예제입니다. 이 컨트롤러는 내부 데이터에 액세스하고 조작하는 몇 가지 메서드를 노출합니다.

@@파일명 (cats.controller)

'@nestjs/common'에서 { Controller, Get, Query, Post, Body, Put, Param, Delete }를 임포트합니다;

'./dto'에서 { CreateCatDto, UpdateCatDto, ListAllEntities }를 가져옵니다;

@Controller('cats')

내보내기 클래스 CatsController {

@Post()

create(@Body() createCatDto: CreateCatDto) {

 return '이 액션은 새 고양이를 추가합니다';

}

@Get()

findAll(@Query() 쿼리: ListAllEntities) {

 반환 `이 함수는 모든 고양이를 반환합니다 (제한: \${query.limit} 항목)`;

}

@Get(':id')

findOne(@Param('id') id: string) {

 반환 `이 함수는 #\${id} 고양이를 반환합니다`;

}

@Put(':id')

update(@Param('id') id: 문자열, @Body() updateCatDto: UpdateCatDto) { 반

환 `이 작업은 #\${id} 고양이를 업데이트합니다`;

}

```
삭제('/:id') remove(@Param('id'))  
  id: 문자열) {  
    반환 `이 작업은 #${id} 고양이를 제거합니다`;  
  }  
}  
@@switch  
'@nestjs/common'에서 { Controller, Get, Query, Post, Body, Put, Param,  
Delete, Bind }를 가져옵니다;  
  
@Controller('cats')  
내보내기 클래스 CatsController {  
  @Post()  
  @Bind(Body())  
  create(createCatDto) {  
    '이 동작은 새 고양이를 추가합니다'를 반환합니다;  
  }  
  
  Get()  
  @Bind(Query())  
  findAll(query) {  
    콘솔 로그(쿼리);  
    반환 `이 함수는 모든 고양이를 반환합니다 (제한: ${query.limit} 항목)`;  
  }  
  
  @Get(':id')  
  @Bind(Param('id'))  
  findOne(id) {  
    반환 `이 함수는 #${id} 고양이를 반환합니다`;  
  }  
  
  Put(':id')  
  @Bind(Param('id'), Body())  
  update(id, updateCatDto) {  
    반환 `이 작업은 #${id} 고양이를 업데이트합니다`;  
  }  
  
  @Delete(':id')  
  @Bind(Param('id'))  
  remove(id) {  
    반환 `이 작업은 #${id} 고양이를 제거합니다`;  
  }  
}
```

정보 힌트 네스트 CLI는 모든 상용구 코드를 자동으로 생성하는 제너레이터(회로도)를 제공하여 이 모든 작업을 피하고 개발자 환경을 훨씬 더 간단하게 만들 수 있도록 도와줍니다. 이 기능에 대한 자세한 내용은

[여기에서 확인하세요.](#)

시작 및 실행하기

위의 컨트롤러가 완전히 정의되었지만 Nest는 여전히 CatsController가 존재한다는 사실을 알지 못하므로 이 클래스의 인스턴스를 생성하지 않습니다.

컨트롤러는 항상 모듈에 속하기 때문에 `@Module()` 데코레이터 안에 컨트롤러 배열을 포함시킵니다. 루트 `AppModule`을 제외한 다른 모듈을 아직 정의하지 않았으므로 이를 사용하여 `CatsController`를 소개하겠습니다:

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./cats/cats.controller'에서 { CatsController }를 가져옵니다;

모듈({  
    컨트롤러: [CatsController],  
})  
내보내기 클래스 AppModule {}
```

`모듈()` 데코레이터를 사용하여 모듈 클래스에 메타데이터를 첨부했고, 이제 Nest에서 어떤 컨트롤러를 마운트해야 하는지 쉽게 반영할 수 있습니다.

라이브러리별 접근 방식

지금까지 Nest의 표준 응답 조작 방법에 대해 설명했습니다. 응답을 조작하는 두 번째 방법은 라이브러리별 `응답 객체`를 사용하는 것입니다. 특정 응답 객체를 삽입하려면 `@Res()` 데코레이터를 사용해야 합니다. 차이점을 보여주기 위해 `CatsController`를 다음과 같이 다시 작성해 보겠습니다:

@@파일명()

'@nestjs/common'에서 { Controller, Get, Post, Res, HttpStatus }를 가져오고,
'express'에서 { Response }를 가져옵니다;

@Controller('cats')

내보내기 클래스 CatsController {

@Post()

```
create(@Res() res: Response) {
    res.status(HttpStatus.CREATED).send();
}
```

@Get()

```
findAll(@Res() res: Response) {
    res.status(HttpStatus.OK).json([]);
}
```

}

@@switch

'@nestjs/common'에서 { Controller, Get, Post, Bind, Res, Body, HttpStatus }를
임포트합니다;

@Controller('cats')

내보내기 클래스 CatsController {

@Post()

```
@Bind(Res(), Body())
create(res, createCatDto) {
    res.status(HttpStatus.CREATED).send();
```

```

    }

    Get()
    @Bind(Res())
    findAll(res) {
        res.status(HttpStatus.OK).json([]);
    }
}

```

이 접근 방식이 효과가 있고 실제로 응답 객체에 대한 완전한 제어(헤더 조작, 라이브러리별 기능 등)를 제공함으로써 어떤 면에서는 더 많은 유연성을 허용하지만, 신중하게 사용해야 합니다. 일반적으로 이 접근 방식은 훨씬 덜 명확하며 몇 가지 단점이 있습니다. 가장 큰 단점은 코드가 플랫폼에 따라 달라지고(기본 라이브러리마다 응답 객체에 대한 API가 다를 수 있으므로) 테스트하기가 더 어려워진다는 점입니다(응답 객체를 모킹해야 하는 등).

또한 위의 예제에서는 인터셉터 및 `@HttpCode()` / `@Header()` 데코레이터와 같이 Nest 표준 응답 처리에 의존하는 Nest 기능과의 호환성을 잃게 됩니다. 이 문제를 해결하려면 다음과 같이 `패스스루` 옵션을 `true`로 설정하면 됩니다:

```

@@파일명() @Get()
findAll(@Res({ passthrough: true }) res: Response) {
    res.status(HttpStatus.OK);
    반환 [];
}

@@스위치
@Get()
@Bind(Res({ 패스스루: true }))
findAll(res) {
    res.status(HttpStatus.OK);
    return [];
}

```

이제 기본 응답 객체와 상호 작용할 수 있지만(예: 특정 조건에 따라 쿠키 또는 헤더 설정) 나머지는 프레임워크에 맡기세요.

모듈

모듈은 `@Module()` 데코레이터로 주석이 달린 클래스입니다. `모듈()` 데코레이터는 Nest가 애플리케이션 구조를 구성하는 데 사용하는 메타데이터를 제공합니다.



각 애플리케이션에는 하나 이상의 모듈, 즉 루트 모듈이 있습니다. 루트 모듈은 Nest가 애플리케이션 그래프를 구축하는 데 사용하는 시작점이며, Nest가 모듈과 공급자 관계 및 종속성을 해결하는 데 사용하는 내부 데이터 구조입니다. 아주 작은 애플리케이션에는 이론적으로 루트 모듈만 있을 수 있지만, 일반적인 경우는 아닙니다. 모듈은 컴포넌트를 효과적으로 구성하는 방법으로 강력히 권장됩니다. 따라서 대부분의 애플리케이션에서 결과 아키텍처는 여러 개의 모듈을 사용하며, 각 모듈은 밀접하게 관련된 기능 집합을 캡슐화합니다.

`모듈()` 데코레이터는 모듈을 설명하는 속성을 가진 단일 객체를 받습니다:

공급자 네스트 인젝터에 의해 인스턴스화되고 적어도 이 모듈 전체에서 공유될 수 있는 공급자입니다.

컨트롤러 - 인스턴스화해야 하는 이 모듈에 정의된 컨트롤러 집합입니다.

수입 에 필요한 공급자를 내보내는 가져온 모듈의 목록입니다.

모듈

수출 이 모듈에서 제공되며 이 모듈을 가져오는 다른 모듈에서 사용할 수 있어야 하는 공급자의 하위 집합입니다. 공급자 자체 또는 토큰만 사용할 수 있습니다(값 `제공`).

모듈은 기본적으로 공급자를 캡슐화합니다. 즉, 현재 모듈에 직접 포함되지 않거나 가져온 모듈에서 내보낸 공급자를 삽입할 수 없습니다. 따라서 모듈에서 내보낸 공급자를 모듈의 공용 인터페이스 또는 API로 간주할 수 있습니다.

기능 모듈

CatsController와 `CatsService`는 동일한 애플리케이션 도메인에 속합니다. 서로 밀접하게 연관되어 있으므로 기능 모듈로 이동하는 것이 좋습니다. 기능 모듈은 특정 기능과 관련된 코드를 간단히 정리하여 코드를 체계적으로 유지하고 명확한 경계를 설정합니다. 이는 특히 애플리케이션 및/또는 팀의 규모가 커짐에 따라 복잡성을 관리하고

SOLID 원칙에 따라 개발하는 데 도움이 됩니다.

이를 시연하기 위해 `CatsModule`을 만들어 보겠습니다.

```
@@파일명(cats/cats.module)
```

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./cats.controller'에서 { CatsController }를 가져오고,  
'./cats.service'에서 { CatsService }를 가져옵니다;
```

```
모듈({
```

```
  컨트롤러: [CatsController],
```

```
    공급자: [CatsService],  
  })  
내보내기 클래스 CatsModule {}
```

정보 힌트 CLI를 사용하여 모듈을 만들려면 `$ nest g module cats` 명령을 실행하기만 하면 됩니다.

위에서 `cats.module.ts` 파일에 `CatsModule`을 정의하고 이 모듈과 관련된 모든 것을 `cats` 디렉토리로 옮겼습니다. 마지막으로 해야 할 일은 이 모듈을 루트 모듈(`app.module.ts` 파일에 정의된 `AppModule`)로 임포트하는 것입니다.

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./cats/cats.module'에서 { CatsModule }을 가져옵니다;
```

모듈({

```
  수입: [CatsModule],  
})
```

```
내보내기 클래스 AppModule {}
```

현재 디렉토리 구조는 다음과 같습니다:

```
src  
cats  
dto  
create-cat.dto.ts  
  
인터페이스  
cat.interface.ts  
cats.controller.ts  
cats.module.ts  
cats.service.ts  
app.module.ts  
main.ts
```

공유 모듈

Nest에서 모듈은 기본적으로 싱글톤이므로 여러 모듈 간에 모든 공급자의 동일한 인스턴스를 손쉽게 공유할 수 있습니다.



모든 모듈은 자동으로 공유 모듈이 됩니다. 일단 생성되면 모든 모듈에서 재사용할 수 있습니다. 다른 여러 모듈 간에 `CatsService` 인스턴스를 공유하고자 한다고 가정해 보겠습니다. 이를 위해서는 먼저 아래 그림과 같이 모듈의 `내보내기` 배열에 `CatsService` 프로바이더를 추가하여 내보내야 합니다:

`@@파일명(cats.module)`

'@nestjs/common'에서 `{ Module }`을 가져옵니다;

```
'./cats.controller'에서 { CatsController }를 가져오고,  
'./cats.service'에서 { CatsService }를 가져옵니다;  
  
모듈({  
    컨트롤러: [CatsController], 제공  
    자: [CatsService], 내보내기:  
    [CatsService]  
})  
내보내기 클래스 CatsModule {}
```

이제 `CatsModule`을 임포트하는 모든 모듈은 `CatsService`에 액세스할 수 있으며 이를 임포트하는 다른 모든 모듈과 동일한 인스턴스를 공유하게 됩니다.

모듈 다시 내보내기

위에서 보았듯이 모듈은 내부 공급자를 내보낼 수 있습니다. 또한 가져온 모듈을 다시 내보낼 수도 있습니다. 아래 예시에서는 `CommonModule`을 `CoreModule`에서 가져오고 내보내어 이 모듈을 가져오는 다른 모듈에서 사용할 수 있도록 합니다.

```
모듈({  
    수입: [CommonModule], 내보내기  
    : [CommonModule],  
})  
export 클래스 CoreModule {}
```

종속성 주입

모듈 클래스는 구성 목적으로 프로바이더를 삽입할 수도 있습니다(예: 구성 목적):

@@파일명 (cats.module)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./cats.controller'에서 { CatsController }를 가져오고,  
'./cats.service'에서 { CatsService }를 가져옵니다;
```

모듈({

```
컨트롤러: [CatsController], 제공자:
```

```
[CatsService],  
})
```

내보내기 클래스 CatsModule {

```
constructor(private catsService: CatsService) {}  
}
```

@@switch

```
'@nestjs/common'에서 { 모듈, 의존성 } 임포트;
```

```
'./cats.controller'에서 { CatsController } 임포트;
```

```
'./cats.service'에서 { CatsService } 임포트;
```

모듈({

```
컨트롤러: [CatsController],
```

```

    공급자: [CatsService],
  })
@Dependencies(CatsService) 내

보내기 클래스 CatsModule {
  constructor(catsService) {
    this.catsService = catsService;
  }
}

```

그러나 모듈 클래스 자체는 [순환 종속성](#)으로 인해 프로바이더로 주입할 수 없습니다. 전역 모듈

모든 곳에서 동일한 모듈 세트를 가져와야 한다면 지루할 수 있습니다. Nest와 달리 [Angular](#) 공급자는 글로벌 범위에 등록됩니다. 일단 정의되면 어디서나 사용할 수 있습니다. 그러나 Nest는 모듈 범위 내에서 프로바이더를 캡슐화합니다. 캡슐화 모듈을 먼저 가져오지 않으면 다른 곳에서 모듈의 프로바이더를 사용할 수 없습니다.

헬퍼, 데이터베이스 연결 등 모든 곳에서 즉시 사용할 수 있어야 하는 공급자 집합을 제공하려는 경우 [@Global\(\)](#) 데코레이터를 사용하여 모듈을 전역으로 만드세요.

```

'@nestjs/common'에서 { Module, Global }을 임포트하고,
'./cats.controller'에서 { CatsController}를 임포트하고,
'./cats.service'에서 { CatsService}를 임포트합니다;

@Global()
@Module({
  컨트롤러: [CatsController], 제공
  자: [CatsService], 내보내기:
  [CatsService],
})
내보내기 클래스 CatsModule {}

```

[Global\(\)](#) 데코레이터는 모듈을 전역 범위로 만듭니다. 전역 모듈은 일반적으로 루트 모듈이나 코어 모듈에 의해 한 번만 등록되어야 합니다. 위의 예시에서 [CatsService](#) 공급자는 유비쿼터스이며, 이 서비스를 삽입하려는 모듈은 import 배열에서 [CatsModule](#)을 임포트할 필요가 없습니다.

정보 힌트 모든 것을 전역으로 만드는 것은 좋은 디자인 결정이 아닙니다. 글로벌 모듈을 사용하면 필요 한 상용구의 양을 줄일 수 있습니다. 일반적으로 가져오기 배열은 소비자가 모듈의 API를 사용할 수 있도록 하는 데 선호되는 방법입니다.

동적 모듈

Nest 모듈 시스템에는 동적 모듈이라는 강력한 기능이 포함되어 있습니다. 이 기능을 사용하면 공급자를 동적으로 등록하고 구성할 수 있는 사용자 지정 가능한 모듈을 쉽게 만들 수 있습니다. [여기서는](#) 동적 모듈에 대해 광범위하게 다룹니다. 이 장에서는 모듈에 대한 소개를 완료하기 위해 간략한 개요를 제공하겠습니다.

다음은 데이터베이스 모듈에 대한 동적 모듈 정의의 예입니다:

```
@@파일명()  
'@nestjs/common'에서 { Module, DynamicModule }을 가져옵니다;  
'./database.providers'에서 { createDatabaseProviders }를 가져오고,  
'./connection.provider'에서 { Connection }을 가져옵니다;  
  
모듈({  
    공급자: [연결],  
})  
데이터베이스 모듈 클래스 내보내기 {  
    정적 forRoot(entities = [], options?): DynamicModule {  
        const providers = createDatabaseProviders(options, entities);  
        return {  
            모듈: DatabaseModule, 공  
                급자: 공급자, 내보내기: 공급  
                    자,  
                };  
        }  
    }  
    @@switch  
    '@nestjs/common'에서 { Module }을 가져옵니다;  
    './database.providers'에서 { createDatabaseProviders }를 가져오고,  
    './connection.provider'에서 { Connection }을 가져옵니다;  
  
모듈({  
    공급자: [연결],  
})  
데이터베이스 모듈 클래스 내보내기 {  
    static forRoot(entities = [], options) {  
        const providers = createDatabaseProviders(options, entities);  
        return {  
            모듈: DatabaseModule, 공  
                급자: 공급자, 내보내기: 공급  
                    자,  
                };  
        }  
    }  
}
```

정보 힌트 `forRoot()` 메서드는 동적 모듈을 동기식 또는 비동기식(즉, [프로미스를](#) 통해)으로 반환할 수 있습니다.

이 모듈은 기본적으로 [연결](#) 공급자를 정의하지만(`@Module()` 데코레이터 메타데이터에 있음), 추가로 `forRoot()` 메서드에 전달된 [엔티티](#) 및 [옵션](#) 객체에 따라 리포지토리와 같은 공급자 컬렉션을 노출합니다. 동적 모듈이 반환하는 프로퍼티는 `@Module()` 데코레이터에 정의된 기본 모듈 메타데이터를 재정의하지 않고 확장한다는 점에 유의하세요. 이것이 정적으로 선언된 [연결](#) 공급자와 동적으로 생성된 리포지토리 공급자가 모듈에서 내보내는 방식입니다.

전역 범위에서 동적 모듈을 등록하려면 [전역](#) 속성을 `true`로 설정합니다.

```
{  
  global: true,  
  모듈: DatabaseModule, 공급자:  
    공급자, 내보내기: 공급자,  
}
```

경고 경고 위에서 언급했듯이 모든 것을 글로벌하게 만드는 것은 좋은 디자인 결정이 아닙니다.

데이터베이스 모듈은 다음과 같은 방법으로 가져오고 구성할 수 있습니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./database/database.module'에서 { DatabaseModule }을 가져오고,  
'./users/entities/user.entity'에서 { User }를 가져옵니다;  
  
모듈({  
  임포트합니다: [DatabaseModule.forRoot([User])],  
})  
내보내기 클래스 AppModule {}
```

동적 모듈을 차례로 다시 내보내려면 내보내기 배열에서 `forRoot()` 메서드 호출을 생략할 수 있습니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./database/database.module'에서 { DatabaseModule }을 가져오고,  
'./users/entities/user.entity'에서 { User }를 가져옵니다;  
  
모듈({  
  임포트합니다:  
    [DatabaseModule.forRoot([User])], 내보내기:  
    [DatabaseModule],  
})  
내보내기 클래스 AppModule {}
```

동적 모듈 챕터에서 이 주제를 자세히 다루며 [작업 예제가 포함되어 있습니다](#).

정보 힌트 다음을 사용하여 고도로 사용자 정의 가능한 동적 모듈을 구축하는 방법을 알아보세요.

[구성 가능한 모듈 빌더를 소개합니다](#).

미들웨어

미들웨어는 라우트 핸들러보다 먼저 호출되는 함수입니다. 미들웨어 함수는 [요청 및 응답](#) 객체와 애플리케이션의 요청-응답 주기에서 [다음\(\)](#) 미들웨어 함수에 액세스할 수 있습니다. 다음 미들웨어 함수는 일반적으로 [next](#)라는 변수로 표시됩니다.



네스트 미들웨어는 기본적으로 [익스프레스](#) 미들웨어와 동일합니다. 공식 익스프레스 문서의 다음 설명은 미들웨어의 기능에 대해 설명합니다:

미들웨어 함수는 다음 작업을 수행할 수 있습니다:

- 코드를 실행합니다.
- 요청 및 응답 객체를 변경합니다. 요청-응답 주기를 종료합니다.
- 합니다.

스택의 다음 미들웨어 함수를 호출합니다.

현재 미들웨어 함수가 요청-응답 사이클을 종료하지 않으면 다음 미들웨어 함수로 제어권을 넘기기 위해 [next\(\)](#)를 호출해야 합니다. 그렇지 않으면 요청이 중단된 상태로 유지됩니다.

사용자 정의 Nest 미들웨어는 함수 또는 [@Injectable\(\)](#) 데코레이터가 있는 클래스에서 구현합니다. 함수는 특별한 요구 사항이 없는 반면, 클래스는 [NestMiddleware](#) 인터페이스를 구현해야 합니다. 클래스 메서드를 사용하여 간단한 미들웨어 기능을 구현하는 것으로 시작하겠습니다.

경고 Express와 Fastify는 미들웨어를 다르게 처리하고 다른 메소드 서명을 제공하므로 [여기에서](#) 자세히 알아보세요.

@@파일명(logger.middleware)

'@nestjs/common'에서 { Injectable, NestMiddleware }를 임포트하고,
'express'에서 { Request, Response, NextFunction }을 임포트합니다;

@Injectable()

```
export class LoggerMiddleware implements NestMiddleware {
  use(req: Request, res: Response, next: NextFunction) {
    console.log('요청...'); next();
  }
}
```

@@switch

'@nestjs/common'에서 { Injectable }을 가져옵니다;

@Injectable()

```
export class LoggerMiddleware {
  use(req, res, next) {
    console.log('요청...'); next();
  }
}
```

종속성 주입

Nest 미들웨어는 의존성 주입을 완벽하게 지원합니다. 프로바이더 및 컨트롤러와 마찬가지로 동일한 모듈 내에서 사용할 수 있는 종속성을 주입할 수 있습니다. 이 작업은 평소와 같이 **생성자**를 통해 수행됩니다.

미들웨어 적용

모듈() 데코레이터에는 미들웨어가 들어갈 자리가 없습니다. 대신 모듈 클래스의 **configure()** 메서드를 사용하여 설정합니다. 미들웨어를 포함하는 모듈은 **NestModule** 인터페이스를 구현해야 합니다.

AppModule 수준에서 **LoggerMiddleware**를 설정해 보겠습니다.

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module, NestModule, MiddlewareConsumer }를 임포트하고,  
'./common/middleware/logger.middleware'에서 { LoggerMiddleware }를 임포트하고,  
'./cats/cats.module'에서 { CatsModule }을 임포트합니다;
```

모듈({

```
    수입: [CatsModule],  
})  
export class AppModule implements NestModule {  
    configure(consumer: MiddlewareConsumer) {  
        소비자  
            .apply(LoggerMiddleware)  
            .forRoutes('cats');  
    }  
}
```

@@switch

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./common/middleware/logger.middleware'에서 { LoggerMiddleware }를 임포트하고,  
'./cats/cats.module'에서 { CatsModule }을 임포트합니다;
```

모듈({

```
    수입: [CatsModule],  
})  
export class AppModule {  
    configure(consumer) {  
        소비자  
            .apply(LoggerMiddleware)  
            .forRoutes('cats');  
    }  
}
```

위의 예제에서는 이전에 `CatsController` 내부에 정의된 `/cats` 라우트 핸들러에 대한 `LoggerMiddleware`를 설정했습니다. 미들웨어를 구성할 때 경로 경로와 요청 메서드가 포함된 객체를 `forRoutes()` 메서드에 전달하여 미들웨어를 특정 요청 메서드로 더 제한할 수도 있습니다. 아래 예제에서는 원하는 요청 메서드 유형을 참조하기 위해 `RequestMethod` 열거형을 임포트한 것을 확인할 수 있습니다.

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module, NestModule, RequestMethod, MiddlewareConsumer }
를 임포트합니다;

'./common/middleware/logger.middleware'에서 { LoggerMiddleware }를 임포트하고,
'./cats/cats.module'에서 { CatsModule }을 임포트합니다;
```

모듈({

```
  수입: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    소비자
      .apply(LoggerMiddleware)
      .forRoutes({ 경로: 'cats', 메서드: RequestMethod.GET });
  }
}
@@switch
'@nestjs/common'에서 { Module, RequestMethod }를 가져옵니다;
'./common/middleware/logger.middleware'에서 { LoggerMiddleware }를 임포트하고,
'./cats/cats.module'에서 { CatsModule }을 임포트합니다;
```

모듈({

```
  수입: [CatsModule],
})
export class AppModule {
  configure(consumer) {
    소비자
      .apply(LoggerMiddleware)
      .forRoutes({ 경로: 'cats', 메서드: RequestMethod.GET });
  }
}
```

정보 힌트 `configure()` 메서드는 `async/await`를 사용하여 비동기화할 수 있습니다(예

구성() 메서드 본문 내에서 비동기 연산이 완료되기를 기다립니다.)

경고 **익스프레스** 어댑터를 사용할 때 NestJS 앱은 기본적으로 패키지 `body-parser`에서 `json` 및

`urlencoded`를 등록합니다. 즉, **미들웨어소비자**를 통해 해당 미들웨어를 사용자 정의하려면

`NestFactory.create()`로 애플리케이션을 생성할 때 `bodyParser` 플래그를 `false`로 설정하여

전역 미들웨어를 꺼야 합니다.

패턴 기반 경로도 지원됩니다. 예를 들어 별표는 와일드카드로 사용되며 어떤 문자 조합과도 일치합니다:

```
forRoutes({ 경로: 'ab*cd' , 메서드: RequestMethod.ALL });
```

'ab*cd' 경로 경로는 abcd, ab_cd, abecd 등과 일치합니다. 문자 ?, +, * 및 ()는 경로 경로에 사용할 수 있으며 정규식 대응 문자의 하위 집합입니다. 하이픈(-)과 점(.)은 문자열 기반 경로에서 문자 그대로 해석됩니다.

경고 경고 fastify 패키지는 더 이상 와일드카드 별표 *를 지원하지 않는 최신 버전의 path-to-regexp 패키지를 사용합니다. 대신 매개변수(예: (.*))를 사용해야 합니다, 스플랫*).

미들웨어 소비자

MiddlewareConsumer는 헬퍼 클래스입니다. 미들웨어를 관리하기 위한 몇 가지 내장 메서드를 제공합니다. 모든 메서드는 유창한 스타일로 간단히 연결할 수 있습니다. forRoutes() 메서드는 단일 문자열, 여러 문자열, RouteInfo 객체, 컨트롤러 클래스, 심지어 여러 컨트롤러 클래스를 받을 수 있습니다. 대부분의 경우 쉼표로 구분된 컨트롤러 목록을 전달할 것입니다. 아래는 단일 컨트롤러를 사용한 예제입니다:

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module, NestModule, MiddlewareConsumer }를 임포트하고,  
'./common/middleware/logger.middleware'에서 { LoggerMiddleware }를 임포트하고,  
'./cats/cats.module'에서 { CatsModule }을 임포트합니다;  
'./cats/cats.controller'에서 { CatsController }를 가져옵니다;
```

모듈({

```
    수입: [CatsModule],  
})  
export class AppModule implements NestModule {  
    configure(consumer: MiddlewareConsumer) {  
        소비자  
            .apply(LoggerMiddleware)  
            .forRoutes(CatsController);  
    }  
}
```

@@switch

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./common/middleware/logger.middleware'에서 { LoggerMiddleware }를 임포트하고,  
'./cats/cats.module'에서 { CatsModule }을 임포트합니다;  
'./cats/cats.controller'에서 { CatsController }를 가져옵니다;
```

모듈({

```
    수입: [CatsModule],  
})  
export class AppModule {  
    configure(consumer) {  
        소비자  
            .apply(LoggerMiddleware)  
            .forRoutes(CatsController);  
    }  
}
```

정보 힌트 `apply()` 메서드는 단일 미들웨어를 받거나 `여러 미들웨어를 지정하기 위해 여러 인수를 받을 수 있습니다.`

노선 제외

때때로 특정 경로를 미들웨어 적용에서 제외시키고 싶을 때가 있습니다. `exclude()` 메서드를 사용하면 특정 경로를 쉽게 제외할 수 있습니다. 이 메서드는 아래와 같이 제외할 경로를 식별하는 단일 문자열, 여러 문자열 또는 `RouteInfo` 객체를 받을 수 있습니다:

```
소비자
.apply(LoggerMiddleware)
.제외(
  { 경로: 'cats', method: RequestMethod.GET },
  { path: 'cats', method: RequestMethod.POST },
  'cats/(.*)',
)
.forRoutes(CatsController);
```

정보 힌트 `exclude()` 메서드는 `경로 정규식` 패키지를 사용하여 와일드카드 매개변수를 지원합니다.

위의 예제에서 `LoggerMiddleware`는 `CatsController` 내부에 정의된 모든 경로에 바인딩됩니다.

`제외()` 메서드에 전달된 세 개를 제외합니다. 함수형 미들웨어

우리가 사용한 `LoggerMiddleware` 클래스는 매우 간단합니다. 멤버도 없고, 추가 메서드도 없으며, 종속성도 없습니다. 클래스 대신 간단한 함수로 정의할 수 없는 이유는 무엇일까요? 사실 가능합니다. 이러한 유형의 미들웨어를 함수형 미들웨어라고 합니다. 로거 미들웨어를 클래스 기반에서 함수형 미들웨어로 변환하여 차이점을 설명해 보겠습니다:

@@파일명(logger.middleware)

'express'에서 { 요청, 응답, 다음 함수 }를 가져옵니다;

```
export 함수 logger(req: 요청, res: 응답, next: NextFunction) { console.log(`요청
...`);
다음();
};

@@switch
export 함수 logger(req, res, next) {
  console.log(`요청...`);
  다음();
};
```

그리고 **앱모듈** 내에서 사용하세요:

@@파일명(앱.모듈) 소비자

```
.apply(logger)
  .forRoutes(CatsController);
```

정보 힌트 미들웨어에 종속성이 필요하지 않은 경우 언제든지 더 간단한 기능의 미들웨어 대안을 사용하는 것을 고려하세요.

여러 미들웨어

위에서 언급했듯이 순차적으로 실행되는 여러 미들웨어를 바인딩하려면 `apply()` 메서드 안에 쉼표로 구분된 목록을 제공하면 됩니다:

```
consumer.apply(cors(), helmet(), logger).forRoutes(CatsController);
```

글로벌 미들웨어

등록된 모든 경로에 미들웨어를 한 번에 바인딩하려면 `INestApplication` 인스턴스에서 제공하는 `사용()` 메서드를 사용할 수 있습니다:

`@@파일명`(메인)

```
const app = await NestFactory.create(AppModule);
app.use(logger);
await app.listen(3000);
```

정보 힌트 글로벌 미들웨어에서 DI 컨테이너에 액세스할 수 없습니다. `app.use()`를 사용할 때 `함수형 미들웨어`를 대신 사용할 수 있습니다. 또는 클래스 미들웨어를 사용하고 `AppModule`(또는 다른 모듈) 내에서 `.forRoutes('*')`를 사용하여 사용할 수 있습니다.

예외 필터

Nest에는 애플리케이션 전체에서 처리되지 않은 모든 예외를 처리하는 예외 계층이 내장되어 있습니다. 예외가 애플리케이션 코드에서 처리되지 않으면 이 계층에서 예외를 포착하여 적절한 사용자 친화적인 응답을 자동으로 전송합니다.



기본적으로 이 작업은 내장된 전역 예외 필터에 의해 수행되며, 이 필터는 `HttpException` 유형(및 그 하위 클래스)의 예외를 처리합니다. 예외가 인식되지 않는 경우(`HttpException`도 아니고 `HttpException`을 상속하는 클래스도 아닌 경우) 기본 제공 예외 필터는 다음과 같은 기본 JSON 응답을 생성합니다:

```
{  
  "상태코드": 500,  
  "메시지": "내부 서버 오류"  
}
```

정보 힌트 전역 예외 필터는 부분적으로 `http-errors` 라이브러리를 지원합니다. 기본적으로 `statusCode` 및 `message` 속성을 포함하는 모든 예외가 올바르게 채워지고 응답으로 다시 전송됩니다(인식할 수 없는 예외의 경우 기본 `InternalServerErrorException` 대신).

표준 예외 던지기

Nest는 `@nestjs/common` 패키지에서 노출되는 기본 제공 `HttpException` 클래스를 제공합니다. 일반적인 HTTP REST/GraphQL API 기반 애플리케이션의 경우, 특정 오류 조건이 발생하면 표준 HTTP 응답 객체를 전송하는 것이 가장 좋습니다.

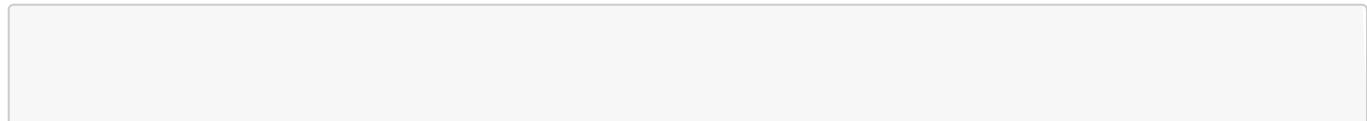
예를 들어, `CatsController`에는 `findAll()` 메서드(`GET` 라우트 핸들러)가 있습니다. 이를 시연하기 위해 다음과 같이 하드코딩하겠습니다:

```
@@파일명(cats.controller)  
@Get()  
async findAll() {  
  새로운 HttpException('금지됨', HttpStatus.FORBIDDEN)을 던집니다;  
}
```

정보 힌트 여기서는 `HttpStatus`을 사용했습니다. 이것은 헬퍼 열거 형의

`nestjs/common` 패키지.

클라이언트가 이 앤드포인트를 호출하면 다음과 같은 응답이 표시됩니다:



```
{  
  "statusCode": 403, "메시지":  
    "금지됨"  
}
```

`HttpException` 생성자는 응답을 결정하는 두 개의 필수 인수를 받습니다:

- `응답` 인수는 JSON 응답 본문을 정의합니다. 아래에 설명된 대로 `문자열` 또는 `객체`일 수 있습니다.
- `status` 인수는 `HTTP 상태 코드`를 정의합니다. 기본적으

로 JSON 응답 본문에는 두 가지 속성이 포함됩니다:

- `statusCode`: 기본값은 `status` 인수에 제공된 HTTP 상태 코드입니다.
- `메시지`: `상태에` 따른 HTTP 오류에 대한 간단한 설명입니다.

JSON 응답 본문의 메시지 부분만 재정의하려면 `응답` 인수에 문자열을 입력합니다. 전체 JSON 응답 본문을 재정의하려면 `응답` 인수에 객체를 전달합니다. Nest는 객체를 직렬화하여 JSON 응답 본문으로 반환합니다.

두 번째 생성자 인자인 `status`는 유효한 HTTP 상태 코드여야 합니다. 가장 좋은 방법은 [@nestjs/common](#)에서 가져온 `HttpStatus` 열거형을 사용하는 것입니다.

오류 `원인`을 제공하는 데 사용할 수 있는 세 번째 생성자 인수(선택 사항)인 옵션이 있습니다. 이 `원인` 객체는 응답 객체로 직렬화되지는 않지만 로깅 목적으로 유용할 수 있으며, `HttpException`을 발생시킨 내부 오류에 대한 중요한 정보를 제공합니다.

다음은 전체 응답 본문을 재정의하고 오류 원인을 제공하는 예제입니다:

```
@@파일명(cats.controller)
@Get()
async findAll() {
    try {
        await this.service.findAll()
    } catch (error) {
        throw new HttpException({
            status: HttpStatus.FORBIDDEN,
            오류: '이것은 사용자 지정 메시지입니다',
        }, HttpStatus.FORBIDDEN, { 원
            인: 오류
        });
    }
}
```

위의 내용을 사용하면 다음과 같이 응답이 표시됩니다:

```
{
    "상태": 403,
```

```

    "오류" : "이것은 사용자 지정 메시지입니다"
}

```

사용자 지정 예외

대부분의 경우 사용자 정의 예외를 작성할 필요가 없으며, 다음 섹션에 설명된 대로 기본 제공 Nest HTTP 예외를 사용할 수 있습니다. 사용자 정의 예외를 작성해야 하는 경우, 사용자 정의 예외가 기본 `HttpException` 클래스에서 상속되는 자체 예외 계층을 만드는 것이 좋습니다. 이 접근 방식을 사용하면 Nest가 예외를 인식하고 오류 응답을 자동으로 처리합니다. 이러한 사용자 정의 예외를 구현해 보겠습니다:

`@@파일명`(`forbidden`.예외)

```

export class ForbiddenException extends HttpException {
  constructor() {
    super('금지됨', HttpStatus.FORBIDDEN);
  }
}

```

`ForbiddenException`은 기본 `HttpException`을 확장하기 때문에 기본 제공 예외 처리기와 원활하게 작동하므로 `findAll()` 메서드 내에서 사용할 수 있습니다.

`@@파일명`(`cats.controller`)

```

@Get()
async findAll() {
  새로운 ForbiddenException()을 던집니다;
}

```

기본 제공 HTTP 예외

Nest는 기본 `HttpException`에서 상속되는 표준 예외 집합을 제공합니다. 이러한 예외는 `@nestjs/common` 패키지에서 노출되며, 가장 일반적인 HTTP 예외 중 다수를 나타냅니다:

- ◆ `BadRequestException`
- ◆ `UnauthorizedException`
- ◆ `NotFoundException`
- ◆ `ForbiddenException`
- ◆ `NotAcceptableException`
- ◆ `RequestTimeoutException`
- ◆ `ConflictException`
- ◆ `GoneException`
- ◆ `HttpVersionNotSupportedException`
- ◆
- ◆
- ◆

PayloadTooLargeException
UnsupportedMediaTypeException
UnprocessableEntityException
InternalServerErrorException

- `NotImplementedException`
- `ImATeapotException`
- `MethodNotAllowedException`
- `BadGatewayException`
- `ServiceUnavailableException`
- `GatewayTimeoutException`
- `PreconditionFailedException`

모든 기본 제공 예외는 옵션을 사용하여 오류 원인과 오류 설명을 모두 제공할 수도 있습니다.

매개변수입니다:

```
throw new BadRequestException('뭔가 나쁜 일이 발생했습니다', { 원인: new Error(), 설명: '일부 오류 설명' })
```

위의 내용을 사용하면 다음과 같이 응답이 표시됩니다:

```
{  
  "메시지": "뭔가 나쁜 일이 발생했습니다", "오류":  
  "일부 오류 설명", "statusCode": 400,  
}
```

예외 필터

기본(기본 제공) 예외 필터가 많은 경우를 자동으로 처리할 수 있지만 예외 계층을 완전히 제어하고 싶을 수도 있습니다. 예를 들어 로깅을 추가하거나 일부 동적 요인에 따라 다른 JSON 스키마를 사용하고 싶을 수 있습니다. 예외 필터는 바로 이러한 목적을 위해 설계되었습니다. 예외 필터를 사용하면 정확한 제어 흐름과 클라이언트에 다시 전송되는 응답의 내용을 제어할 수 있습니다.

`HttpException` 클래스의 인스턴스인 예외를 포착하고 이에 대한 사용자 정의 응답 로직을 구현하는 예외 필터를 만들어 보겠습니다. 이를 위해서는 기본 플랫폼의 요청 및 응답 객체에 액세스해야 합니다. 요청 객체에 액세스하여 원본 URL을 가져와 로깅 정보에 포함할 수 있습니다. `Response` 객체를 사용하여 `response.json()` 메서드를 사용하여 전송된 응답을 직접 제어할 것입니다.

@@파일명(http-exception.filter)

'@nestjs/common'에서 { ExceptionFilter, Catch, ArgumentsHost, HttpException }을 가져옵니다;

'express'에서 { 요청, 응답 }을 가져옵니다;

@Catch(HttpException)

```
export class HttpExceptionFilter implements ExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
```

```
const status = 예외.getStatus();

응답

    .status(상태)
    .json({
        statusCode: 상태,
        timestamp: new Date().toISOString(), 경
        로: request.url,
    });
}

}

@@switch
'@nestjs/common'에서 { Catch, HttpException }을 가져옵니다;
```

```
@Catch(HttpException)
export class HttpExceptionFilter {
    catch(exception, host) {
        const ctx = host.switchToHttp();
        const response = ctx.getResponse();
        const request = ctx.getRequest();
        const status = exception.getStatus();
```

응답

```
.status(상태)
.json({
    statusCode: 상태,
    timestamp: new Date().toISOString(), 경
    로: request.url,
});
}
```

정보 힌트 모든 예외 필터는 일반 `ExceptionFilter<T>` 인터페이스를 구현해야 합니다. 이를 위해서는 `catch(예외: T, 호스트: ArgumentsHost)` 메서드에 지정된 서명을 제공해야 합니다. `T`는 예외의 유

형을 나타냅니다. `@nestjs/platform-fastify`를 사용하는 경우 `응답.send()`을 사용할 수 있습니다.

대신 `응답.json()`을 사용하세요. `fastify`에서 올바른 유형을 가져오는 것을 잊지 마세요.

`캐치(HttpException)` 데코레이터는 필요한 메타데이터를 예외 필터에 바인딩하여 이 특정 필터가 `HttpException` 유형의 예외만 찾고 있음을 Nest에 알려줍니다. `캐치()` 데코레이터는 단일 매개변수 또는 쉼표로 구분된 목록을 받을 수 있습니다. 이를 통해 한 번에 여러 유형의 예외에 대한 필터를 설정할 수 있습니다.

인수 호스트

`catch()` 메서드의 매개변수를 살펴봅시다. 예외 매개변수는 현재 처리 중인 예외 객체입니다. `host` 매개변수는 `ArgumentsHost` 객체입니다. `ArgumentsHost`는 실행 컨텍스트 챕터*에서 자세히 살펴볼 강력한 유ти리티 객체입니다. 이 코드 샘플에서는 이 객체를 사용하여 원래 요청 처리기(예외가 발생한 컨트롤러에서)로 전달되는 요청 및 응답 객체에 대한 참조를 얻습니다. 이 코드 샘플에서는 일부

헬퍼 메서드를 사용하여 원하는 요청 및 응답 객체를 가져올 수 있습니다. 자세히 알아보기

[ArgumentsHost를 입력합니다.](#)

*이 수준의 추상화가 필요한 이유는 ArgumentsHost가 모든 컨텍스트(예: 지금 작업 중인 HTTP 서버 컨텍스트뿐만 아니라 마이크로서비스와 웹소켓도 포함)에서 작동하기 때문입니다. 실행 컨텍스트 챕터에서는 ArgumentsHost와 그 도우미 함수를 이용해 모든 실행 컨텍스트에 적합한 기본 인자에 접근하는 방법을 살펴볼 것입니다. 이를 통해 모든 컨텍스트에서 작동하는 일반 예외 필터를 작성할 수 있습니다.

바인딩 필터

새로운 HttpExceptionFilter를 CatsController의 create() 메서드에 연결해 보겠습니다.

```
@@파일명(cats.controller)
@Post()
사용 필터(새로운 HttpExceptionFilter())
async create(@Body() createCatDto: CreateCatDto) {
  throw new ForbiddenException();
}

@@스위치

@포스트()

사용필터(새로운 HttpExceptionFilter())
@Bind(Body())
async create(createCatDto) {
  throw new ForbiddenException();
}
```

정보 힌트 @UseFilters() 데코레이터는 @nestjs/common 패키지에서 가져온 것입니다.

여기서는 @UseFilters() 데코레이터를 사용했습니다. 캐치() 데코레이터와 마찬가지로 단일 필터 인스턴스 또는 쉼표로 구분된 필터 인스턴스 목록을 받을 수 있습니다. 여기서는 HttpExceptionFilter 인스턴스를 생성했습니다. 또는 인스턴스 대신 클래스를 전달하여 인스턴스화에 대한 책임을 프레임워크에 맡기고 의존성 주입을 활성화할 수 있습니다.

```
@@파일명(cats.controller) @Post()
@UseFilters(HttpExceptionFilter)
async create(@Body() createCatDto: CreateCatDto) {
    throw new ForbiddenException();
}

@@스위치

@포스트()

사용필터(HttpExceptionFilter)
@Bind(Body())
async create(createCatDto) {
    throw new ForbiddenException();
}
```

정보 힌트 가능하면 인스턴스 대신 클래스를 사용하여 필터를 적용하는 것을 선호합니다. Nest는 전체 모듈

에서 동일한 클래스의 인스턴스를 쉽게 재사용할 수 있으므로 메모리 사용량을 줄일 수 있습니다.

위의 예제에서 `HttpExceptionFilter`는 단일 `create()` 라우트 핸들러에만 적용되어 메소드 범위가 지정되어 있습니다. 예외 필터는 컨트롤러/리졸버/게이트웨이의 메서드 범위, 컨트롤러 범위 또는 전역 범위 등 다양한 수준에서 범위를 지정할 수 있습니다.

예를 들어 컨트롤러 범위로 필터를 설정하려면 다음과 같이 하면 됩니다:

```
@@filename(cats.controller)
@UseFilters(new HttpExceptionFilter())
export class CatsController {}
```

이 구성은 내부에 정의된 모든 라우트 핸들러에 대해 `HttpExceptionFilter`를 설정합니다.

`CatsController`.

전역 범위 필터를 만들려면 다음을 수행합니다:

```
@@파일명(메인)

비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new HttpExceptionFilter());
  await app.listen(3000);
}

부트스트랩();
```

경고 경고 `useGlobalFilters()` 메서드는 게이트웨이 또는 하이브리드 애플리케이션에 대한 필터를 설정하지 않습니다.

전역 범위 필터는 모든 컨트롤러와 모든 라우트 핸들러에 대해 전체 애플리케이션에서 사용됩니다. 종속성 주입과 관련하여 모듈 외부에서 등록한 전역 필터(위 예제에서와 같이 `useGlobalFilters()`를 사용)는 모듈의 컨텍스트 외부에서 수행되므로 종속성을 주입할 수 없습니다. 이 문제를 해결하기 위해 다음 구성을 사용하여 모든 모듈에서 직접 전역 범위 필터를 등록할 수 있습니다:

@@파일명(앱.모듈)

'@nestjs/common'에서 { Module }을 가져오고,
'@nestjs/core'에서 { APP_FILTER }를 가져옵니다;

모듈({ providers:

```
[  
  {  
    제공해야 합니다: APP_FILTER,  
    useClass: HttpExceptionFilter,  
  },  
],  
})
```

내보내기 클래스 AppModule {}

정보 힌트 이 접근 방식을 사용하여 필터에 대한 종속성 주입을 수행할 때는 이 구조가 사용되는 모듈에 관계없이 필터가 실제로는 전역이라는 점에 유의하세요. 이 작업을 어디에서 수행해야 할까요? 필터가 정의된 모듈(위 예제에서는 `HttpExceptionFilter`)을 선택하면 됩니다. 또한 사용 클래스만이 사용자 정의 공급자 등록을 처리하는 유일한 방법은 아닙니다. [여기에서](#) 자세히 알아보세요.

이 기술을 사용하여 필요한 만큼 필터를 추가할 수 있으며, 각 필터를 공급자 배열에 추가하기만 하면 됩니다.

모든 것을 포착

처리되지 않은 모든 예외를 잡으려면 (예외 유형에 관계없이) `@Catch()` 데코레이터의 매개변수 목록이 비어있는 경우(예: `@Catch()`).

아래 예시에서는 [HTTP 어댑터](#)를 사용하여 응답을 전달하고 플랫폼별 객체([요청](#) 및 [응답](#))를 직접 사용하지 않기 때문에 플랫폼에 구애받지 않는 코드가 있습니다:

```
import {
  ExceptionFilter,
  Catch,
  ArgumentsHost,
  HttpException,
  HttpStatus,
}를 '@nestjs/common'에서 가져옵니다;
'@nestjs/core'에서 { HttpAdapterHost }를 가져옵니다;

@Catch()
export class AllExceptionsFilter implements ExceptionFilter {
  constructor(private readonly httpAdapterHost: HttpAdapterHost) {}

  catch(예외: 알 수 없음, 호스트: ArgumentsHost): void {
    // 특정 상황에서는 `httpAdapter`를 사용하지 못할 수 있습니다.
    // 생성자 메서드가 있으므로 여기서 해결해야 합니다. const {
    httpAdapter } = this.httpAdapterHost;

    const ctx = host.switchToHttp();

    const httpStatus =
      예외 인스턴스 오브 HttpException
        예외.getStatus()
        : HttpStatus.INTERNAL_SERVER_ERROR;

    const responseBody = {
      statusCode: httpStatus,
      timestamp: 새로운 Date().toISOString(),
      경로: httpAdapter.getRequestUrl(ctx.getRequest()),
    };

    httpAdapter.reply(ctx.getResponse(), responseBody, httpStatus);
  }
}
```

경고 모든 것을 캐치하는 예외 필터와 특정 유형에 바인딩된 필터를 결합할 때는 특정 필터가 바인딩 된 유형을 올바르게 처리할 수 있도록 '무엇이든 캐치' 필터를 먼저 선언해야 합니다.

상속

일반적으로 애플리케이션 요구 사항을 충족하기 위해 완전히 사용자 정의된 예외 필터를 만듭니다. 그러나 기본 제공되는 기본 전역 예외 필터를 간단히 확장하고 특정 요인에 따라 동작을 재정의하려는 사용 사례가 있을 수 있습니다.

예외 처리를 기본 필터에 위임하려면 `BaseExceptionFilter`를 확장해야 합니다.

를 생성하고 상속된 `catch()` 메서드를 호출합니다.

`@@파일명`(모든 예외 .필터)

```
'@nestjs/common'에서 { Catch, ArgumentsHost }를 임포트하고
, '@nestjs/core'에서 { BaseExceptionFilter }를 임포트합니다;
```

`@Catch()`

```
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(예외, 호스트);
  }
}
```

`@@switch`

```
'@nestjs/common'에서 { Catch }를 가져옵니다;
 '@nestjs/core'에서 { BaseExceptionFilter }를 가져옵니다;
```

`@Catch()`

```
export class AllExceptionsFilter extends BaseExceptionFilter {
  catch(exception, host) {
    super.catch(예외, 호스트);
  }
}
```

경고 메서드 범위 필터와 컨트롤러 범위 필터를 확장하는 메서드 범위 필터는 `새로` 만들기로 인스턴스화 해서는 안 됩니다. 대신 프레임워크가 자동으로 인스턴스화하도록 하세요.

위의 구현은 접근 방식을 보여주는 셀일 뿐입니다. 확장 예외 필터의 구현에는 맞춤형 비즈니스 로직(예: 다양한 조건 처리)이 포함될 수 있습니다.

전역 필터는 기본 필터를 확장할 수 있습니다. 이 작업은 두 가지 방법 중 하나로 수행할 수 있습니다.

첫 번째 방법은 사용자 정의 전역 필터를 인스턴스화할 때 `HttpAdapter` 참조를 삽입하는 것입니다:

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule);
```

```
const { httpAdapter } = app.get(HttpAdapterHost);
app.useGlobalFilters(new AllExceptionsFilter(httpAdapter));

await app.listen(3000);
}

부트스트랩();
```

두 번째 방법은 여기에 표시된 것처럼 APP_FILTER 토큰을 사용하는 것입니다.

파이프

파이프는 `@Injectable()` 데코레이터로 주석이 달린 클래스로, 파이프 트랜스폼을 구현합니다.

인터페이스.



파이프에는 두 가지 일반적인 사용 사례가 있습니다:

- 변환: 입력 데이터를 원하는 형식으로 변환(예: 문자열에서 정수로)
- 유효성 검사: 입력 데이터를 평가하여 유효하면 변경하지 않고 통과시키고, 그렇지 않으면 예외를 던집니다.

두 경우 모두 파이프는 [컨트롤러 라우트 핸들러에서](#) 처리 중인 `인수`를 대상으로 작동합니다. Nest는 메서드가 호출되기 직전에 파이프를 삽입하고, 파이프는 메서드에 전달되는 인수를 받아 이를 대상으로 작업합니다. 이때 모든 변환 또는 유효성 검사 작업이 수행되고, 그 후에 라우트 핸들러가 (잠재적으로) 변환된 인수를 사용하여 호출됩니다.

Nest에는 바로 사용할 수 있는 다양한 내장 파이프가 제공됩니다. 사용자 정의 파이프를 직접 구축할 수도 있습니다. 이 장에서는 기본 제공 파이프를 소개하고 이를 라우트 핸들러에 바인딩하는 방법을 보여드리겠습니다. 그런 다음 몇 가지 사용자 지정 파이프를 살펴보고 처음부터 파이프를 구축하는 방법을 보여드리겠습니다.

정보 힌트 파이프는 예외 영역 내에서 실행됩니다. 즉, 파이프가 예외를 던지면 예외 레이어(전역 예외 필터 및 현재 컨텍스트에 적용되는 모든 [예외 필터](#))에서 처리됩니다. 위의 내용을 고려할 때, 파이프에서 예외가 발생하면 컨트롤러 메서드가 이후에 실행되지 않는다는 것을 분명히 알 수 있습니다. 이는 시스템 경계에서 외부 소스에서 애플리케이션으로 들어오는 데이터의 유효성을 검사하는 모범 사례 기법을 제공합니다.

내장 파이프

Nest에는 9개의 파이프가 기본으로 제공됩니다:

- `ValidationPipe`
- `ParseIntPipe`
- `ParseFloatPipe`
- `ParseBoolPipe`

`ParseArrayPipe`•

`ParseUUIDPipe`•

`ParseEnumPipe`

- `DefaultValuePipe`

- `ParseFilePipe`

이 패키지는 `@nestjs/common` 패키지에서 내보내집니다.

`ParseIntPipe` 사용에 대해 간단히 살펴보겠습니다. 이것은 파이프가 메서드 핸들러 매개변수가 자바스크립트 정수로 변환되도록 하거나 변환에 실패할 경우 예외를 던지는 변환 사용 사례의 예시입니다. 이 장의 뒷부분에서는 `ParseIntPipe`에 대한 간단한 사용자 정의 구현을 보여드리겠습니다. 아래의 예제 기법은 다른 기본 제공 변환 파이프에도 적용됩니다.

(이 장에서는 `ParseBoolPipe`, `ParseFloatPipe`, `ParseEnumPipe`, `ParseArrayPipe` 및 `ParseUUIDPipe`를 `Parse*` 파이프라고 부릅니다).

바인딩 파이프

파이프를 사용하려면 파이프 클래스의 인스턴스를 적절한 컨텍스트에 바인딩해야 합니다. `ParseIntPipe` 예제에서는 파이프를 특정 라우트 핸들러 메서드와 연결하고 메서드가 호출되기 전에 파이프가 실행되도록 하려고 합니다. 이를 위해 메서드 매개변수 수준에서 파이프를 바인딩하는 다음 구문을 사용합니다:

```
@Get(':id')
async findOne(@Param('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}
```

이렇게 하면 다음 두 가지 조건 중 하나가 참인지 확인합니다. `findOne()` 메서드에서 받은 매개 변수가 숫자이거나 (`this.catsService.findOne()` 호출에서 예상한 대로) 라우트 핸들러가 호출되기 전에 예외가 발생했습니다.

예를 들어 경로가 다음과 같이 호출된다고 가정합니다:

```
GET localhost:3000/abc
```

Nest는 이와 같은 예외를 발생시킵니다:

```
{
  "상태코드": 400,
  "메시지": "유효성 검사에 실패했습니다(숫자 문자열이 예상됨)", "오류": "잘
  못된 요청"
}
```

예외가 발생하면 `findOne()` 메서드의 본문이 실행되지 않습니다.

위의 예제에서는 인스턴스가 아닌 클래스(`ParseIntPipe`)를 전달하여 인스턴스화에 대한 책임을 프레임워크에 맡기고 의존성 주입을 활성화합니다. 파이프 및 가드와 마찬가지로, 대신 제자리 인스턴스를 전달할 수 있습니다. 제자리 인스턴스를 전달하는 것은 옵션을 전달하여 내장된 파이프의 동작을 사용자 정의하려는 경우에 유용합니다:

```
@Get(':id')
async findOne(
  @Param('id', new ParseIntPipe({ errorHttpStatusCode:
HttpStatus.NOT_ACCEPTABLE })))
  ID: 숫자,
) {
```

```

    this.catsService.findOne(id)을 반환합니다;
}

```

다른 변환 파이프(모든 Parse* 파이프)를 바인딩하는 것도 비슷하게 작동합니다. 이러한 파이프는 모두 경로 매개 변수, 쿼리 문자열 매개변수 및 요청 본문 값의 유효성을 검사하는 컨텍스트에서 작동합니다.

예를 들어 쿼리 문자열 매개변수를 사용합니다:

```

@Get()
async findOne(@Query('id', ParseIntPipe) id: number) {
  return this.catsService.findOne(id);
}

```

다음은 문자열 매개변수를 구문 분석하고 해당 매개변수가 UUID인지 확인하는 ParseUUIDPipe의 예제입니다.

```

@@파일명() @Get(':uuid')
async findOne(@Param('uuid', new ParseUUIDPipe()) uuid: string) {
  return this.catsService.findOne(uuid)을 반환합니다;
}
@@switch
@Get(':uuid')
@Bind(Param('uuid', new ParseUUIDPipe()))
async findOne(uuid) {
  이캣서비스.findOne(uuid)을 반환합니다;
}

```

정보 힌트 ParseUUIDPipe()를 사용할 때 버전 3, 4 또는 5의 UUID를 구문 분석하는 경우 특정 버전의 UUID만 필요한 경우 파이프 옵션에서 버전을 전달할 수 있습니다.

위에서 다양한 Parse* 내장 파이프 제품군을 바인딩하는 예제를 살펴봤습니다. 유효성 검사 파이프를 바인딩하는 것은 조금 다르므로 다음 섹션에서 설명하겠습니다.

정보 힌트 또한 유효성 검사 파이프에 대한 광범위한 예제는 유효성 검사 [기술](#)을 참조하세요.

맞춤형 파이프

앞서 언급했듯이 자신만의 사용자 정의 파이프를 구축할 수 있습니다. Nest는 강력한 기본 제공 ParseIntPipe 및 ValidationPipe를 제공하지만, 사용자 정의 파이프가 어떻게 구성되는지 알아보기 위해 각각의 간단한 사

용자 정의 버전을 처음부터 빌드해 보겠습니다.

간단한 `ValidationPipe`부터 시작하겠습니다. 처음에는 단순히 입력값을 받고 즉시 동일한 값을 반환하도록 하여 동일성 함수처럼 동작하도록 하겠습니다.

`@@파일명`(유효성 검사.파이프)

에서 { PipeTransform, 인젝터블, 인자 메타데이터 }를 임포트합니다.

```
'@nestjs/common';

@Injectable()
export class ValidationPipe 구현 PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    반환 값입니다;
  }
}
@@switch
'@nestjs/common'에서 { Injectable }을 가져옵니다;

@Injectable()
export class ValidationPipe {
  transform(value, metadata) {
    반환 값입니다;
  }
}
```

정보 힌트 `PipeTransform<T, R>`은 모든 파이프에서 구현해야 하는 일반 인터페이스입니다. 일반 인터페이스는 `T`를 사용하여 입력 `값의` 유형을 나타내고 `R`을 사용하여 `transform()` 메서드의 반환 유형을 나타냅니다.

모든 파이프는 `파이프 트랜스폼` 인터페이스 컨트랙트를 이행하기 위해 `transform()` 메서드를 구현해야 합니다. 이 메서드에는 두 개의 매개변수가 있습니다:

- `값`
- `메타데이터`

`value` 매개변수는 현재 처리된 메서드 인자(경로 처리 메서드에서 수신하기 전)이며, `메타데이터`는 현재 처리된 메서드 인자의 메타데이터입니다. 메타데이터 객체에는 이러한 속성이 있습니다:

```
내보내기 인터페이스 ArgumentMetadata {
  type: 'body' | 'query' | 'param' | 'custom';
  metatype? 유형<알 수 없음>;
  데이터?: 문자열;
}
```

이러한 속성은 현재 처리된 인수를 설명합니다.

유형
메타타입

데이터

인수가 본문 `@Body()`, 쿼리 `@Query()`, 매개변수 `@Param()` 또는 사용자 정의 매개변수인지 여부를 나타냅니다(자세한 내용은 [여기를 참조하세요](#)).

인수의 메타타입(예: `문자열`)을 제공합니다. 참고: 라우트 핸들러 메서드 서명에서 타입 선언을 생략하거나 바닐라 자바스크립트를 사용하는 경우 이 값은 `정의되지 않`습니다.

데코레이터에 전달된 문자열(예: `@Body('문자열')`). 데코레이터 괄호를 비워두면 `정의되지 않`습니다.

경고 TypeScript 인터페이스는 트랜슬레이션 중에 사라집니다. 따라서 메서드 매개변수의 유형이 클래

스 대신 인터페이스로 선언된 경우 **메타타입** 값은 **Object**가 됩니다.

스키마 기반 유효성 검사

유효성 검사 파이프를 좀 더 유용하게 만들어 봅시다. 서비스 메서드를 실행하기 전에 포스트 본문 객체가 유효한지 확인해야 하는 **CatsController**의 **create()** 메서드를 자세히 살펴봅시다.

```
@@파일명() @Post()
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@스위치

@포스트()
async create(@Body() createCatDto) {
  this.catsService.create(createCatDto);
}
```

CreateCatDto 본문 매개변수를 집중적으로 살펴봅시다. 이 매개변수의 유형은 **CreateCatDto**입니다:

```
@@파일명(create-cat.dto) 내보내기

클래스 CreateCatDto {
  이름: 문자열; 나이
  : 숫자; 품종: 문
  자열;
}
```

create 메서드로 들어오는 모든 요청에 유효한 본문이 포함되어 있는지 확인하고자 합니다. 따라서

createCatDto 객체의 세 멤버의 유효성을 검사해야 합니다. 라우트 핸들러 메서드 내부에서 이 작업을 수행 할 수 있지만 단일 책임 규칙(SRP)을 위반하므로 이상적이지 않습니다.

또 다른 접근 방식은 유효성 검사기 클래스를 생성하고 그곳에 작업을 위임하는 것입니다. 이 방법은 각 메서드를 시작할 때마다 이 유효성 검사기를 호출하는 것을 기억해야 한다는 단점이 있습니다.

유효성 검사 미들웨어를 만드는 것은 어떨까요? 이 방법이 효과가 있을 수 있지만, 안타깝게도 전체 애플리케이션의 모든 컨텍스트에서 사용할 수 있는 일반 미들웨어를 만드는 것은 불가능합니다. 미들웨어는 호출될 때

들려와 그 매개변수 등 실행 컨텍스트를 인식하지 못하기 때문입니다.

물론 이것이 바로 파이프를 설계하는 사용 사례입니다. 이제 유효성 검사 파이프를 구체화해 보겠습니다.

개체 스키마 유효성 검사

깔끔하고 깔끔한 방식으로 객체 유효성 검사를 수행하는 데 사용할 수 있는 몇 가지 접근 방식이 있습니다. 일반적인 접근 방식 중 하나는 스키마 기반 유효성 검사를 사용하는 것입니다. 이 접근 방식을 사용해 보겠습니다.

Zod 라이브러리를 사용하면 읽기 쉬운 API를 사용하여 간단한 방식으로 스키마를 생성할 수 있습니다. Zod 기반 스키마를 사용하는 유효성 검사 파이프를 구축해 보겠습니다.

필요한 패키지를 설치하는 것으로 시작하세요:

```
$ npm install --save zod
```

아래 코드 샘플에서는 스키마를 생성자 인수로 받는 간단한 클래스를 생성합니다. 그런 다음 제공된 스키마에 대해 들어오는 인수의 유효성을 검사하는 `schema.parse()` 메서드를 적용합니다.

앞서 언급했듯이 유효성 검사 파이프는 변경되지 않은 값을 반환하거나 예외를 던집니다.

다음 섹션에서는 `@UsePipes()` 데코레이터를 사용하여 주어진 컨트롤러 메서드에 적절한 스키마를 제공하는 방법을 살펴보겠습니다. 이렇게 하면 우리가 의도한 대로 컨텍스트 간에 유효성 검사 파이프를 재사용할 수 있습니다.

@@파일명()

'@nestjs/common'에서 { PipeTransform, ArgumentMetadata, BadRequestException }을
임포트합니다;

'zod'에서 { ZodObject }를 가져옵니다;

```
export class ZodValidationPipe 구현 PipeTransform { constructor(private schema: ZodObject<any>) {}

transform(value: unknown, metadata: ArgumentMetadata) {
    try {
        this.schema.parse(value);
    } catch (error) {
        새로운 BadRequestException('유효성 검사 실패')을 던집니다;
    }
    반환 값입니다;
}
}

@@switch
'@nestjs/common'에서 { BadRequestException }을 가져오고,
'zod'에서 { ZodObject }를 가져옵니다;
```

```
export class ZodValidationPip {
constructor(private schema) {}

transform(value, 메타데이터) {
    try {
        this.schema.parse(value);
    } catch (error) {
        새로운 BadRequestException('유효성 검사 실패')을 던집니다;
    }
}
```

```
    반환 값입니다;  
}  
}
```

바인딩 유효성 검사 파이프

앞서 변환 파이프를 바인딩하는 방법(예: `ParseIntPipe` 및 나머지 `Parse*` 파이프)을 살펴봤습니다. 유효성 검사

파이프를 바인딩하는 방법도 매우 간단합니다.

이 경우 메서드 호출 수준에서 파이프를 바인딩하고 싶습니다. 현재 예제에서는 `ZodValidationPipe`를 사용하려면 다음을 수행해야 합니다:

- . `ZodValidationPipe`의 인스턴스를 생성합니다.
- . 파이프의 클래스 생성자에서 컨텍스트별 Zod 스키마를 전달합니다.
- . 파이프를 메서드 Zod 스키마 예

제에 바인딩합니다:

```
'zod'에서 { z }를 가져옵니다;
```

```
내보내기 const createCatSchema = z  
  .object({  
    이름: z.string(),  
    나이: z.number(), 품종:  
    z.array(),  
  })  
  .required();
```

```
내보내기 유형 CreateCatDto = z.infer<typeof createCatSchema>;
```

아래 그림과 같이 `@UsePipes()` 데코레이터를 사용하여 이를 수행합니다:

```
@@파일명(cats.controller)
@Post()
UsePipes(new ZodValidationPipe(createCatSchema))
async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
}

스위치 @포스트()

@바인드(본문())
UsePipes(new ZodValidationPipe(createCatSchema))
async create(createCatDto) {
    this.catsService.create(createCatDto);
}
```

정보 힌트 `@UsePipes()` 데코레이터는 `@nestjs/common` 패키지에서 가져옵니다.

경고 `zod` 라이브러리를 사용하려면 `엄격널체크` 구성을 사용하도록 설정해야 합니다.

`tsconfig.json` 파일을 만듭니다.

클래스 유효성 검사기

경고 이 섹션의 기술에는 TypeScript가 필요하며 앱이 바닐라 JavaScript를 사용하여 작성된 경우 사용할 수 없습니다.

유효성 검사 기법에 대한 다른 구현을 살펴보겠습니다.

Nest는 [클래스 유효성](#) 검사기 라이브러리와 잘 작동합니다. 이 강력한 라이브러리를 사용하면 데코레이터 기반 유효성 검사를 사용할 수 있습니다. 데코레이터 기반 유효성 검사는 특히 처리된 프로퍼티의 [메타타입](#)에 액세스할 수 있기 때문에 Nest의 파이프 기능과 결합할 때 매우 강력합니다. 시작하기 전에 필요한 패키지를 설치해야 합니다:

```
$ npm i --save class-validator class-transformer
```

설치가 완료되면 `CreateCatDto` 클래스에 몇 가지 데코레이터를 추가할 수 있습니다. 이 기법의 중요한 장점은 별도의 유효성 검사 클래스를 만들지 않고도 `CreateCatDto` 클래스가 Post 본문 객체에 대한 단일 소스로 유지된다는 점입니다.

```
@@파일명(create-cat.dto)
```

```
'class-validator'에서 { IsString, IsInt }를 가져옵니다;
```

```
export 클래스 CreateCatDto {
```

```
  @IsString()
```

```
  이름: 문자열;
```

```
  IsInt() 나이:
```

```
  숫자;
```

```
  IsString() 품종
```

정보 힌트 [문자열](#): 클래스 유효성 검사기 데코레이터에 대한 자세한 내용은 [여기를](#) 참조하세요.

이제 이러한 어노테이션을 사용하는 `ValidationPipe` 클래스를 만들 수 있습니다.

`@@파일명`(유효성 검사.파이프)

```
import { PipeTransform, Injector, Validator, BadRequestException
} from '@nestjs/common';
import { validate } from 'class-validator';
import { plainToInstance } from 'class-transformer';
```

```

@Injectable()
export class ValidationPipe 구현 PipeTransform<any> { async
  transform(value: any, { metatype }: ArgumentMetadata) {
    if (!메타타입 || !this.toValidate(메타타입)) { 반환값을
      반환합니다;
    }
    const object = plainToInstance(메타타입, 값); const
    errors = await validate(object);
    if (errors.length > 0) {
      새로운 BadRequestException('유효성 검사 실패')을 던집니다;
    }
    반환 값입니다;
  }

  private toValidate(메타타입: 함수): boolean {
    상수 타입: 함수[] = [문자열, 부울, 숫자, 배열, 객체]; 반환 !types.includes(메타
    타입);
  }
}

```

정보 힌트 다시 한 번 말씀드리자면, ValidationPipe는 Nest에서 기본으로 제공되므로 일반적인 유효성 검사 파이프를 직접 빌드할 필요가 없습니다. 기본 제공 ValidationPipe는 이 장에서 빌드한 샘플보다 더 많은 옵션을 제공하지만, 사용자 정의 구축 파이프의 메커니즘을 설명하기 위해 기본으로 유지했습니다. ~~자체 한 대용은 여기에서 많은 예제와 함께 확인할 수 있습니다.~~ 라이브러리와 동일한 작성자가 만든 라이브러리로, 결과적으로 두 라이브러리는 매우 잘 어울립니다.

이 코드를 살펴봅시다. 먼저 transform() 메서드가 비동기로 표시되어 있다는 점에 주목하세요. 이는 Nest가 동기 및 비동기 파이프를 모두 지원하기 때문에 가능합니다. 이 메서드를 비동기로 만든 이유는 클래스 유효성 검사기 유효성 검사 중 일부가 비동기일 수 있기 때문입니다(프로미스 활용).

다음으로 메타타입 필드를 메타타입 매개변수로 추출하기 위해 구조조정을 사용하고 있습니다 (ArgumentMetadata에서 이 멤버만 추출). 이것은 전체 ArgumentMetadata를 가져온 다음 메타타입 변수를 할당하기 위해 추가 문을 사용하는 것을 줄인 것입니다.

다음으로 도우미 함수 toValidate()를 주목하세요. 이 함수는 현재 처리 중인 인수가 네이티브 JavaScript 유형 일 때 유효성 검사 단계를 우회하는 역할을 합니다(유효성 검사 데코레이터를 첨부할 수 없으므로 유효성 검사 단계를 통해 실행할 이유가 없음).

다음으로, 클래스 변환기 함수인 `plainToInstance()`를 사용하여 일반 JavaScript 인수 객체를 타입이 지정된 객체로 변환하여 유효성 검사를 적용할 수 있도록 합니다. 이 작업을 수행해야 하는 이유는 네트워크 요청에서 역직렬화된 수신 포스트 본문 객체에는 유형 정보가 없기 때문입니다(Express와 같은 기본 플랫폼이 작동하는 방식입니다). 클래스 유효성 검사기는 앞서 DTO에 대해 정의한 유효성 검사 데코레이터를 사용해야 하므로 이 변환을 수행하여 들어오는 본문을 단순한 바닐라 객체가 아닌 적절하게 데코레이션된 객체로 처리해야 합니다.

마지막으로, 앞서 언급했듯이 유효성 검사 파이프이므로 값을 변경하지 않고 반환하거나 예외를 던집니다.

마지막 단계는 `ValidationPipe`를 바인딩하는 것입니다. 파이프는 매개변수 범위, 메서드 범위, 컨트롤러 범위 또는 전역 범위가 될 수 있습니다. 앞서 Joi-기반 유효성 검사 파이프를 사용하여 메서드 수준에서 파이프를 바인딩하는 예제를 살펴봤습니다. 아래 예제에서는 파이프 인스턴스를 라우트 핸들러 `@Body()` 데코레이터에 바인딩하여 파이프를 호출하여 포스트 본문을 유효성 검사하도록 하겠습니다.

```
@@파일명(cats.controller)
@Post()
비동기 생성(
  Body(new ValidationPipe()) createCatDto: CreateCatDto,
) {
  this.catsService.create(createCatDto);
}
```

매개변수 범위 파이프는 유효성 검사 로직이 지정된 매개변수 하나에만 관련될 때 유용합니다. 전역 범위 파이프

프

`ValidationPipe`는 가능한 한 일반적이도록 만들어졌기 때문에 다음과 같은 방법으로 완전한 유용성을 실현할 수 있습니다.

전체 애플리케이션의 모든 라우트 핸들러에 적용되도록 전역 범위 파이프로 설정합니다.

```
@@파일명(메인)
비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}
부트스트랩();
```

경고 **하이브리드 앱의 경우** `useGlobalPipes()` 메서드는 게이트웨이 및 마이크로 서비스에 대한 파이프를 설정하지 않습니다. "표준"(비하이브리드) 마이크로서비스 앱의 경우, `useGlobalPipes()`는 파이프를 전역적으로 마운트합니다.

글로벌 파이프는 모든 컨트롤러와 모든 라우트 핸들러에 대해 전체 애플리케이션에서 사용됩니다.

종속성 주입과 관련하여 모듈 외부에서 등록된 전역 파이프(위 예제에서와 같이 `useGlobalPipes()`를 사용)는 바인딩이 모듈의 컨텍스트 외부에서 이루어졌기 때문에 종속성을 주입할 수 없다는 점에 유의하세요. 이 문제를

해결하기 위해 다음 구성을 사용하여 모든 모듈에서 직접 전역 파이프를 설정할 수 있습니다:

@@파일명(앱.모듈)

'@nestjs/common'에서 { Module }을 가져오고,
'@nestjs/core'에서 { APP_PIPE }를 가져옵니다;

모듈({ providers:

```
[  
  {
```

```
    제공: APP_PIPE, useClass:  
    유효성 검사 파이프,  
    },  
    ],  
})  
내보내기 클래스 AppModule {
```

정보 힌트 이 접근 방식을 사용하여 파이프에 대한 종속성 주입을 수행할 때는 이 구조가 사용되는 모듈에 관계없이 파이프가 실제로는 전역이라는 점에 유의하세요. 이 작업을 어디에서 수행해야 할까요? 파이프가 정의된 모듈(위 예제에서는 [ValidationPipe](#))을 선택하면 됩니다. 또한 사용 클래스만이 사용자 지정 공급자 등록을 처리하는 유일한 방법은 아닙니다. [여기에서](#) 자세히 알아보세요.

내장된 ValidationPipe

다시 한 번 말씀드리자면, ValidationPipe는 Nest에서 기본으로 제공되므로 일반적인 유효성 검사 파이프를 직접 빌드할 필요가 없습니다. 기본 제공 [ValidationPipe](#)는 이 장에서 빌드한 샘플보다 더 많은 옵션을 제공하지만, 사용자 정의 구축 파이프의 메커니즘을 설명하기 위해 기본으로 유지했습니다. 자세한 내용은 [여기에서](#) 많은 예제와 함께 확인할 수 있습니다.

혁신 사용 사례

유효성 검사만이 사용자 정의 파이프의 유일한 사용 사례는 아닙니다. 이 장의 서두에서 파이프를 사용하여 입력 데이터를 원하는 형식으로 변환할 수도 있다고 언급했습니다. 이는 [변환](#) 함수에서 반환된 값이 인수의 이전 값을 완전히 재정의하기 때문에 가능합니다.

언제 유용할까요? 클라이언트에서 전달된 데이터가 라우트 핸들러 메서드에서 제대로 처리되기 전에 문자열을 정수로 변환하는 등 일부 변경을 거쳐야 하는 경우가 있다고 생각해 보세요. 또한 일부 필수 데이터 필드가 누락되어 기본값을 적용하고자 할 수도 있습니다. 변환 파이프는 클라이언트 요청과 요청 핸들러 사이에 처리 함수를 삽입하여 이러한 기능을 수행할 수 있습니다.

다음은 문자열을 정수 값으로 구문 분석하는 간단한 [ParseIntPipe](#)입니다. (위에서 언급했듯이 Nest에는 좀 더 정교한 ParseIntPipe가 내장되어 있으며, 여기서는 사용자 정의 변환 파이프의 간단한 예로 포함시켰습니다).

@@파일명(parse-int.pipe)

import { PipeTransform, 인젝터블, 인자 메타데이터, BadRequestException }를 '@nestjs/common'에서 가져옵니다;

```
@Injectable()
export class ParseIntPipe 구현 PipeTransform<string, number> {
  transform(value: string, metadata: ArgumentMetadata): number {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      새로운 BadRequestException('유효성 검사 실패')을 던집니다;
    }
    반환 값;
  }
}
```

```

}
@@switch
'@nestjs/common'에서 { Injectable, BadRequestException }을 가져옵니다;

@Injectable()
export class ParseIntPipe {
  transform(value, metadata) {
    const val = parseInt(value, 10);
    if (isNaN(val)) {
      새로운 BadRequestException('유효성 검사 실패')을 던집니다;
    }
    반환 값;
  }
}

```

그런 다음 아래와 같이 이 파이프를 선택한 매개변수에 바인딩할 수 있습니다:

```

@@파일명()
@Get(':id')
async findOne(@Param('id', new ParseIntPipe()) id) {
  return this.catsService.findOne(id);
}
@@switch
@Get(':id')
@Bind(Param('id', new ParseIntPipe()))
async findOne(id) {
  this.catsService.findOne(id)을 반환합니다;
}

```

또 다른 유용한 변환 사례는 요청에 제공된 ID를 사용하여 데이터베이스에서 기존 사용자 엔티티를 선택하는 것입니다:

```

@@파일명()
@Get(':id')
findOne(@Param('id', UserByIdPipe) userEntity: UserEntity) {
  return userEntity;
}
@@switch
@Get(':id')
@Bind(Param('id', UserByIdPipe))
findOne(userEntity) {
  사용자 엔티티를 반환합니다;
}

```

이 파이프의 구현은 독자에게 맡기지만, 다른 모든 변환 파이프와 마찬가지로 입력 값(`id`)을 받고 출력 값

(`UserEntity` 객체)을 반환한다는 점에 유의하세요. 이렇게 하면 보일러플레이트 코드를 핸들러에서 공통 파일로 추상화하여 코드를 보다 선언적이고 간결하게 만들 수 있습니다.

기본값 제공

`Parse*` 파이프는 매개변수 값이 정의되어 있을 것으로 기대합니다. `null` 또는 정의되지 않은 값을 받으면 예외가 발생합니다. 엔드포인트에서 누락된 쿼리 문자열 매개변수 값을 처리할 수 있도록 하려면 `Parse*` 파이프가 이러한 값에 대해 작동하기 전에 주입할 기본값을 제공해야 합니다. `DefaultValuePipe`가 바로 그 역할을 합니다. 아래 그림과 같이 관련 `Parse*` 파이프 앞에 `@Query()` 데코레이터에서 `DefaultValuePipe`를 인스턴스화하기만 하면 됩니다:

```
@@파일명() @Get()
비동기 findAll(
    @Query('activeOnly', new DefaultValuePipe(false), ParseBoolPipe)
activeOnly: boolean,
    쿼리('페이지', 새로운 DefaultValuePipe(0), ParseIntPipe) 페이지: 숫자,
) {
    이.catsService.findAll({ activeOnly, page })을 반환합니다;
}
```

경비병

가드는 `@Injectable()` 데코레이터로 주석이 달린 클래스로, `CanActivate`를 구현합니다.

인터페이스.



가드는 단일 책임이 있습니다. 이들은 런타임에 존재하는 특정 조건(권한, 역할, ACL 등)에 따라 특정 요청이 라우트 핸들러에 의해 처리될지 여부를 결정합니다.

이를 흔히 권한 부여라고 합니다. 권한 부여(및 일반적으로 함께 사용되는 사촌인 인증)는 일반적으로 기존 Express 애플리케이션의 [미들웨어에서](#) 처리되었습니다.

토큰 유효성 검사 및 `요청` 개체에 프로퍼티를 첨부하는 것과 같은 작업은 특정 경로 컨텍스트(및 메타데이터)와 밀접하게 연결되어 있지 않으므로 미들웨어는 인증에 적합한 선택입니다.

하지만 미들웨어는 본질적으로 명청합니다. `다음()` 함수를 호출한 후 어떤 핸들러가 실행될지 모릅니다. 반면에 가드는 `ExecutionContext` 인스턴스에 액세스할 수 있으므로 다음에 무엇이 실행될지 정확히 알 수 있습니다. 예외 필터, 파이프, 인터셉터와 마찬가지로 요청/응답 주기의 정확한 지점에 처리 로직을 삽입할 수 있도록 설계되었으며, 이를 선언적으로 수행할 수 있습니다. 따라서 코드를 간결하고 선언적으로 유지하는 데 도움이 됩니다.

정보 힌트 가드는 모든 미들웨어 다음에 실행되지만 인터셉터나 파이프보다 먼저 실행됩니다.

권한 가드

앞서 언급했듯이, 호출자(일반적으로 인증된 특정 사용자)에게 충분한 권한이 있는 경우에만 특정 경로를 사용할 수 있어야 하므로 권한 부여는 가드의 훌륭한 사용 사례입니다. 지금 빌드할 `AuthGuard`는 인증된 사용자를 가정합니다(따라서 요청 헤더에 토큰이 첨부되어 있습니다). 토큰을 추출하여 유효성을 검사하고 추출된 정보를 사용하여 요청을 진행할 수 있는지 여부를 결정합니다.

@@파일명(auth.guard)

'@nestjs/common'에서 { Injectable, CanActivate, ExecutionContext }를 임포트합니다;
'rxjs'에서 { Observable }을 가져옵니다;

@Injectable()

내보내기 클래스 AuthGuard 구현 CanActivate {

canActivate(

컨텍스트입니다: 실행 컨텍스트,

): boolean | Promise<boolean> | Observable<boolean> {
const request = context.switchToHttp().getRequest();
return validateRequest(request);

}

}

@@switch

'@nestjs/common'에서 { Injectable }을 가져옵니다;

@Injectable()

내보내기 클래스 AuthGuard {

async canActivate(context) {

```
    const request = context.switchToHttp().getRequest();
    return validateRequest(request);
}
}
```

정보 힌트 애플리케이션에서 인증 메커니즘을 구현하는 방법에 대한 실제 예제를 찾고 있다면 [이 챕터를 참조하세요](#). 마찬가지로 더 정교한 인증 예제를 보려면 [이 페이지를 확인하세요](#).

유효성 검사 요청() 함수 내부의 로직은 필요에 따라 간단하거나 정교할 수 있습니다. 이 예제의 요점은 가드가 요청/응답 주기에 어떻게 들어맞는지 보여주는 것입니다.

모든 가드는 `canActivate()` 함수를 구현해야 합니다. 이 함수는 현재 요청이 허용되는지 여부를 나타내는 부울을 반환해야 합니다. 이 함수는 응답을 동기식 또는 비동기식([프로미스](#) 또는 익스프레스 라우터를 통해)으로 반환할 수 있습니다. Nest는 반환값을 사용하여 다음 작업을 제어합니다:

- **참을** 반환하면 요청이 처리됩니다. • **거짓을** 반환

하면 Nest는 요청을 거부합니다.

실행 컨텍스트

`canActivate()` 함수는 단일 인자, `ExecutionContext` 인스턴스를 받습니다. `ExecutionContext`는 `ArgumentsHost`를 상속합니다. 앞서 예외 필터 챕터에서 `ArgumentsHost`를 살펴봤습니다. 위의 샘플에서는 이전에 사용한 것과 동일한 헬퍼 메서드를 `ArgumentsHost`에 정의하여 `요청` 객체에 대한 참조를 가져오고 있습니다. 이 주제에 대한 자세한 내용은 [예외 필터](#) 챕터의 `Arguments host` 섹션을 참조하세요.

`ArgumentsHost`를 확장함으로써 `ExecutionContext`는 현재 실행 프로세스에 대한 추가 세부 정보를 제공하는 몇 가지 새로운 헬퍼 메서드도 추가합니다. 이러한 세부 정보는 광범위한 컨트롤러, 메서드 및 실행 컨텍스트에서 작동할 수 있는 보다 일반적인 가드를 구축하는 데 유용할 수 있습니다. 여기에서 `ExecutionContext`에 대해 자세히 알아보세요.

역할 기반 인증

특정 역할을 가진 사용자에게만 액세스를 허용하는 보다 기능적인 가드를 구축해 보겠습니다. 기본 가드 템플릿으로 시작하여 다음 섹션에서 이를 기반으로 구축해 보겠습니다. 지금은 모든 요청이 진행되도록 허용합니다:

@@파일명(roles.guard)

'@nestjs/common'에서 { Injectable, CanActivate, ExecutionContext }를 임포트합니다;

'rxjs'에서 { Observable }을 가져옵니다;

@Injectable()

내보내기 클래스 RolesGuard가 구현하는 캔 액티베이트 { 캔 액티베이

트(

컨텍스트입니다: 실행 컨텍스트,

): boolean | Promise<boolean> | Observable<boolean> { 반환

참입니다;

}

```

}
@@switch
'@nestjs/common'에서 { Injectable }을 가져옵니다;

@Injectable()
export class RolesGuard {
  canActivate(context) {
    참을 반환합니다;
  }
}

```

바인딩 가드

파이프 및 예외 필터와 마찬가지로 가드는 컨트롤러 범위, 메서드 범위 또는 전역 범위로 지정할 수 있습니다. 아래에서는 `@UseGuards()` 데코레이터를 사용하여 컨트롤러 범위 가드를 설정했습니다. 이 데코레이터는 단일 인자 또는 쉼표로 구분된 인자 목록을 사용할 수 있습니다. 이를 통해 하나의 선언으로 적절한 가드 세트를 쉽게 적용할 수 있습니다.

```

@@파일명() @Controller('cats')
@UseGuards(RolesGuard)
내보내기 클래스 CatsController {}

```

정보 힌트 `@UseGuards()` 데코레이터는 `@nestjs/common` 패키지에서 가져옵니다.

위에서는 인스턴스 대신 `RolesGuard` 클래스를 전달하여 인스턴스화에 대한 책임을 프레임워크에 맡기고 의존성 주입을 활성화했습니다. 파이프 및 예외 필터와 마찬가지로 인-플레이스 인스턴스를 전달할 수도 있습니다:

```

@@filename()
@Controller('cats')
@UseGuards(new RolesGuard())
export class CatsController {}

```

위의 구조는 이 컨트롤러가 선언한 모든 핸들러에 가드를 붙입니다. 가드를 단일 메서드에만 적용하려면 메서드 수준에서 `@UseGuards()` 데코레이터를 적용하면 됩니다.

전역 가드를 설정하려면 Nest 애플리케이션 인스턴스의 `useGlobalGuards()` 메서드를 사용합니다:

경고 하이브리드 앱의 경우 `useGlobalGuards()` 메서드는 기본적으로 게이트웨이 및 마이크로 서비스에 대한 가드를 설정하지 않습니다(변경 방법에 대한 자세한 내용은 [하이브리드 애플리케이션](#) 참조).

이 동작). "표준"(비하이브리드) 마이크로서비스 앱의 경우, `useGlobalGuards()`는 가드를 전역적으로 마운트합니다.

글로벌 가드는 모든 컨트롤러와 모든 라우트 핸들러에 대해 전체 애플리케이션에서 사용됩니다. 종속성 주입과 관련하여 모듈 외부에서 등록된 글로벌 가드(위 예제에서와 같이 `useGlobalGuards()`를 사용하여)는 모듈의 컨텍스트 외부에서 수행되므로 종속성을 주입할 수 없습니다. 이 문제를 해결하기 위해 다음 구문을 사용하여 모든 모듈에서 직접 가드를 설정할 수 있습니다:

`@@파일명(앱.모듈)`

```
'@nestjs/common'에서 { Module }을 임포트하고,  
'@nestjs/core'에서 { APP_GUARD }를 임포트합니다;
```

모듈({ providers:

```
[  
  {  
    제공: APP_GUARD,  
    useClass: RolesGuard,  
  },
```

] ,
정보 힌트 이 접근 방식을 사용하여 가드에 대한 종속성 주입을 수행할 때, 이 구조가 사용되는 모듈에 관계 없어 가드는 실제로 전역에 바운드 됨에 유의하세요. 이 작업을 어디에서 수행해야 할까요? 가드가 정의된 모듈 (위 예제에서는 `RolesGuard`)을 선택합니다. 또한 사용 클래스만이 사용자 지정 공급자 등록을 처리하는 유일한 방법은 아닙니다. [여기에서](#) 자세히 알아보세요.

핸들러별 역할 설정

`RolesGuard`가 작동하고 있지만 아직은 그다지 똑똑하지는 않습니다. 가장 중요한 가드 기능인 [실행 컨텍스트](#)를 아직 활용하지 못하고 있습니다. 아직 역할이나 각 핸들러에 허용되는 역할에 대해 알지 못합니다. 예를 들어, `CatsController`는 경로마다 다른 권한 체계를 가질 수 있습니다. 일부는 관리자 사용자만 사용할 수 있고 다른 일부는 모든 사용자에게 개방될 수 있습니다.

유연하고 재사용 가능한 방식으로 역할과 경로를 일치시키려면 어떻게 해야 할까요?

여기서 커스텀 메타데이터가 중요한 역할을 합니다([여기에서](#) 자세히 알아보세요). Nest는 `Reflector#createDecorator` 정적 메서드를 통해 생성된 데코레이터 또는 내장된 `@SetMetadata()` 데코

레이터를 통해 라우트 핸들러에 사용자 정의 메타데이터를 첨부할 수 있는 기능을 제공합니다.

예를 들어, 핸들러에 메타데이터를 첨부하는 `Reflector#createDecorator` 메서드를 사용하여 `@Roles()` 데코레이터를 생성해 보겠습니다. 리플렉터는 프레임워크에서 기본적으로 제공되며 `@nestjs/core` 패키지에서 노출됩니다.

`@@파일명`(역할.데코레이터)

'`@nestjs/core`'에서 `{ Reflector }`를 가져옵니다;

```
export const Roles = Reflector.createDecorator<string[]>();
```

여기서 `Roles` 데코레이터는 문자열[] 타입의 단일 인수를 받는 함수입니다. 이제 이 데코레이터를 사용하려면 핸들러에 주석을 달기만 하면 됩니다:

```
@@파일명(cats.controller)
@Post()
@Roles(['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@스위치

@포스트()
@Roles(['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

여기에서는 관리자 역할이 있는 사용자만 이 경로에 액세스하도록 허용해야 함을 나타내는 역할 데코레이터 메타데이터를 `create()` 메서드에 첨부했습니다.

또는 `Reflector#createDecorator` 메서드를 사용하는 대신 내장된

메타데이터 설정() 데코레이터. [여기에서](#) 자세히 알아보

세요. 모든 것을 종합하기

이제 돌아가서 이것을 `RolesGuard`와 연결해 보겠습니다. 현재는 모든 경우에 참을 반환합니다, 모든 요청이 진행되도록 허용합니다. 현재 사용자에게 할당된 역할과 현재 처리 중인 경로에 필요한 실제 역할을 비교하여 반환값을 조건부로 만들고 싶습니다. 경로의 역할(사용자 지정 메타데이터)에 액세스하기 위해 다음과 같이 `Reflector` 헬퍼 클래스를 다시 사용하겠습니다:

@@파일명(roles.guard)

'@nestjs/common'에서 { Injectable, CanActivate, ExecutionContext }를 임포트합니다;

'@nestjs/core'에서 { Reflector }를 가져오고,

'./roles.decorator'에서 { Roles }를 가져옵니다;

```
@Injectable()
```

```
export class RolesGuard implements CanActivate {
```

```
  constructor(private reflector: Reflector) {}
```

```
  canActivate(context: ExecutionContext): boolean {
```

```
    const roles = this.reflector.get(Roles, context.getHandler());
```

```
    if (!roles) {
```

```

    참을 반환합니다;
}
const request = context.switchToHttp().getRequest();
const user = request.user;
반환 matchRoles(roles, user.roles);
}
}

@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 임포트하고,
'@nestjs/core'에서 { Reflector }를 임포트합니다;
'./roles.decorator'에서 { Roles }를 가져옵니다;

```

주입 가능() `@의존성`(반사기) 내보내기

```

클래스 RolesGuard {
  constructor(reflector) {
    this.reflector = reflector;
  }

  canActivate(context) {
    const roles = this.reflector.get(Roles, context.getHandler());
    if (!roles) {
      참을 반환합니다;
    }
    const request = context.switchToHttp().getRequest();
    const user = request.user;
    반환 matchRoles(roles, user.roles);
  }
}

```

정보 힌트 node.js 세계에서는 권한이 부여된 사용자를 요청 객체에 첨부하는 것이 일반적입니다. 따라서 위의 샘플 코드에서는 `요청.user`에 사용자 인스턴스와 허용된 역할이 포함되어 있다고 가정합니다. 앱에서는 사용자 지정 인증 가드(또는 미들웨어)에서 이러한 연결을 만들 것입니다. 이 주제에 대한 자세한 내용은 [matchRoles\(\)](#) 학습 내부의 로직은 필요에 따라 단순하거나 정교할 수 있습니다. 이 예제의 요점은 가드가 요청/응답 주기에 어떻게 들어맞는지 보여주는 것입니다.

[리플렉션 및 메타데이터](#) 활용에 대한 자세한 내용은 실행 컨텍스트 장의 [리플렉션 및 메타데이터](#) 섹션을 참조하세요.

상황에 맞는 방식으로 [리플렉터](#).

권한이 부족한 사용자가 엔드포인트를 요청하면 Nest는 자동으로 다음과 같은 응답을 반환합니다:

```
{  
    "상태코드": 403,  
    "메시지": "금지된 리소스", "오류": "금지  
됨"  
}
```

백그라운드에서 가드가 거짓을 반환하면 프레임워크는 `ForbiddenException`을 던집니다. 다른 오류 응답을 반환하려면 고유한 예외를 던져야 합니다. 예를 들어

새로운 `UnauthorizedException()`을 던집니다;

가드가 던진 모든 예외는 [예외 계층](#)(전역 예외 필터 및 현재 컨텍스트에 적용되는 모든 예외 필터)에서 처리됩니다.

정보 힌트 인증을 구현하는 방법에 대한 실제 사례를 찾고 있다면 [이 챕터를](#) 확인하세요.

인터셉터

인터셉터는 `@Injectable()` 데코레이터로 주석이 달린 클래스로, `@Injectable()` 데코레이터를 통해 `NestInterceptor` 인터페이스.



인터셉터에는 [측면 지향 프로그래밍\(AOP\)](#) 기법에서 영감을 얻은 유용한 기능들이 있습니다. 인터셉터는 다음을 가능하게 합니다:

- 메서드 실행 전후에 추가 로직을 바인딩
- 함수에서 던져진 예외를 변환
- 기본 함수 동작을 확장합니다.
- 특정 조건에 따라 함수를 완전히 재정의합니다(예: 캐싱 목적) 기본 사항

각 인터셉터는 `인터셉트()` 메서드를 구현하는데, 이 메서드는 두 개의 인수를 받습니다. 첫 번째는 `ExecutionContext` 인스턴스입니다([가드와 정확히 동일한 객체](#)). `ExecutionContext`는 `ArgumentsHost`를 상속합니다. 앞서 예외 필터 챕터에서 `ArgumentsHost`를 살펴봤습니다. 거기서 원래 처리기로 전달된 인수를 감싸는 래퍼이며 애플리케이션 유형에 따라 다른 인자 배열을 포함한다는 것을 보았습니다. 이 주제에 대한 자세한 내용은 [예외 필터를](#) 다시 참조하세요.

실행 컨텍스트

`ArgumentsHost`를 확장함으로써 `ExecutionContext`는 현재 실행 프로세스에 대한 추가 세부 정보를 제공하는 몇 가지 새로운 헬퍼 메서드도 추가합니다. 이러한 세부 정보는 광범위한 컨트롤러, 메서드 및 실행 컨텍스트에서 작동할 수 있는 보다 일반적인 인터셉터를 구축하는 데 유용할 수 있습니다. [여기에서](#) `ExecutionContext`에 대해 자세히 알아보세요.

통화 처리기

두 번째 인자는 `CallHandler`입니다. `CallHandler` 인터페이스는 `handle()` 메서드를 구현하며, 인터셉터의 특정 지점에서 라우트 핸들러 메서드를 호출하는 데 사용할 수 있습니다. `인터셉트()` 메서드 구현에서 `handle()`

메서드를 호출하지 않으면 라우트 핸들러 메서드가 전혀 실행되지 않습니다.

이 접근 방식은 [인터셉트\(\)](#) 메서드가 요청/응답 스트림을 효과적으로 래핑한다는 것을 의미합니다. 따라서 최종 경로 핸들러의 실행 전후에 사용자 정의 로직을 구현할 수 있습니다. [intercept\(\)](#) 메서드에 [handle\(\)](#) 호출 전에 실행되는 코드를 작성할 수 있다는 것은 분명하지만, 그 이후에 일어나는 일에 어떤 영향을 미칠까요? [handle\(\)](#) 메서드는 [Observable](#)을 반환하므로 강력한 RxJS 연산자를 사용하여 응답을 추가로 조작할 수 있습니다. 객체지향 프로그래밍 용어를 사용하면 라우트 핸들러의 호출(즉, [handle\(\)](#) 호출)을 [포인트컷이라고](#) 하며, 이는 추가 로직이 삽입되는 지점임을 나타냅니다.

예를 들어 수신되는 [POST /cats](#) 요청을 생각해 보세요. 이 요청은 [create\(\)](#)

핸들러를 호출합니다. 핸들() 메서드를 호출하지 않는 인터셉터가

를 호출하면 `create()` 메서드가 실행되지 않습니다. `handle()`가 호출되고 해당 `Observable`이 반환되면 `create()` 핸들러가 트리거됩니다. 그리고 `Observable`을 통해 응답 스트림이 수신되면 스트림에서 추가 작업을 수행하고 최종 결과를 호출자에게 반환할 수 있습니다.

측면 차단

첫 번째 사용 사례는 인터셉터를 사용하여 사용자 상호 작용(예: 사용자 호출 저장, 비동기 이벤트 디스패치 또는 타임스탬프 계산)을 기록하는 것입니다. 아래는 간단한 `LoggingInterceptor`를 보여줍니다:

```
@@파일명(logging.interceptor)
'@nestjs/common'에서 { Injectable, NestInterceptor, ExecutionContext,
CallHandler }를 임포트합니다;
'rxjs'에서 { 관찰 가능 } 임포트; 'rxjs/운영
자'에서 { 탭 } 임포트;

@Injectable()
export class LoggingInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
  console.log('Before...');

  const now = Date.now();
  return next
    .handle()
    .pipe(
      tap(() => console.log(`After... ${Date.now() - now}ms`)),
    );
}
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트하고
, 'rxjs'에서 { Observable }을 임포트합니다;
'rxjs/operators'에서 { tap }을 가져옵니다;

@Injectable()
export class LoggingInterceptor {
  intercept(context, next) {
    console.log('Before...');

    const now = Date.now();
    return next
      .handle()
      .pipe(
        tap(() => console.log(`After... ${Date.now() - now}ms`)),
      );
  }
}
```

```
) ;  
}  
}
```

정보 힌트 `NestInterceptor<T, R>`은 일반 인터페이스로, T는 (응답 스트림을 지원하는)

`Observable<T>`의 유형을 나타내고, R은 `Observable<R>`로 래핑된 값의 유형입니다.

경고 주의 컨트롤러, 공급자, 가드 등과 같은 인터셉터는 `생성자를` 통해 종속성을 주입할 수 있습니다.

`handle()`는 RxJS `Observable`을 반환하므로 스트림을 조작하는 데 사용할 수 있는 연산자를 폭넓게 선택할 수 있습니다. 위의 예에서는 관찰 가능한 스트림이 정상적으로 종료되거나 예외적으로 종료될 때 익명 로깅 함수를 호출하지만 그 외에는 응답 주기를 방해하지 않는 `tap()` 연산자를 사용했습니다.

바인딩 인터셉터

인터셉터를 설정하기 위해 `@nestjs/common` 패키지에서 가져온 `@UseInterceptors()` 데코레이터를 사용합니다. [파이프](#) 및 [가드와](#) 마찬가지로 인터셉터는 컨트롤러 범위, 메서드 범위 또는 전역 범위로 설정할 수 있습니다.

```
@@파일명(cats.controller) @사용 인터셉터
```

```
(LoggingInterceptor) 내보내기 클래스
```

```
CatsController {}
```

정보 힌트 `@UseInterceptors()` 데코레이터는 `@nestjs/common` 패키지에서 가져옵니다.

위의 구조를 사용하면 `CatsController`에 정의된 각 라우트 핸들러는 `LoggingInterceptor`를 사용합니다. 누군가 `GET /cats` 엔드포인트를 호출하면 표준 출력에서 다음과 같은 출력을 볼 수 있습니다:

```
Before...
```

```
After... 1ms
```

인스턴스 대신 `LoggingInterceptor` 유형을 전달하여 인스턴스화에 대한 책임을 프레임워크에 맡기고 의존성 주입을 활성화했습니다. 파이프, 가드, 예외 필터와 마찬가지로 인-플레이스 인스턴스도 전달할 수 있습니다:

```
@@filename(cats.controller)
@UseInterceptors(new LoggingInterceptor())
export class CatsController {}
```

앞서 언급했듯이 위의 구조는 이 컨트롤러가 선언한 모든 핸들러에 인터셉터를 첨부합니다. 인터셉터의 범위를 단일 메서드로 제한하려면 메서드 수준에서 데코레이터를 적용하기만 하면 됩니다.

글로벌 인터셉터를 설정하기 위해 Nest 애플리케이션 인스턴스의 `useGlobalInterceptors()` 메서드를 사용합니다:

```
const app = await NestFactory.create(AppModule);
app.useGlobalInterceptors(new LoggingInterceptor());
```

전역 인터셉터는 모든 컨트롤러와 모든 라우트 핸들러에 대해 전체 애플리케이션에서 사용됩니다. 종속성 주입과 관련하여 모듈 외부에서 등록한 전역 인터셉터(위 예제에서와 같이 `useGlobalInterceptors()`를 사용)는 모듈의 컨텍스트 외부에서 수행되므로 종속성을 주입할 수 없습니다. 이 문제를 해결하기 위해 다음 구성을 사용하여 모든 모듈에서 직접 인터셉터를 설정할 수 있습니다:

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/core'에서 { APP_INTERCEPTOR }를 임포트합니다;
```

모듈({ providers:

```
[  
  {  
    제공: APP_INTERCEPTOR,  
    useClass: LoggingInterceptor,  
  },  
],  
})
```

정보 헌트 어제 'AppModule'에 사용하여 인터셉터에 대한 종속성 주입을 수행할 때 이 구조가 사용되는 모듈에 관계없이 인터셉터는 실제로 전역이라는 점에 유의하세요.

이 작업을 어디에서 수행해야 하나요? 인터셉터(위 예제에서는 `LoggingInterceptor`)가 정의된 모듈을 선택하세요. 또한 사용 클래스만이 사용자 지정 공급자 등록을 처리하는 유일한 방법은 아닙니다. [여기에서](#) 자세히 알아보세요.

응답 매핑

우리는 이미 `handle()`가 `Observable`을 반환한다는 것을 알고 있습니다. 스트림에는 라우트 핸들러에서 반환된 값이 포함되어 있으므로 RxJS의 `map()` 연산자를 사용하여 쉽게 변경할 수 있습니다.

경고 경고 응답 매핑 기능은 라이브러리별 응답 전략에서 작동하지 않습니다(`@Res()` 객체를 직접 사용하는 것은 금지됨).

프로세스를 보여주기 위해 각 응답을 간단한 방식으로 수정하는 `TransformInterceptor`를 만들어 보겠습니다.
이 함수는 RxJS의 `map()` 연산자를 사용하여 응답 객체를 새로 생성된 객체의 `데이터` 프로퍼티에 할당하고 새
객체를 클라이언트에 반환합니다.

`@@파일명(transform.interceptor)`

```
'@nestjs/common'에서 { Injectable, NestInterceptor, ExecutionContext,
CallHandler }를 임포트합니다;
'rxjs'에서 { 관찰 가능 } 임포트; 'rxjs/운영자
'에서 { 지도 } 임포트;
```

```

내보내기 인터페이스 Response<T> {
  data: T;
}

@Injectable()
export class TransformInterceptor<T> 구현 NestInterceptor<T, Response<T>>
{
  인터셉트(context: ExecutionContext, next: CallHandler):
  Observable<Response<T>> {
    반환 다음.핸들().파이프(맵(데이터 => ({ 데이터 })));
  }
}

@@switch
'@nestjs/common'에서 { Injectable }을 임포트하고,
'rxjs/operators'에서 { map }을 임포트합니다;

@Injectable()
export class TransformInterceptor {
  intercept(context, next) {
    반환 다음.핸들().파이프(맵(데이터 => ({ 데이터 })));
  }
}

```

정보 힌트 네스트 인터셉터는 동기 및 비동기 [인터셉트\(\)](#) 메서드 모두에서 작동합니다. 필요한 경우 메서드를 [비동기로](#) 전환하면 됩니다.

위의 구성을 사용하면 누군가 [GET /cats](#) 엔드포인트를 호출하면 다음과 같은 응답이 표시됩니다(라우트 핸들러가 빈 배열 []을 반환한다고 가정):

```
{
  "데이터": []
}
```

인터셉터는 전체 애플리케이션에서 발생하는 요구사항에 대해 재사용 가능한 솔루션을 만드는데 큰 가치가 있습니다. 예를 들어, [널](#) 값의 각 발생을 빈 문자열 [''](#)로 변환해야 한다고 가정해 보겠습니다. 한 줄의 코드를 사용하여 이 작업을 수행하고 인터셉터를 전역적으로 바인딩하여 등록된 각 핸들러에서 자동으로 사용하도록 할 수 있습니다.

@@파일명()

'@nestjs/common'에서 { Injectable, NestInterceptor, ExecutionContext, CallHandler }를 임포트합니다;

'rxjs'에서 { 관찰 가능 } 임포트; 'rxjs/운영자'에서 { 지도 } 임포트;

@Injectable()

```
export class ExcludeNullInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
```

```
    다음 반환
    .handle()
    .pipe(map(value => value === null ? '' : value ));
}
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트하고,
'rxjs/operators'에서 { map }을 임포트합니다;

@Injectable()
export class ExcludeNullInterceptor {
  intercept(context, next) {
    다음 반환
    .handle()
    .pipe(map(value => value === null ? '' : value ));
  }
}
```

예외 매팅

또 다른 흥미로운 사용 사례는 RxJS의 `catchError()` 연산자를 활용하여 던져진 예외를 재정의하는 것입니다:

@@파일명(errors.interceptor) 가져오

```
기 {
    Injectable,
    NestInterceptor,
    ExecutionContext,
    BadGatewayException,
    CallHandler,
}를 '@nestjs/common'에서 가져옵니다;
'rxjs'에서 { Observable, throwError }를 임포트하고,
'rxjs/operators'에서 { catchError }를 임포트합니다;

@Injectable()
내보내기 클래스 ErrorsInterceptor 구현 NestInterceptor { intercept(context:
    ExecutionContext, next: CallHandler): Observable<any>
{
    다음 반환
    .handle()
    .pipe(
        catchError(err => throwError(() => new BadGatewayException())),
    );
}
}
@switch
'@nestjs/common'에서 { Injectable, BadGatewayException }을 임포트하고,
'rxjs'에서 { throwError }를 임포트합니다;
'rxjs/operators'에서 { catchError }를 가져옵니다;

@Injectable()
```

```
export class ErrorsInterceptor {
  intercept(context, next) {
    다음 반환
    .handle()
    .pipe(
      catchError(err => throwError(() => new BadGatewayException())),
    );
  }
}
```

스트림 재정의

핸들러 호출을 완전히 방지하고 대신 다른 값을 반환하는 데에는 몇 가지 이유가 있습니다. 응답 시간을 개선하기 위해 캐시를 구현하는 것이 대표적인 예입니다. 캐시에서 응답을 반환하는 간단한 캐시 인터셉터를 살펴보겠습니다. 현실적인 예제에서는 TTL, 캐시 무효화, 캐시 크기 등과 같은 다른 요소도 고려해야 하지만 이는 이 논의의 범위를 벗어납니다. 여기서는 주요 개념을 설명하는 기본 예제를 제공하겠습니다.

@@파일명(캐시.인터셉터)

'@nestjs/common'에서 { Injectable, NestInterceptor, ExecutionContext, CallHandler }를 **임포트합니다**;
'rxjs'에서 { Observable, of }를 **가져옵니다**;

@Injectable()

내보내기 클래스 CacheInterceptor 구현

```
NestInterceptor { intercept(context: ExecutionContext, next: CallHandler): Observable<any>
{
    const isCached = true;
    if (isCached) {
        의 반환([]);
    }
    다음.핸들()을 반환합니다;
}
```

@@switch

'@nestjs/common'에서 { Injectable }을 **임포트하고**,
'rxjs'에서 { of }을 **임포트합니다**;

@Injectable()

```
export class CacheInterceptor {
    intercept(context, next) {
        const isCached = true;
        if (isCached) {
            의 반환([]);
        }
        다음.핸들()을 반환합니다;
    }
}
```

캐시인터셉터에는 하드코딩된 `isCached` 변수와 하드코딩된 응답 []도 있습니다. 여기서 주목해야 할 핵심 사항은 RxJS의() 연산자에 의해 생성된 새 스트림을 반환하므로 라우트 핸들러가 전혀 호출되지 않는다는 것입니다. 누군가 `CacheInterceptor`를 사용하는 앤드포인트를 호출하면 응답(하드코딩된 빈 배열)이 즉시 반환됩니다. 일반적인 솔루션을 만들기 위해 `Reflector`를 활용하고 사용자 정의 데코레이터를 만들 수 있습니다. 리플렉터는 [가드](#) 챕터에 잘 설명되어 있습니다.

더 많은 운영자

RxJS 연산자를 사용하여 스트림을 조작할 수 있는 가능성은 우리에게 많은 기능을 제공합니다. 또 다른 일반적인 사용 사례를 고려해 봅시다. 경로 요청에 대한 시간 초과를 처리하고 싶다고 가정해 보겠습니다. 일정 시간이 지나도 앤드포인트에서 아무 것도 반환하지 않으면 오류 응답으로 종료하고 싶을 것입니다. 다음 구조가 이를 가능하게 합니다:

```
@@파일명(timeout.interceptor)
'@nestjs/common'에서 { Injectable, NestInterceptor, ExecutionContext,
CallHandler, RequestTimeoutException }을 임포트합니다;
'rxjs'에서 { 관찰 가능, throwError, 시간 초과 오류 }를 임포트하고,
'rxjs/운영자'에서 { catchError, 시간 초과 }를 임포트합니다;

@Injectable()
export class TimeoutInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    return next.handle().pipe(
      timeout(5000),
      catchError(err => {
        if (err 인스턴스 오브 타임아웃 에러) {
          반환 throwError(() => new RequestTimeoutException());
        }
        반환 throwError(() => err);
      }),
    );
  };
}
@@switch
'@nestjs/common'에서 { Injectable, RequestTimeoutException }을 임포트하고,
'rxjs'에서 { Observable, throwError, TimeoutError }를 임포트합니다;
'rxjs/operators'에서 { catchError, timeout }을 가져옵니다;

@Injectable()
export class TimeoutInterceptor {
  intercept(context, next) {
    return next.handle().pipe(
```

```
timeout(5000),  
catchError(err => {  
  if (err 인스턴스 오브 타임아웃 에러) {  
    반환 throwError(() => new RequestTimeoutException());  
  }  
  반환 throwError(() => err);  
}),
```

```
) ;  
};  
};
```

5초가 지나면 요청 처리가 취소됩니다. 또한 사용자 지정 로직을 추가한 후

요청 시간 초과 예외(예: 리소스 릴리스).

사용자 지정 경로 데코레이터

Nest는 데코레이터라는 언어 기능을 중심으로 구축되었습니다. 데코레이터는 일반적으로 사용되는 많은 프로그래밍 언어에서 잘 알려진 개념이지만 자바스크립트 세계에서는 아직 비교적 새로운 개념입니다. 데코레이터의 작동 방식을 더 잘 이해하려면 [이 글을](#) 읽어보시기 바랍니다. 다음은 간단한 정의입니다:

ES2016 데코레이터는 함수를 반환하고 대상, 이름 및 속성 설명자를 인수로 받을 수 있는 표현식입니다. 데코레이터 앞에 `@` 문자를 붙여서 데코레이터를 적용하고 데코레이션하려는 항목의 맨 위에 배치하면 됩니다. 데코레이터는 클래스, 메서드 또는 프로퍼티에 대해 정의할 수 있습니다.

매개변수 데코레이터

Nest는 HTTP 라우트 핸들러와 함께 사용할 수 있는 유용한 매개변수 데코레이터 세트를 제공합니다.

다음은 제공된 데코레이터와 해당 데코레이터가 나타내는 일반 Express(또는 Fastify) 객체 목록입니다.

요청()	req
응답(), @Res()	res
@Next()	다음
세션()	req.session
@Param(param?: 문자열)	req.params / req.params[param]
@Body(param?: 문자열)	req.body / req.body[param]
@Query(param?: 문자열)	req.query / req.query[param]
@Headers(param?: 문자열)	req.headers / req.headers[param]
@Ip()	req.ip
호스트 파라미터()	req.hosts

또한 나만의 맞춤 데코레이터를 만들 수도 있습니다. 이것이 왜 유용한가요?

node.js 세계에서는 요청 객체에 프로퍼티를 첨부하는 것이 일반적인 관행입니다. 그런 다음 다음과 같은 코드를 사용하여 각 라우트 핸들러에서 프로퍼티를 수동으로 추출합니다:

```
const user = req.user;
```

코드를 더 읽기 쉽고 투명하게 만들기 위해 `@User()` 데코레이터를 생성하여 모든 컨트롤러에서 재사용할 수 있습니다.

`@@파일명`(사용자.데코레이터)

'@nestjs/common'에서 `{ createParamDecorator, ExecutionContext }`를 임포트합니다;

```
export const User = createParamDecorator(
```

```
(데이터: 알 수 없음, ctx: 실행 컨텍스트) => {
    const request = ctx.switchToHttp().getRequest();
    return request.user;
},
);
```

그런 다음 요구 사항에 맞는 곳에서 간단히 사용할 수 있습니다.

```
@@파일명() @Get()
async findOne(@User() user: UserEntity) {
    console.log(user);
}

스위치 @Get()
@Bind(User())
async findOne(user) {
    console.log(user);
}
```

데이터 전달

데코레이터의 동작이 특정 조건에 따라 달라지는 경우 데이터 매개 변수를 사용하여 데코레이터의 팩토리 함수에 인수를 전달할 수 있습니다. 이에 대한 한 가지 사용 사례는 요청 객체에서 키별로 속성을 추출하는 사용자 정의 데코레이터입니다. 예를 들어 [인증 계층이](#) 요청의 유효성을 검사하고 사용자 엔티티를 요청 개체에 첨부한다고 가정해 보겠습니다. 인증된 요청에 대한 사용자 엔티티는 다음과 같을 수 있습니다:

```
{
    "id": 101,
    "이름": "앨런", "성": "튜링",
    "이메일": "alan@email.com",
    "roles": ["admin"]
}
```

프로퍼티 이름을 키로 받아 프로퍼티가 존재하면 관련 값을 반환하고, 존재하지 않거나 사용자 객체가 생성되지 않은 경우 정의되지 않은 값을 반환하는 데코레이터를 정의해 보겠습니다.

@@파일명(사용자.데코레이터)

'@nestjs/common'에서 { createParamDecorator, ExecutionContext }를 임포트합니다;

```
export const User = createParamDecorator(  
  (data: string, ctx: ExecutionContext) => {  
    const request = ctx.switchToHttp().getRequest();  
    const user = request.user;
```

```

반환 데이터 ? 사용자?.[데이터] : 사용자;
},
);
@@switch
'@nestjs/common'에서 { createParamDecorator }를 가져옵니다;

export const User = createParamDecorator((data, ctx) => {
  const request = ctx.switchToHttp().getRequest();
  const user = request.user;

  반환 데이터 ? 사용자 && 사용자[데이터] : 사용자;
});

```

컨트롤러의 `@User()` 데코레이터를 통해 특정 프로퍼티에 액세스하는 방법은 다음과 같습니다:

```

@@파일명() @Get()
async findOne(@User('firstName') firstName: string) {
  console.log(`Hello ${firstName}`);
}

@@스위치
@Get()
@Bind(User('firstName'))
async findOne(firstName) {
  console.log(`Hello ${firstName}`);
}

```

동일한 데코레이터를 다른 키와 함께 사용하여 다른 프로퍼티에 액세스할 수 있습니다. `사용자` 객체가 깊거나 복잡한 경우 요청 핸들러 구현을 더 쉽고 가독성 있게 만들 수 있습니다.

정보 힌트 타입스크립트 사용자의 경우, `createParamDecorator<T>()`는 제네릭이라는 점에 유의하세요. 즉, `createParamDecorator<string>((data, ctx) => ...)`와 같이 명시적으로 유형 안전을 적용할 수 있습니다. 또는 팩토리 함수에서 매개변수 유형을 지정할 수 있습니다(예):

`createParamDecorator((data: string, ctx) => ...).` 둘 다 생략하면 `데이터` 유형은 아무 것이나 됩니다.

파이프 작업

Nest는 사용자 정의 매개변수 데코레이터를 기본 제공 매개변수(`@Body()`, `@Param()` 및 `@Query()`)와 동일한 방식으로 처리합니다. 즉, 사용자 정의 주석이 달린 매개변수(예제에서는 `사용자` 인수)에 대해서도 파이프가 실행됩니다.

다. 또한 파이프를 사용자 정의 데코레이터에 직접 적용할 수도 있습니다:

```
@@파일명() @Get()
async findOne(
    사용자(새로운 유효성 검사 파이프({ 유효성 검사 사용자: true })) 사용자:
    UserEntity,
```

```

) {
  콘솔 로그(사용자);
}

@@스위치
@Get()
바인드(User(new ValidationPipe({ validateCustomDecorators: true })))
async findOne(user) {
  콘솔 로그(사용자);
}

```

정보 힌트 유효성 검사 사용자 정의 데코레이터 옵션을 true로 설정해야 합니다. 유효성 검사 파이프는 기본적으로 사용자 지정 데코레이터로 주석 처리된 인수의 유효성을 검사하지 않습니다.

데코레이터 구성

Nest는 여러 데코레이터를 구성하는 헬퍼 메서드를 제공합니다. 예를 들어 인증과 관련된 모든 데코레이터를 하나의 데코레이터로 결합하고 싶다고 가정해 보겠습니다. 다음과 같은 구성을 통해 이를 수행할 수 있습니다:

```

@@파일명(auth.decorator)
'@nestjs/common'에서 { applyDecorators }를 가져옵니다;

export function Auth(...roles: Role[]) {
  return applyDecorators(
    SetMetadata('roles', roles),
    UseGuards(AuthGuard, RolesGuard),
    ApiBearerAuth(),
    ApiUnauthorizedResponse({ description: 'Unauthorized' }),
  );
}

@@switch
'@nestjs/common'에서 { applyDecorators }를 가져옵니다;

export function Auth(...roles) {
  return applyDecorators(
    SetMetadata('roles', roles),
    UseGuards(AuthGuard, RolesGuard),
    ApiBearerAuth(),
    ApiUnauthorizedResponse({ description: 'Unauthorized' }),
  );
}

```

그런 다음 이 사용자 정의 `@Auth()` 데코레이터를 다음과 같이 사용할 수 있습니다:

```
Get('users')
@Auth('admin')
findAllUsers() {}
```

이렇게 하면 한 번의 선언으로 네 가지 데코레이터를 모두 적용하는 효과가 있습니다.

경고 경고 `@nestjs/swagger` 패키지의 `@ApiHideProperty()` 데코레이터는 컴포저블이 불가능하며 `applyDecorators` 함수에서 제대로 작동하지 않습니다.

사출 범위

다른 프로그래밍 언어 배경을 가진 사람들에게는 Nest에서 거의 모든 것이 들어오는 요청에서 공유된다는 사실이 의외로 느껴질 수 있습니다. 데이터베이스에 대한 연결 풀, 전역 상태를 가진 싱글톤 서비스 등이 있습니다. Node.js는 모든 요청이 별도의 스레드에서 처리되는 요청/응답 다중 스레드 상태 비저장 모델을 따르지 않는다는 점을 기억하세요. 따라서 싱글톤 인스턴스를 사용하는 것은 애플리케이션에 완전히 안전합니다.

그러나 요청 기반 수명이 바람직한 동작일 수 있는 에지 케이스(예: GraphQL 애플리케이션의 요청별 캐싱, 요청 추적, 멀티테넌시)가 있습니다. 주입 범위는 원하는 공급자 수명 동작을 얻을 수 있는 메커니즘을 제공합니다.

공급자 범위

공급자는 다음 범위 중 하나를 가질 수 있습니다:

기본값 공급자의 단일 인스턴스가 전체 애플리케이션에서 공유됩니다. 인스턴스 수명은 애플리케이션 수명 주기에 직접 연결됩니다. 애플리케이션이 부트스트랩되면 모든 싱글톤 공급자가 인스턴스화됩니다. 기본적으로 싱글톤 범위가 사용됩니다.

요청 들어오는 각 요청에 대해 공급자의 새 인스턴스가 독점적으로 생성됩니다. 인스턴스는 요청 처리 **TRANSIENT** 가 완료된 후 가비지 수집됩니다.

임시 공급자는 소비자 간에 공유되지 않습니다. 임시 공급자를 주입하는 각 소비자는 새로운 전용 인스턴스를 받게 됩니다.

정보 힌트 대부분의 사용 사례에는 싱글톤 범위를 사용하는 것이 좋습니다. 소비자 및 요청 간에 공급자를 공유하면 인스턴스를 캐시할 수 있고 애플리케이션 시작 중에 한 번만 초기화가 수행됩니다.

사용법

범위 속성을 `@Injectable()` 데코레이터 옵션 객체에 전달하여 주입 범위를 지정합니다:

```
'@nestjs/common'에서 { Injectable, Scope } import; @Injectable({  
  scope: Scope.REQUEST })  
내보내기 클래스 CatsService {}
```

마찬가지로 사용자 지정 공급업체의 경우 공급자 등록에 대한 장문 양식에서 범위 속성을 설정합니다:

```
{  
    제공: 'CACHE_MANAGER',  
    useClass: CacheManager,  
    scope: Scope.TRANSIENT,  
}
```

정보 힌트 `@nestjs/common`에서 범위 열거형 가져오기

싱글톤 범위는 기본적으로 사용되며 선언할 필요가 없습니다. 공급자를 싱글톤 범위로 선언하려면 **범위 속성에 Scope.DEFAULT 값을 사용하세요.**

경고 웹소켓 게이트웨이는 싱글톤으로 작동해야 하므로 요청 범위가 지정된 공급자를 사용해서는 안 됩니다. 각 게이트웨이는 실제 소켓을 캡슐화하며 여러 번 인스턴스화할 수 없습니다. 이 제한은 [패스포트](#) 전략이나 [크론 컨트롤러](#)와 같은 일부 다른 공급자에게도 적용됩니다.

컨트롤러 범위

컨트롤러는 해당 컨트롤러에 선언된 모든 요청 메서드 핸들러에 적용되는 스코프를 가질 수도 있습니다. 공급자 범위와 마찬가지로 컨트롤러의 범위는 수명을 선언합니다. 요청 범위 컨트롤러의 경우 각 인바운드 요청에 대해 새 인스턴스가 생성되고 요청이 처리를 완료하면 가비지 수집됩니다.

`ControllerOptions` 객체의 **범위 속성**을 사용하여 컨트롤러 범위를 선언합니다:

```
컨트롤러({ 경로:  
  'cats',  
  scope: Scope.REQUEST,  
})  
내보내기 클래스 CatsController {}
```

범위 계층 구조

요청 범위는 인젝션 체인에 버블을 형성합니다. 요청 범위가 지정된 공급자에 의존하는 컨트롤러는 그 자체로 요청 범위가 지정됩니다.

다음 종속성 그래프를 상상해 보세요: `CatsController <- CatsService <- CatsRepository`.

`CatsService`가 요청 범위가 지정되어 있고 나머지는 기본 싱글톤인 경우, 주입된 서비스에 종속되어 있기 때문에 `CatsController`는 요청 범위가 지정됩니다. 종속적이지 않은 `CatsRepository`는 싱글톤 범위로 유지됩니다.

일시적 범위의 종속성은 이러한 패턴을 따르지 않습니다. 싱글톤 범위의 `DogsService`가 일시적인 `LoggerService` 프로바이더를 주입하면 새로운 인스턴스를 받게 됩니다. 그러나 `DogsService`는 싱글톤 범

위로 유지되므로 아무 곳에나 주입해도 DogsService의 새 인스턴스로 해결되지 않습니다. 이러한 동작을 원한다면 DogsService도 명시적으로 TRANSIENT로 표시해야 합니다.

요청 공급자

HTTP 서버 기반 애플리케이션(예: [@nestjs/platform-express](#) 또는 [@nestjs/platform-fastify](#) 사용)에서는 요청 범위가 지정된 공급자를 사용할 때 원본 요청 객체에 대한 참조에 액세스하고 싶을 수 있습니다. REQUEST 객체를 주입하면 이 작업을 수행할 수 있습니다.

```
'@nestjs/common'에서 { Injectable, Scope, Inject }를 임포트하고,  
'@nestjs/core'에서 { REQUEST }를 임포트합니다;  
'express'에서 { Request }를 가져옵니다;
```

주입 가능({ 범위: Scope.REQUEST }) 내보내기 클

```
래스 CatsService {  
    생성자(@Inject(REQUEST) 비공개 요청: Request) {}  
}
```

기본 플랫폼/프로토콜 차이로 인해 마이크로서비스 또는 GraphQL 애플리케이션의 경우 인바운드 요청에 약간 다르게 액세스합니다. GraphQL 애플리케이션에서는 REQUEST 대신 CONTEXT를 주입합니다:

```
'@nestjs/common'에서 { Injectable, Scope, Inject }를 가져오고,  
'@nestjs/graphql'에서 { CONTEXT }를 가져옵니다;
```

주입 가능({ 범위: Scope.REQUEST }) 내보내기 클

```
래스 CatsService {  
    constructor(@Inject(CONTEXT) private context) {}  
}
```

그런 다음 요청을 속성으로 포함하도록 컨텍스트 값(GraphQLModule에서)을 구성합니다. 인콰이어러 공급자

예를 들어 로깅 또는 메트릭 공급자에서 공급자가 생성된 클래스를 가져오고자 하는 경우,
를 입력하면 INQUIRER 토큰을 주입할 수 있습니다.

```
'@nestjs/common'에서 { Inject, Injectable, Scope }를 임포트하고,  
'@nestjs/core'에서 { INQUIRER }를 임포트합니다;
```

주사 가능({ 범위: Scope.TRANSIENT }) 내보내기

```
래스 HelloService {  
    constructor(@Inject(INQUIRER) private parentClass: object) {}  
  
    sayHello(message: 문자열) {  
        console.log(`this.parentClass?.constructor?.name}: ${message}`);  
    }  
}
```

그런 다음 다음과 같이 사용하세요:

'@nestjs/common'에서 { Injectable }을 임포트하고,
'./hello.service'에서 { HelloService }를 임포트합니다;

```
@Injectable()  
내보내기 클래스 AppService {
```

```

constructor(private helloService: HelloService) {}

getRoot(): 문자열 {
    this.helloService.sayHello('내 이름은 getRoot입니다');

    'Hello world!'를 반환합니다;
}
}

```

위의 예제에서 AppService#getRoot가 호출되면 "AppService: 내 이름은 getRoot입니다." 가 콘솔에 기록됩니다. 성능

요청 범위가 지정된 공급자를 사용하면 애플리케이션 성능에 영향을 미칩니다. Nest가 캐싱을 시도하는 동안 를 최대한 많은 메타데이터로 설정해도 각 요청마다 클래스의 인스턴스를 생성해야 합니다. 따라서 평균 응답 시간과 전반적인 벤치마킹 결과가 느려집니다. 공급자가 요청 범위를 지정해야 하는 경우가 아니라면 기본 싱글톤 범위를 사용하는 것이 좋습니다.

정보 힌트 상당히 위협적으로 들리지만, 요청 범위가 지정된 공급자를 활용하는 적절하게 설계된 애플리케이션은 자연 시간이 최대 5% 이상 느려지지 않아야 합니다.

내구성 있는 공급자

위 섹션에서 언급했듯이 요청 범위가 지정된 공급자를 하나 이상(컨트롤러 인스턴스에 주입되거나 더 깊게는 공급자 중 하나에 주입됨) 사용하면 컨트롤러도 요청 범위가 지정되므로 자연 시간이 늘어날 수 있습니다. 즉, 각 개별 요청마다 컨트롤러를 다시 생성(인스턴스화)해야 하고 나중에 가비지 컬렉션을 해야 합니다. 즉, 3만 개의 요청이 병렬로 발생한다고 가정하면 컨트롤러(및 요청 범위가 지정된 공급자)의 임시 인스턴스가 3만 개가 된다는 뜻이기도 합니다.

대부분의 공급자가 의존하는 공통 공급자(데이터베이스 연결 또는 로거 서비스를 생각해보세요)가 있으면 모든 공급자가 자동으로 요청 범위 공급자로 변환됩니다. 이렇게 하면 멀티테넌트 애플리케이션, 특히 중앙 요청 범위가 '데이터'인 애플리케이션의 경우 더욱 그렇습니다. "소스" 공급자는 요청 객체에서 헤더/토큰을 가져와서 그 값을 기반으로 해당 데이터베이스 연결/스키마(해당 테넌트에만 해당).

예를 들어 10명의 고객이 번갈아 사용하는 애플리케이션이 있다고 가정해 보겠습니다. 각 고객마다 고유한 전용 데이터 소스가 있으며, 고객 A가 고객 B의 데이터베이스에 절대 접근할 수 없도록 하고 싶을 것입니다. 이를 달성하는 한 가지 방법은 요청 개체를 기반으로 '현재 고객'이 무엇인지 판단하고 해당 데이터베이스를 검색하는 요청 범

위가 지정된 '데이터 소스' 공급자를 선언하는 것입니다. 이 접근 방식을 사용하면 단 몇 분 만에 애플리케이션을 멀티테넌트 애플리케이션으로 전환할 수 있습니다. 하지만 이 접근 방식의 가장 큰 단점은 애플리케이션 구성 요소의 상당 부분이 "데이터 소스" 공급자에 의존할 가능성이 높기 때문에 암시적으로 "요청 범위"가 지정되므로 앱 성능에 영향을 미칠 수 있다는 것입니다.

하지만 더 나은 솔루션이 있다면 어떨까요? 고객이 10명뿐이므로 요청마다 각 트리를 다시 만드는 대신 고객당 10개의 개별 [DI 하위 트리를](#) 가질 수는 없을까요? 공급자가 각 연속 요청에 대해 진정으로 고유한 속성(예: 요청 UUID)에 의존하지 않고 대신 몇 가지

특정 속성을 집계(분류)할 수 있다면 들어오는 모든 요청에 대해 DI 하위 트리를 다시 생성할 이유가 없습니다.

바로 이때 내구성 있는 공급업체가 유용합니다.

공급자를 내구성이 있는 것으로 플래그를 지정하기 전에 먼저 Nest에 "공통 요청 속성"이 무엇인지 알려주는 전략을 등록하고 요청을 그룹화하는 로직을 제공하여 해당 DI 하위 트리와 연결해야 합니다.

내구성 설정하기

가져 오기 { 호스트 컴포넌트

정보, 컨텍스트 ID, 컨텍스

트 ID 팩토리, 컨텍스트 ID

전략,

}를 '@nestjs/core'에서 가져옵니다;

'express'에서 { Request }를 가져옵니다;

```
const tenants = 새로운 맵<스트링, 컨텍스트아이디>();
```

내보내기 클래스 AggregateByTenantContextIdStrategy 는

ContextIdStrategy 를 구현합니다 {

```
attach(contextId: ContextId, request: 요청) {
```

```
  const tenantId = request.headers['x-tenant-id'] as string;
```

```
  let tenantSubTreeId: ContextId;
```

```
  if (tenants.has(tenantId)) {
```

```
    tenantSubTreeId = tenants.get(tenantId);
```

```
  } else {
```

```
    tenantSubTreeId = ContextIdFactory.create();
```

```
    tenants.set(tenantId, tenantSubTreeId);
```

```
}
```

// 트리가 내구성이 없는 경우 원래 "contextId" 객체를 반환 반환 (정보:

[HostComponentInfo](#)) =>.

```
  info.isTreeDurable ? tenantSubTreeId : contextId;
```

정복 힌트 요청 범위와 유사하게, 내구성은 주입 체인에 버블을 형성합니다. 즉, A가 내구성으로 플래그가

지정된 B에 종속된 경우 A도 암시적으로 내구성 있게 됩니다(A 공급자에 대해 내구성이 명시적으로 거

경고 경로에 천재한 양을 줄여 테넌트로 운영되는 애플리케이션에는 적합하지 않습니다.

attach 메서드에서 반환된 값은 주어진 호스트에 대해 어떤 컨텍스트 식별자를 사용해야 하는지 Nest에 지시합니다. 이 예제에서는 호스트 컴포넌트(예: 요청 범위 컨트롤러)가 내구성으로 플래그가 지정된 경우 원래의

자동 생성된 `contextId` 객체 대신 `tenantSubTreeId`를 사용하도록 지정했습니다(아래에서 공급자를 내구성으로 표시하는 방법을 확인할 수 있습니다). 또한, 위의 예시에서는 페이로드가 없습니다.

가 등록될 것입니다(여기서 페이로드 = 하위 트리의 부모인 "루트"를 나타내는 요청/컨텍스트 공급자).

내구성 있는 트리에 페이로드를 등록하려면 다음 구문을 대신 사용하세요:

```
// `AggregateByTenantContextIdStrategy#attach` 메서드의 반환: 반환 {{  
    resolve: (info: HostComponentInfo) =>  
        info.isTreeDurable ? tenantSubTreeId : contextId,  
        페이로드: { tenantId },  
    }  
}
```

이제 `@Inject(REQUEST)/@Inject(CONTEXT)`를 사용하여 요청 공급자(또는 GraphQL 애플리케이션의 경우 CONTEXT)를 주입할 때마다 페이로드 객체가 주입됩니다(단일 속성(이 경우 `tenantId`)으로 구성됨).

이 전략이 마련되면 코드 어딘가에 등록할 수 있으므로(어쨌든 전 세계적으로 적용되므로) 예를 들어 `main.ts` 파일에 배치할 수 있습니다:

```
ContextIdFactory.apply(new AggregateByTenantContextIdStrategy());
```

정보 힌트 `ContextIdFactory` 클래스는 `@nestjs/core` 패키지에서 임포트됩니다.

요청이 애플리케이션에 도달하기 전에 등록이 이루어지면 모든 것이 의도한 대로 작동합니다.

마지막으로 일반 프로바이더를 내구성 프로바이더로 전환하려면 내구성 플래그를 `true`로 설정하고 해당 범위를 `Scope.REQUEST`로 변경하면 됩니다(`REQUEST` 범위가 이미 인젝션 체인에 있는 경우 필요 없음):

```
'@nestjs/common'에서 { Injectable, Scope }를 가져옵니다;
```

```
주입 가능({ 범위: Scope.REQUEST, 내구성: true }) 내보내기 클래스  
CatsService {}
```

마찬가지로 [사용자 지정](#) 공급업체의 경우 공급자 등록을 위한 장문 양식에서 내구성 속성을 설정합니다:

```
{  
    제공: 'foobar', useFactory:  
        () => { ... }, scope:  
        Scope.REQUEST, durable:  
        true,  
}
```

비동기 공급자

하나 이상의 비동기 작업이 완료될 때까지 애플리케이션 시작을 지연시켜야 하는 경우가 있습니다. 예를 들어 데이터베이스와의 연결이 설정될 때까지 요청 수락을 시작하지 않으려는 경우가 있습니다. 비동기 공급자를 사용하여 이를 달성할 수 있습니다.

이를 위한 구문은 `useFactory` 구문과 함께 `async/await`을 사용하는 것입니다. 팩토리는 프로미스를 반환하고 팩토리 함수는 비동기 작업을 대기할 수 있습니다. Nest는 이러한 프로바이더에 의존하는(주입하는) 클래스를 인스턴스화하기 전에 프로미스의 해결을 기다립니다.

```
{  
  제공: 'async_connection',  
  useFactory: async () => {  
    const connection = await createConnection(options);  
    반환 연결;  
  },  
}
```

정보 힌트 [여기에서](#) 사용자 지정 공급자 구문에 대해 자세히 알아보세요.

주입

비동기 공급자는 다른 공급자와 마찬가지로 토큰을 통해 다른 컴포넌트에 주입됩니다. 위의 예시에서는 `@Inject('ASYNC_CONNECTION')` 구문을 사용합니다.

예

[TypeORM 레시피](#)에는 비동기 공급자에 대한 보다 실질적인 예시가 있습니다.

동적 모듈

모듈 장에서는 네스트 모듈의 기본 사항을 다루고 동적 모듈에 대한 간략한 소개를 포함합니다. 이 장에서는 동적 모듈에 대한 주제를 확장합니다. 이 장이 끝나면 동적 모듈이 무엇이며 언제 어떻게 사용하는지 잘 이해하게 될 것입니다.

소개

문서의 개요 섹션에 있는 대부분의 애플리케이션 코드 예제는 일반 모듈 또는 정적 모듈을 사용합니다. 모듈은 전체 애플리케이션의 모듈식 부분으로 서로 맞는 공급자 및 컨트롤러와 같은 구성 요소 그룹을 정의합니다. 모듈은 이러한 컴포넌트에 대한 실행 컨텍스트 또는 범위를 제공합니다. 예를 들어 모듈에 정의된 프로바이더는 내보낼 필요 없이 모듈의 다른 멤버가 볼 수 있습니다. 프로바이더를 모듈 외부에 표시해야 하는 경우 먼저 호스트 모듈에서 내보낸 다음 소비 모듈로 가져옵니다.

익숙한 예를 살펴보겠습니다.

먼저 `UserService`를 제공하고 내보낼 `UsersModule`을 정의하겠습니다. `UsersModule`은 `UserService`의 호스트 모듈입니다.

```
'@nestjs/common'에서 { Module }을 가져오고,  
.users.service'에서 { UserService }를 가져옵니다;
```

```
모듈({  
    제공자: [UserService], 내보내  
    기: [UserService],  
})  
사용자 모듈 클래스 {} 내보내기
```

다음으로, `UsersModule`을 가져와서 `UsersModule`의 내보낸 프로바이더를 `AuthModule` 내에서 사용할 수 있게 만드는 `AuthModule`을 정의하겠습니다:

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다  
;  
'./users/users.module'에서 { UsersModule }을 가져옵니다;  
  
모듈({  
    임포트: [UsersModule], 공급자:  
        [AuthService], 내보내기:  
        [AuthService],  
})  
내보내기 클래스 AuthModule {}
```

이러한 구성을 사용하면 예를 들어 다음에서 호스팅되는 **AuthService**에 **UsersService**를 삽입할 수 있습니다.
AuthModule:

```
'@nestjs/common'에서 { Injectable }을 가져옵니다;  
'./users/users.service'에서 { UserService }를 가져옵니다;  
  
@Injectable()  
내보내기 클래스 AuthService {  
  constructor(private usersService: UserService) {}  
  /*  
    this.usersService를 사용하는 구현  
  */  
}
```

이를 정적 모듈 바인딩이라고 합니다. Nest가 모듈을 서로 연결하는 데 필요한 모든 정보는 호스트와 소비 모듈에 이미 선언되어 있습니다. 이 과정에서 어떤 일이 일어나는지 살펴봅시다. Nest는 **AuthModule** 내에서 **UserService**를 다음과 같이 사용할 수 있도록 합니다:

- . **UsersModule** 자체가 소비하는 다른 모듈을 일시적으로 가져오고 종속성을 일시적으로 해결하는 것을 포함하여 **UsersModule**을 인스턴스화합니다([사용자 정의 공급자](#) 참조).
- . **AuthModule**을 인스턴스화하고, **UsersModule**의 내보낸 프로바이더를 **AuthModule**의 컴포넌트에서 사용할 수 있도록 합니다(마치 **AuthModule**에서 선언한 것처럼).
- . **AuthService**에 **UserService** 인스턴스를 주입합니다.

동적 모듈 사용 사례

정적 모듈 바인딩을 사용하면 소비 모듈이 호스트 모듈의 공급자 구성 방식에 영향을 미칠 기회가 없습니다. 이것이 왜 중요할까요? 사용 사례에 따라 다르게 동작해야 하는 범용 모듈이 있는 경우를 생각해 보세요. 이는 많은 시스템에서 '플러그인'이라는 개념과 유사하며, 일반 기능을 소비자가 사용하기 전에 약간의 구성이 필요합니다.

Nest의 좋은 예로 구성 모듈을 들 수 있습니다. 많은 애플리케이션에서 구성 모듈을 사용하여 구성 세부 정보를 외부화하는 것이 유용합니다. 이렇게 하면 개발자를 위한 개발 데이터베이스, 스테이징/테스트 환경을 위한 스테이징 데이터베이스 등 다양한 배포에서 애플리케이션 설정을 동적으로 쉽게 변경할 수 있습니다. 구성 매개변수 관리를 구성 모듈에 위임하면 애플리케이션 소스 코드는 구성 매개변수와 독립적으로 유지됩니다.

문제는 구성 모듈 자체가 일반적('플러그인'과 유사)이기 때문에 이를 사용하는 모듈에 따라 사용자 정의해야 한다는 점입니다. 바로 이때 **동적 모듈**이 등장합니다. 동적 모듈 기능을 사용하면 구성 모듈을 동적으로 만들어 소비 모듈이 API를 사용하여 구성 모듈을 가져올 때 구성 모듈이 사용자 지정되는 방식을 제어할 수 있습니다.

즉, 동적 모듈은 지금까지 살펴본 정적 바인딩을 사용하는 것과 달리 한 모듈을 다른 모듈로 가져오고 가져온 모듈의 속성 및 동작을 사용자 정의할 수 있는 API를 제공합니다.

구성 모듈 예제

이 섹션에서는 [구성](#) 장에 있는 예제 코드의 기본 버전을 사용하겠습니다. 이 장이 끝날 때 완성된 버전은 [여기에서](#) 작업 [예제로](#) 사용할 수 있습니다.

우리의 요구 사항은 [구성 모듈이 옵션](#) 객체를 수락하여 사용자 정의하도록 하는 것입니다. 지원하고자 하는 기능은 다음과 같습니다. 기본 샘플은 프로젝트 루트 폴더에 있는 `.env` 파일의 위치를 하드코딩합니다. 이를 구성할 수 있도록 하여 원하는 폴더에서 `.env` 파일을 관리할 수 있도록 하겠다고 가정해 보겠습니다. 예를 들어, 다양한 `.env` 파일을 프로젝트 루트 아래 `config`라는 폴더(즉, `src`의 형제 폴더)에 저장하고 싶다고 가정해 봅시다. 다른 프로젝트에서 [컨피그모듈을](#) 사용할 때 다른 폴더를 선택할 수 있기를 원할 것입니다.

동적 모듈을 사용하면 가져오는 모듈에 매개 변수를 전달하여 동작을 변경할 수 있습니다. 어떻게 작동하는지 살펴봅시다. 사용하는 모듈의 관점에서 어떻게 보일지에 대한 최종 목표에서 시작하여 거꾸로 작업하는 것이 도움이 됩니다. 먼저, 정적으로 [컨피그 모듈을](#) 임포트하는 예제(즉, 임포트된 모듈의 동작에 영향을 주지 않는 접근 방식)를 빠르게 살펴봅시다. [모듈\(\)](#) 데코레이터의 [임포트](#) 배열을 주의 깊게 살펴보세요:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./app.controller'에서 { AppController }를 임포트하고
, './app.service'에서 { AppService }를 임포트합니다;
'./config/config.module'에서 { ConfigModule }을 가져옵니다;

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

구성 객체를 전달하는 [동적 모듈](#) 임포트가 어떤 모습일지 생각해 봅시다. 이 두 예제에서 [임포트](#) 배열의 차이를 비교해 보세요:

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./app.controller'에서 { AppController }를 임포트하고  
, './app.service'에서 { AppService }를 임포트합니다;  
'./config/config.module'에서 { ConfigModule }을 가져옵니다;
```

모듈({

```
    임포트합니다: [ConfigModule.register({ 풀더: './config' })], 컨트
```

```
    롤러: [AppController],
```

```
    공급자: [앱서비스],
```

```
)
```

```
내보내기 클래스 AppModule {}
```

위의 동적 예시에서 어떤 일이 일어나는지 살펴봅시다. 움직이는 부분은 무엇인가요?

. ConfigModule은 일반 클래스이므로 정적 메서드인

`register()`. 이 함수는 정적이라는 것을 알고 있습니다.

- 인스턴스를 생성합니다. 참고: 곧 생성할 이 메서드는 임의의 이름을 가질 수 있지만, 관례에 따라 `forRoot()` 또는 `register()` 중 하나로 호출해야 합니다.
- . `register()` 메서드는 우리가 정의한 것이므로 원하는 입력 인수를 받을 수 있습니다. 이 경우 적절한 속성을 가진 간단한 옵션 객체를 받아들이는 것이 일반적인 경우입니다.
 - . 지금까지 살펴본 익숙한 `import` 목록에 반환값이 모듈 목록이 포함되어 있으므로 `register()` 메서드가 모듈과 같은 것을 반환해야 한다는 것을 유추할 수 있습니다.

실제로 `register()` 메서드가 반환하는 것은 `DynamicModule`입니다. 동적 모듈은 런타임에 생성되는 모듈로, 정적 모듈과 동일한 프로퍼티에 `module`이라는 프로퍼티를 하나 더 추가한 것에 불과합니다. 데코레이터에 전달된 모듈 옵션을 주의 깊게 살펴보면서 샘플 정적 모듈 선언을 빠르게 검토해 보겠습니다:

```
모듈({  
    임포트: [DogsModule], 컨트롤러:  
        [CatsController], 제공자:  
            [CatsService], 내보내기:  
            [CatsService]  
})
```

동적 모듈은 정확히 동일한 인터페이스를 가진 객체와 `module`이라는 추가 프로퍼티 하나를 반환해야 합니다.

`module` 속성은 모듈의 이름 역할을 하며, 아래 예시와 같이 모듈의 클래스 이름과 동일해야 합니다.

정보 힌트 동적 모듈의 경우 모듈 옵션 객체의 모든 속성은 다음을 제외하고 선택 사항입니다.

모듈입니다.

정적 `register()` 메서드는 어떨까요? 이제 이 메서드의 임무가 `DynamicModule` 인터페이스를 가진 객체를 반환하는 것임을 알 수 있습니다. 이 메서드를 호출하면 정적인 경우 모듈 클래스 이름을 나열하는 방식과 유사하게 `임포트` 목록에 모듈을 효과적으로 제공하게 됩니다. 즉, 동적 모듈 API는 단순히 모듈을 반환하지만 `@Module` 데코레이터에서 프로퍼티를 수정하는 대신 프로그래밍 방식으로 프로퍼티를 지정합니다.

그림을 완성하는 데 도움이 되는 몇 가지 세부 사항이 아직 남아 있습니다:

- . 이제 `@Module()` 데코레이터의 `imports` 프로퍼티는 모듈 클래스 이름(예: `[UsersModule]`)뿐만 아니라 동적 모듈을 반환하는 함수(예: `[imports: [ConfigModule.register(...)]]`).
- . 동적 모듈은 자체적으로 다른 모듈을 임포트할 수 있습니다. 이 예제에서는 그렇게 하지 않겠지만, 동적

모듈이 다른 모듈의 프로바이더에 의존하는 경우 선택적 `import` 속성을 사용하여 해당 프로바이더를 가져올 수 있습니다. 다시 말하지만, 이는 `@Module()` 데코레이터를 사용하여 정적 모듈에 대한 메타데이터를 선언하는 방식과 정확히 유사합니다.

이러한 이해를 바탕으로 이제 동적 `구성 모듈` 선언이 어떤 모습이어야 하는지 살펴볼 수 있습니다. 한번 살펴봅시다.

```
'@nestjs/common'에서 { DynamicModule, Module }을 가져오고,
'./config.service'에서 { ConfigService }를 가져옵니다;
```

모듈({})

```
내보내기 클래스 ConfigModule {
    정적 register(): DynamicModule {
        {
            모듈을 사용합니다: 컨피그모듈, 공
            급자: [컨피그서비스], 내보내기:
            [ConfigService],
        };
    }
}
```

이제 조각들이 어떻게 서로 연결되는지 명확해졌을 것입니다. `ConfigModule.register(...)`을 호출하면 지금 까지 `@Module()` 데코레이터를 통해 메타데이터로 제공한 것과 본질적으로 동일한 프로퍼티를 가진 `DynamicModule` 객체가 반환됩니다.

정보 힌트 `@nestjs/common`에서 `DynamicModule`을 가져옵니다.

하지만 동적 모듈은 아직 우리가 원하는 대로 구성할 수 있는 기능을 도입하지 않았기 때문에 그다지 흥미롭지 않습니다. 이 부분은 다음에 다루겠습니다.

모듈 구성

위에서 추측한 것처럼 정적 `register()` 메서드에 옵션 객체를 전달하는 것이 `ConfigModule`의 동작을 커스터마이징하는 가장 확실한 해결책입니다. 소비 모듈의 `import` 프로퍼티를 다시 한 번 살펴봅시다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./app.controller'에서 { AppController }를 임포트하고
, './app.service'에서 { AppService }를 임포트합니다;
'./config/config.module'에서 { ConfigModule }을 가져옵니다;
```

모듈({

```
    임포트합니다: [ConfigModule.register({ 풀더: './config' })], 컨트
    롤러: [AppController],
    공급자: [AppService],
})
내보내기 클래스 AppModule {}
```

이렇게 하면 옵션 객체를 동적 모듈에 전달하는 작업이 잘 처리됩니다. 그러면 이 옵션 객체를 컨피그모듈에서 어떻게 사용할까요? 잠시 생각해 봅시다. 우리는 기본적으로 컨피그모듈이 다른 공급자가 사용할 수 있도록 인젝터를 서비스인 컨피그서비스를 제공하고 내보내기 위한 호스트라는 것을 알고 있습니다. 실제로 동작을 사용자 정의하기 위해 옵션 객체를 읽어야 하는 것은 컨피그서비스입니다. 일단 등록() 메서드에서 어떻게든 옵션을 컨피그서비스로 가져오는 방법을 알고 있다고 가정해 봅시다. 이 가정에 따라 서비스를 몇 가지 변경하여 옵션 객체의 속성을 기반으로 동작을 사용자 지정할 수 있습니다. (참고: 현재

전달 방법을 실제로 결정하지 않았기 때문에 옵션을 하드코딩할 것입니다. 이 문제는 곧 수정하겠습니다).

```
'@nestjs/common'에서 { Injectable }을 임포트하고,
'dotenv'에서 *를 dotenv로 임포트합니다;
'fs'에서 *를 fs로 가져오기; '경로'
에서 *를 경로로 가져오기;
'./interfaces'에서 { EnvConfig }를 임포트합니다;

@Injectable()
내보내기 클래스 ConfigService {
  비공개 읽기 전용 envConfig: EnvConfig;

  constructor() {
    const options = { 폴더: './config' };

    const filePath = `${process.env.NODE_ENV || 'development'}.env`;
    const envFile = path.resolve(__dirname, '../..', options.folder,
파일 경로);
    this.envConfig = dotenv.parse(fs.readFileSync(envFile));
  }

  get(키: 문자열): 문자열 { return
    this.envConfig[키];
  }
}
```

이제 컨피그서비스는 옵션에서 지정한 폴더에서 .env 파일을 찾는 방법을 알고 있습니다. 남은 작업은 등록

() 단계의 옵션 객체를 어떻게든 우리의

ConfigService. 물론 이를 위해 의존성 주입을 사용할 것입니다. 이것이 핵심이므로 반드시 이해해야 합니다.

우리의 컨피그모듈은 컨피그서비스를 제공하고 있습니다. ConfigService는 런타임에만 제공되는 옵션 객체에 따라 달라집니다. 따라서 런타임에 먼저 옵션 객체를 Nest IoC 컨테이너에 바인딩한 다음 Nest가 이를 컨피그서비스에 주입하도록 해야 합니다. 사용자 지정 공급자 챕터에서 공급자는 서비스뿐만 아니라 모든 값을 포함할 수 있으므로 종속성 주입을 사용하여 간단한 옵션 객체를 처리해도 괜찮다는 것을 기억하세요.

먼저 옵션 객체를 IoC 컨테이너에 바인딩하는 방법을 살펴봅시다. 정적 register() 메서드에서 이 작업을 수행합니다. 모듈을 동적으로 구성하고 있으며 모듈의 속성 중 하나는 프로바이더 목록이라는 점을 기억하세요. 따라서 우리가 해야 할 일은 옵션 객체를 프로바이더로 정의하는 것입니다. 이렇게 하면 다음 단계에서 활용하게 될 컨피그서비스에 주입할 수 있게 됩니다. 아래 코드에서 공급자 배열에 주목하세요:

'@nestjs/common'에서 { DynamicModule, Module }을 가져오고,
'./config.service'에서 { ConfigService }를 가져옵니다;

모듈({})

내보내기 클래스 컨피그모듈 {

정적 register(옵션: Record<string, any>): DynamicModule {

```

반환 {

    모듈을 사용합니다: 컨피그모듈

    , 공급자: [
        {
            제공: 'CONFIG_OPTIONS',
            useValue: 옵션,
        },
        ConfigService,
    ],
    내보내기: [구성 서비스],
};

}
}
}

```

이제 컨피그서비스에 'CONFIG_OPTIONS' 프로바이더를 주입하여 프로세스를 완료할 수 있습니다. 클래스가 아닌 토큰을 사용하여 프로바이더를 정의할 때는 [여기에 설명된 대로 @Inject\(\)](#) 데코레이터를 사용해야 한다는 것을 기억하세요.

```

'dotenv'에서 *를 *로 가져옵니다. 'fs'
에서 *를 *로 가져옵니다;
'경로'에서 *를 경로로 가져옵니다;
'@nestjs/common'에서 { Injectable, Inject }를 임포트하고,
'./interfaces'에서 { EnvConfig }를 임포트합니다;

@Injectable()
내보내기 클래스 ConfigService {
    비공개 읽기 전용 envConfig: EnvConfig;

    생성자(@Inject('CONFIG_OPTIONS') 비공개 옵션: Record<string, any>) {
        const filePath = `${process.env.NODE_ENV || 'development'}.env`;
        const envFile = path.resolve(__dirname, '../..', options.folder,
파일 경로);
        this.envConfig = dotenv.parse(fs.readFileSync(envFile));
    }

    get(키: 문자열): 문자열 {
        return
            this.envConfig[키];
    }
}

```

마지막으로 한 가지 참고 사항: 간단하게 하기 위해 위에서 문자열 기반 인젝션 토큰('CONFIG_OPTIONS')을

사용했지만, 가장 좋은 방법은 별도의 파일에 상수(또는 **심볼**)로 정의하고 해당 파일을 임포트하는 것입니다.

예를 들어

```
export const CONFIG_OPTIONS = 'CONFIG_OPTIONS';
```

예

이 장의 전체 코드 예제는 [여기에서](#) 확인할 수 있습니다. 커뮤니티 가

이드라인

`forRoot`, `register`, `forFeature`와 같은 메서드에 사용되는 것을 보셨을 것입니다.

패키지를 사용하면서 이 모든 메서드의 차이점이 무엇인지 궁금할 수 있습니다. 이에 대한 명확한 규칙은 없지만

`@nestjs/` 패키지는 다음 가이드라인을 따르려고 노력합니다:

모듈을 만들 때

- 등록하는 경우, 호출 모듈에서만 사용할 수 있도록 특정 구성으로 동적 모듈을 구성해야 합니다. 예를 들어 Nest의 `HttpModule.register({{ '{' }} baseUrl: 'someUrl' {{ '}' }})`. 다른 모듈에서 사용하는 경우 `HttpModule.register({{ '{' }} baseUrl: '어딘가 다른' {{ '}' }})`를 사용하면 다른 구성을 갖게 됩니다. 원하는 만큼 많은 모듈에 대해 이 작업을 수행할 수 있습니다.
- `forRoot`를 사용하면 동적 모듈을 한 번 구성하고 여러 곳에서 해당 구성을 재사용할 수 있습니다(추상화되어 있기 때문에 자신도 모르게 재사용할 수도 있지만). 그렇기 때문에 `GraphQLModule.forRoot()`가 하나, `TypeOrmModule.forRoot()`가 하나 등입니다.
- `forFeature`의 구성을 사용해야 하지만 호출 모듈의 요구 사항(예: 이 모듈이 액세스할 수 있는 리포지토리 또는 로거가 사용해야 하는 컨텍스트)에 따라 일부 구성을 수정해야 하는 경우입니다.

이 모든 것에는 일반적으로 비동기 대응 함수인 `registerAsync`, `forRootAsync`, `forFeatureAsync`가 있으며, 이는 같은 의미이지만 구성에도 Nest의 의존성 주입을 사용합니다.

구성 가능한 모듈 빌더

특히 초보자에게는 비동기 메서드(`registerAsync`, `forRootAsync` 등)를 노출하는 고도로 구성 가능한 동적 모듈을 수동으로 생성하는 것이 매우 복잡하므로 Nest는 이 과정을 용이하게 하고 단 몇 줄의 코드만으로 모듈 "청사진"을 구성할 수 있는 `ConfigurableModuleBuilder` 클래스를 노출합니다.

예를 들어, 위에서 사용한 예제(`ConfigModule`)를 컨피규러블 모듈 빌더를 사용하도록 변환해 보겠습니다. 시작하기 전에 `ConfigModule`이 받는 옵션을 나타내는 전용 인터페이스를 만들어 보겠습니다.

```
내보내기 인터페이스 ConfigModuleOptions { 풀더  
  : 문자열;  
}
```

이렇게 하면 기존 `config.module.ts` 파일과 함께 새로운 전용 파일을 생성하고 이름을 `config.module-definition.ts`로 지정합니다. 이 파일에서 `ConfigModuleBuilder`를 활용하여 `컨피그모듈` 정의를 작성해 보겠습니다.

```

@@파일명(config.module-definition)
'@nestjs/common'에서 { ConfigurableModuleBuilder }를 임포트하고,
'./interfaces/config-module-options.interface'에서 {
ConfigModuleOptions }를 임포트합니다;

export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new ConfigurableModuleBuilder<ConfigModuleOptions>().build();
@@switch
'@nestjs/common'에서 { ConfigurableModuleBuilder }를 가져옵니다;

export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  새로운 ConfigurableModuleBuilder().build();

```

이제 config.module.ts 파일을 열고 자동 생성된 ConfigurableModuleClass를 활용하도록 구현을 수정해 보겠습니다:

```

'@nestjs/common'에서 { Module }을 가져옵니다;
'./config.service'에서 { ConfigService }를 가져옵니다;
'./config.module-definition'에서 { ConfigurableModuleClass }를 임포트합니다;

모듈({
  제공자: [구성 서비스], 내보내기:
  [ConfigService],
})
내보내기 클래스 ConfigModule extends ConfigurableModuleClass {}

```

ConfigurableModuleClass를 확장한다는 것은 이제 컨피그모듈이 (이전 사용자 정의 구현에서와 같이) register 메서드뿐만 아니라 비동기 팩토리를 제공하여 소비자가 해당 모듈을 비동기적으로 구성할 수 있도록 하는 registerAsync 메서드도 제공한다는 의미입니다:

```
모듈({ import: [
    ConfigModule.register({ 풀더: './config' }),
    // 또는 그 반대로:
    // ConfigModule.registerAsync({
    //useFactory : () => {
        //return {
    //풀더 : './config',
    //},
    //},
    //inject : [...추가 종속성...]
    //}),
],
})
```

내보내기 [클래스](#) AppModule {}

마지막으로, 지금까지 사용한 '`CONFIG_OPTIONS`' 대신 생성된 모듈 옵션의 프로바이더를 주입하도록 `ConfigService` 클래스를 업데이트하겠습니다.

```
@Injectable()
export class ConfigService {
  constructor(@Inject(MODULE_OPTIONS_TOKEN) private options:
ConfigModuleOptions) { ... }
}
```

사용자 지정 메서드 키

`ConfigurableModuleClass`는 기본적으로 `register`와 그에 대응하는 `registerAsync` 메서드를 제공합니다. 다른 메서드 이름을 사용하려면 다음과 같이 `ConfigurableModuleBuilder#setClassMethodName` 메서드를 사용하세요:

```
@@파일명(config.module-definition)
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  새로운 ConfigurableModuleBuilder<ConfigModuleOptions>
() .setClassMethodName('forRoot').build();
@@switch
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  새로운 ConfigurableModuleBuilder().setClassMethodName('forRoot').build();
```

이 구성은 `ConfigurableModuleBuilder`가 `forRoot`를 노출하는 클래스를 생성하도록 지시합니다.

를 대신 사용할 수 있습니다. 예시:

```
모듈({ import: [
  ConfigModule.forRoot({ 풀더: './config' }), // <-- "register" 대신
  "forRoot"를 사용했음에 유의하세요.
  // 또는 그 반대로:
  // ConfigModule.forRootAsync({
  //useFactory : () => {
    //return {
    //풀더 : './config',
    //}
    //},
    //inject : [...추가 종속성...]
    //}),
  ],
})
```

내보내기 클래스 AppModule {}

사용자 지정 옵션 팩토리 클래스

등록동기 메서드(또는 구성에 따라 `forRootAsync` 또는 다른 이름)를 사용하면 소비자가 모듈 구성의 확인하는 공급자 정의를 전달할 수 있으므로 라이브러리 소비자는 잠재적으로 구성 객체를 구성하는 데 사용할 클래스를 제공할 수 있습니다.

```
모듈({ import: [
  ConfigModule.registerAsync({
    사용 클래스: 구성모듈옵션팩토리,
  }),
],
})

내보내기 클래스 AppModule {}
```

이 클래스는 기본적으로 모듈 구성 객체를 반환하는 `create()` 메서드를 제공해야 합니다. 그러나 라이브러리가 다른 명명 규칙을 따르는 경우 해당 동작을 변경하고

`ConfigurableModuleBuilder#setFactoryMethodName` 메서드를 사용하여 다른 메서드(예: `createConfigOptions()`)를 기대하도록 `ConfigurableModuleBuilder`에 지시할 수 있습니다:

```
@@파일명(config.module-definition)
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  새로운 ConfigurableModuleBuilder<ConfigModuleOptions>
  () .setFactoryMethodName('createConfigOptions').build();
@@switch
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } =
  new
  ConfigurableModuleBuilder().setFactoryMethodName('createConfigOptions').bu
  ild();
```

이제 `ConfigModuleOptionsFactory` 클래스는 `create` 대신 `createConfigOptions` 메서드를 노출해야 합니다:

```
모듈({ import: [
  ConfigModule.registerAsync({
    useClass: ConfigModuleOptionsFactory, // <-- 이 클래스는
    "createConfigOptions" 메서드를 제공해야 합니다.
  }),
],
})

내보내기 클래스 AppModule {}
```

추가 옵션

모듈의 동작 방식을 결정하는 추가 옵션(이러한 옵션의 좋은 예는 `isGlobal` 플래그 또는 그냥 전역)을 동시에 사용해야 하는 예지 케이스가 있을 수 있습니다,

는 해당 모듈 내에 등록된 서비스/프로바이더와 관련이 없으므로(예를 들어, ConfigService는 호스트 모듈이 전역 모듈로 등록되었는지 여부를 알 필요가 없음) **MODULE_OPTIONS_TOKEN** 공급자에 포함되지 않아야 합니다.

이러한 경우 **ConfigurableModuleBuilder#setExtras** 메서드를 사용할 수 있습니다. 다음 예제를 참조하세요 :

```
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN } = new
ConfigurableModuleBuilder<ConfigModuleOptions>()
.setExtras(
{
  isGlobal: true,
},
(정의, 추가) => ({
  ...정의,
  글로벌: extras.isGlobal,
}),
)
.build();
```

위의 예에서 **setExtras** 메서드에 전달된 첫 번째 인수는 "추가" 속성에 대한 기본값이 포함된 객체입니다. 두 번째 인수는 자동 생성된 모듈 정의(공급자, 내보내기 등 포함)와 추가 속성(소비자가 지정하거나 기본값)을 나타내는 **추가** 객체를 취하는 함수입니다. 이 함수의 반환 값은 수정된 모듈 정의입니다. 이 특정 예제에서는 **extras.isGlobal** 속성을 가져와 모듈 정의의 **전역** 속성에 할당합니다(모듈이 전역인지 아닌지를 결정합니다. 자세한 내용은 [여기를](#) 참조하세요).

이제 이 모듈을 사용할 때 다음과 같이 추가적으로 **isGlobal** 플래그를 전달할 수 있습니다:

```
모듈({ import: [
  ConfigModule.register({
    isGlobal: true, 풀더:
    './config',
  }),
],
})
내보내기 클래스 AppModule {}
```

그러나 **isGlobal**은 "추가" 속성으로 선언되었기 때문에

MODULE_OPTIONS_TOKEN 공급자:

```
@Injectable()
export class ConfigService { 생성자
  (@Inject(MODULE_OPTIONS_TOKEN) 비공개 옵션:
ConfigModuleOptions) {
```

```
// "옵션" 객체에는 "isGlobal" 속성이 없습니다.
// ...
}
```

자동 생성 메서드 확장

자동 생성된 정적 메서드(`register`, `registerAsync` 등)는 필요한 경우 다음과 같이 확장할 수 있습니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./config.service'에서 { ConfigService }를 가져옵니다;
'./config.module-definition'에서 { ConfigurableModuleClass,
ASYNC_OPTIONS_TYPE, OPTIONS_TYPE }을 임포트합니다;

모듈({
  제공자: [구성 서비스], 내보내기:
  [ConfigService],
})

내보내기 클래스 ConfigModule extends ConfigurableModuleClass { 정적
  register(options: typeof OPTIONS_TYPE): DynamicModule {
    반환 {
      // 여기에 사용자 정의 로직
      ...super.register(options),
    };
  }

  정적 registerAsync(옵션: typeof ASYNC_OPTIONS_TYPE): DynamicModule
  {
    반환 {
      // 여기에 사용자 정의 로직
      ...super.registerAsync(options),
    };
  }
}
```

모듈 정의 파일에서 내보내야 하는 `OPTIONS_TYPE` 및 `ASYNC_OPTIONS_TYPE` 유형 사용에 유의하세요:

```
export const { ConfigurableModuleClass, MODULE_OPTIONS_TOKEN,
OPTIONS_TYPE, ASYNC_OPTIONS_TYPE } = new
ConfigurableModuleBuilder<ConfigModuleOptions>().build();
```

사용자 지정 공급자

이전 챕터에서는 의존성 주입(DI)의 다양한 측면과 Nest에서 어떻게 사용되는지에 대해 살펴보았습니다. 그 중 한 가지 예로 인스턴스(주로 서비스 프로바이더)를 클래스에 주입하는 데 사용되는 [생성자 기반](#) 의존성 주입을 들 수 있습니다. 의존성 주입이 Nest 코어에 기본적으로 내장되어 있다는 사실에 놀라지 않으실 것입니다. 지금까지는 한 가지 주요 패턴만 살펴봤습니다. 애플리케이션이 더 복잡해지면 DI 시스템의 모든 기능을 활용해야 할 수도 있으므로 좀 더 자세히 살펴보겠습니다.

DI 기본 사항

종속성 주입은 종속성 인스턴스화를 자체 코드에서 필수적으로 수행하는 대신 IoC 컨테이너(이 경우 NestJS 런타임 시스템)에 위임하는 [제어의 역전\(IoC\)](#) 기법입니다. [프로바이더 챕터의](#) 이 예제에서 어떤 일이 일어나고 있는지 살펴봅시다.

먼저 프로바이더를 정의합니다. `Injectable()` 데코레이터는 `CatsService` 클래스를 프로바이더로 표시합니다

```
@@파일명(cats.service)
'@nestjs/common'에서 { Injectable }을 임포트하고,
'./interfaces/cat.interface'에서 { Cat }을 임포트합니다;
다;

@Injectable()
내보내기 클래스 CatsService {
    비공개 읽기 전용 고양이: Cat[] = [];

    findAll(): Cat[] {
        return this.cats;
    }
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트합니다;

@Injectable()
내보내기 클래스 CatsService {
    constructor() {
        this.cats = [];
    }

    findAll() {
        this.cats를 반환합니다;
    }
}
```

그런 다음 Nest가 컨트롤러 클래스에 프로바이더를 주입하도록 요청합니다:

```
@@파일명(cats.controller)
'@nestjs/common'에서 { Controller, Get }를 가져오고,
'./cats.service'에서 { CatsService }를 가져옵니다;
```

```
'./interfaces/cat.interface'에서 { Cat }을 임포트합니다;

@Controller('cats')
내보내기 클래스 CatsController {
    constructor(private catsService: CatsService) {}

    @Get()
    비동기 findAll(): Promise<Cat[]> {
        return this.catsService.findAll();
    }
}

@@switch
'@nestjs/common'에서 { Controller, Get, Bind, Dependencies }를 임포트하고,
'./cats.service'에서 { CatsService }를 임포트합니다;
```

컨트롤러('cats') @의존성

```
(CatsService) 내보내기 클래스
CatsController {
    constructor(catsService) {
        this.catsService = catsService;
    }

    @Get()
    async findAll() {
        this.catsService.findAll()을 반환합니다;
    }
}
```

마지막으로 Nest IoC 컨테이너에 공급자를 등록합니다:

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./cats/cats.controller'에서 { CatsController }를 임포트하고
, './cats/cats.service'에서 { CatsService }를 임포트합니다;
```

모듈({

컨트롤러: [CatsController], 제공자:

[CatsService],

})

내보내기 클래스 AppModule {}

이 작업을 수행하기 위해 이면에서는 정확히 어떤 일이 일어나고 있을까요? 이 과정에는 세 가지 핵심 단계가 있습니다:

- . `cats.service.ts`에서 `@Injectable()` 데코레이터는 `CatsService` 클래스를 Nest IoC 컨테이너에 서 관리할 수 있는 클래스로 선언합니다.
- . `cats.controller.ts`에서 `CatsController`는 생성자 주입을 통해 `CatsService` 토큰에 대한 종 속성을 선언합니다:

```
생성자(private catsService: CatsService)
```

. `app.module.ts`에서 토큰 `CatsService`를 `cats.service.ts` 파일의 `CatsService` 클래스와 연결합니다. 아래에서 이 연결(등록이라고도 함)이 정확히 어떻게 발생하는지 살펴보겠습니다.

Nest IoC 컨테이너가 `CatsController`를 인스턴스화할 때, 먼저 모든 종속성을 찾습니다. `CatsService` 종속성을 찾으면 등록 단계(위 #3)에 따라 `CatsService` 토큰에 대한 조회를 수행하여 `CatsService` 클래스를 반환합니다. 기본 동작인 싱글톤 범위로 가정하면 Nest는 `CatsService` 인스턴스를 생성하여 캐시한 후 반환하거나, 이미 캐시된 인스턴스가 있는 경우 기존 인스턴스를 반환합니다.

*이 설명은 요점을 설명하기 위해 약간 단순화했습니다. 여기서 간과한 한 가지 중요한 부분은 종속성에 대한 코드 분석 프로세스가 매우 정교하며 애플리케이션 부트스트랩 중에 발생한다는 점입니다. 한 가지 중요한 특징은 종속성 분석(또는 "종속성 그래프 생성")이 전이적이라는 점입니다. 위의 예시에서 `CatsService` 자체에 종속성이 있었다면 이 종속성도 해결되었을 것입니다. 종속성 그래프는 종속성이 올바른 순서로, 즉 본질적으로 "상향식"으로 해결되도록 보장합니다. 이 메커니즘은 개발자가 복잡한 종속성 그래프를 관리할 필요를 덜어줍니다.

표준 공급자

모듈() 데코레이터를 자세히 살펴봅시다. `app.module`에서 선언합니다:

```
모듈({
  컨트롤러: [CatsController], 제공자:
  [CatsService],
})
```

프로바이더 프로퍼티는 프로바이더 배열을 받습니다. 지금까지는 클래스 이름 목록을 통해 이러한 공급자를 제공했습니다. 사실, 구문 제공자: `[CatsService]`는 보다 완전한 구문을 줄여서 표현한 것입니다:

```
제공자: [
{
  제공: CatsService,
  useClass: CatsService,
},
];
```

이제 이 명시적인 구조를 확인했으니 등록 프로세스를 이해할 수 있습니다. 여기서는 토큰 `CatsService`를 `CatsService` 클래스와 명확하게 연결하고 있습니다. 약식 표기는 토큰이 같은 이름의 클래스 인스턴스를 요청하는 데 사용되는 가장 일반적인 사용 사례를 단순화하기 위한 편의상 표기일 뿐입니다.

사용자 지정 공급자

표준 제공업체가 제공하는 요구 사항을 초과하는 요구 사항이 있으면 어떻게 되나요? 다음은 몇 가지 예입니다:

- Nest가 클래스를 인스턴스화(또는 캐시된 인스턴스를 반환)하는 대신 사용자 정의 인스턴스를 생성하려고 합니다.
- 두 번째 종속성에서 기존 클래스를 재사용하려는 경우 • 테스트를 위해 모의 버전으로 클래스를 재정의하려는 경우

Nest를 사용하면 이러한 경우를 처리할 사용자 정의 공급자를 정의할 수 있습니다. 사용자 정의 공급자를 정의하는 몇 가지 방법을 제공합니다. 몇 가지 방법을 살펴보겠습니다.

정보 힌트 종속성 해결에 문제가 있는 경우 `NEST_DEBUG`를 설정할 수 있습니다.

환경 변수를 설정하고 시작 중에 추가 종속성 해결 로그를 가져옵니다.

값 공급자: `useValue`

사용값 구문은 상수 값을 주입하거나, 외부 라이브러리를 Nest 컨테이너에 넣거나, 실제 구현을 모의 객체로 대체할 때 유용합니다. Nest가 테스트 목적으로 모의 `CatsService`를 사용하도록 강제하고 싶다고 가정해 보겠습니다.

```
'./cats.service'에서 { CatsService } 임포트;
```

```
const mockCatsService = {
  /* 모의 구현
  ...
}
};
```

```
모듈({
  임포트: [CatsModule], 제공자:
  [
    {
      제공: CatsService, useClass:
      mockCatsService,
    },
  ],
})
```

```
내보내기 클래스 AppModule {}
```

이 예제에서 `CatsService` 토큰은 `mockCatsService` 모의 객체로 리졸브됩니다. 사용값에는 값(이 경우 대

체하는 `CatsService` 클래스와 동일한 인터페이스를 가진 리터럴 객체)이 필요합니다. TypeScript의 구조적 타이핑으로 인해 리터럴 객체 또는 `new`로 인스턴스화된 클래스 인스턴스를 포함하여 호환되는 인터페이스를 가진 모든 객체를 사용할 수 있습니다.

클래스 기반이 아닌 공급자 토큰

지금까지는 클래스 이름을 프로바이더 토큰(`프로바이더` 배열에 나열된 프로바이더의 프로퍼티 값)으로 사용했습니다. 이는 생성자 [기반 주입에](#) 사용되는 표준 패턴과 일치하며, 토큰도 클래스 이름입니다. (토큰에 대한 자세한 내용은 [DI 기초](#)를 다시 참조하세요.

이 개념은 완전히 명확하지 않습니다). 때로는 문자열이나 기호를 DI 토큰으로 사용할 수 있는 유연성이 필요할 수 있습니다. 예를 들어

```
'./connection'에서 { connection } 임포트;

@Module({
  제공자: [
    {
      제공: '연결', 사용값: 연결,
    },
  ],
})
내보내기 클래스 AppModule {}
```

이 예에서는 문자열 값 토큰('CONNECTION')을 기준 **연결**에 연결합니다.

객체를 가져올 수 있습니다.

경고 토큰 값으로 문자열을 사용하는 것 외에도 JavaScript **기호** 또는 TypeScript **열거형**을 사용할 수도 있습니다.

앞서 표준 **생성자 기반 주입 패턴**을 사용하여 프로바이더를 주입하는 방법을 살펴봤습니다. 이 패턴을 사용하려면 의존성을 클래스 이름으로 선언해야 합니다. 'CONNECTION' 사용자 정의 프로바이더는 문자열 값 토큰을 사용합니다. 이러한 프로바이더를 주입하는 방법을 살펴봅시다. 이를 위해 **@Inject()** 데코레이터를 사용합니다. 이 데코레이터는 토큰이라는 단일 인수를 받습니다.

```
@@파일명()
@Injectable()
export class CatsRepository {
  constructor(@Inject('CONNECTION') connection: Connection) {}
}

@@스위치
@Injectable()
@Dependencies('CONNECTION')
export class CatsRepository {
  constructor(connection) {}
}
```

정보 힌트 **@Inject()** 데코레이터는 **@nestjs/common** 패키지에서 가져옵니다.

위의 예시에서는 예시용으로 '**CONNECTION**' 문자열을 직접 사용했지만, 깔끔한 코드 정리를 위해서는 **constants.ts**와 같은 별도의 파일에 토큰을 정의하는 것이 가장 좋습니다. 토큰을 자체 파일에 정의하고 필요한 경우 가져오는 심볼이나 열거형과 마찬가지로 취급하세요.

클래스 공급자: **사용 클래스**

useClass 구문을 사용하면 토큰이 확인해야 하는 클래스를 동적으로 결정할 수 있습니다. 예를 들어 추상(또는 기본) **ConfigService** 클래스가 있다고 가정해 보겠습니다. 현재

환경에서는 Nest가 다른 구성 서비스 구현을 제공하길 원합니다. 다음 코드는 이러한 전략을 구현합니다.

```
const configServiceProvider = {
  provide: 컨피그서비스, 사용클래스:
    process.env.NODE_ENV === '개발'
      ? 개발 컨피그 서비스
      : ProductionConfigService,
};

모듈({
  공급자: [구성 서비스 공급자],
})

내보내기 클래스 AppModule {}
```

이 코드 샘플에서 몇 가지 세부 사항을 살펴봅시다. 먼저 리터럴 객체로 configServiceProvider를 정의한 다음 모듈 데코레이터의 프로바이더 프로퍼티에 전달한 것을 알 수 있습니다. 이것은 약간의 코드 구성일 뿐이지만 가능적으로는 이 장에서 지금까지 사용한 예제와 동일합니다.

또한 ConfigService 클래스 이름을 토큰으로 사용했습니다. ConfigService에 종속된 모든 클래스의 경우 Nest는 제공된 클래스(DevelopmentConfigService 또는 ProductionConfigService)의 인스턴스를 주입하여 다른 곳에서 선언되었을 수 있는 기본 구현(예: @Injectable() 데코레이터로 선언된 ConfigService)을 재정의합니다.

팩토리 공급자: useFactory

useFactory 구문을 사용하면 프로바이더를 동적으로 생성할 수 있습니다. 실제 프로바이더는 팩토리 함수에서 반환된 값으로 제공됩니다. 팩토리 함수는 필요에 따라 단순하거나 복잡할 수 있습니다. 단순한 팩토리는 다른 프로바이더에 의존하지 않을 수 있습니다. 더 복잡한 팩토리는 결과를 계산하는 데 필요한 다른 공급자를 자체적으로 주입할 수 있습니다. 후자의 경우 팩토리 공급자 구문에는 한 쌍의 관련 메커니즘이 있습니다:

- . 팩토리 함수는 (선택적) 인수를 받을 수 있습니다.
- . (선택 사항인) inject 프로퍼티는 인스턴스화 프로세스 중에 Nest가 확인하여 팩토리 함수에 인수로 전달할 공급자 배열을 허용합니다. 또한 이러한 공급자는 선택 사항으로 표시할 수 있습니다. 두 목록은 서로 연관되어 있어야 합니다: Nest는 인젝트 목록의 인스턴스를 동일한 순서로 팩토리 함수에 인수로 전달합니다. 아래 예시가 이를 보여줍니다.

```
@@파일명()
const connectionProvider = {
  provide: '연결',
  useFactory: (optionsProvider: 옵션 제공자, optionalProvider?: 문자열)
=> {
  const options = optionsProvider.get(); 반환
  새 데이터베이스 연결(옵션);
},
주입합니다: [OptionsProvider, { 토큰: '일부 옵션 제공자', 선택 사항:
```

```

true ],
  //           \_____/
  //           이 공급자
  //           는 필수입니다.
};

모듈({ providers: [
  연결 제공자, 옵션 제공자,
  // { provide: 'SomeOptionalProvider', useValue: 'anything' },
],
})

내보내기 클래스 AppModule {}

@@switch
const connectionProvider = {
  provide: '연결',
  useFactory: (optionsProvider, optionalProvider) => {
    const options = optionsProvider.get();
    새 데이터베이스 연결(옵션)을 반환합니다;
  },
  주입합니다: [OptionsProvider, { 토큰: '일부 옵션 제공자', 옵션: true }],
  //           \_____/
  //           이 공급자
  //           는 필수입니다.
};

모듈({ providers: [
  연결 제공자, 옵션 제공자,
  // { provide: 'SomeOptionalProvider', useValue: 'anything' },
],
})

내보내기 클래스 AppModule {}

```

별칭 공급자: 사용기준

사용 **Existing** 구문을 사용하면 기존 공급업체에 대한 별칭을 만들 수 있습니다. 이렇게 하면 동일한 공급자에 액세스할 수 있는 두 가지 방법이 생성됩니다. 아래 예제에서 (문자열 기반) 토큰인 '**AliasedLoggerService**'는 (클래스 기반) 토큰인 **LoggerService**의 별칭입니다. '**AliasedLoggerService**'에 대한 종속성과 **LoggerService**에 대한 종속성이 각각 하나씩 있다고 가정해 보겠습니다. 두 종속성 모두 **싱글톤** 범위로 지정되면 둘 다 동일한 인스턴스로 리졸브됩니다.

```
@Injectable()
로거 서비스 클래스 {
    /* 구현 세부 정보 */
}
```

```
const loggerAliasProvider = {
  provide: 'AliasedLoggerService',
  useExisting: LoggerService,
};

모듈({
  제공자: [로거 서비스, 로거 앤리어스 공급자],
})

내보내기 클래스 AppModule {}
```

서비스 기반이 아닌 공급자

공급자는 종종 서비스를 제공하지만, 그 용도에 국한되지 않습니다. 공급자는 모든 값을 제공할 수 있습니다. 예를 들어 공급자는 아래와 같이 현재 환경을 기반으로 구성 개체 배열을 제공할 수 있습니다:

```
const configFactory = {
  provide: 'CONFIG',
  useFactory: () => {
    반환 프로세스.env.NODE_ENV === '개발' ? devConfig : prodConfig;
  },
};

모듈({
  제공자: [configFactory],
})

내보내기 클래스 AppModule {}
```

사용자 지정 공급자 내보내기

다른 공급자와 마찬가지로 사용자 정의 공급자는 선언하는 모듈로 범위가 제한됩니다. 다른 모듈에서 볼 수 있도록 하려면 내보내야 합니다. 사용자 정의 공급자를 내보내려면 토큰이나 전체 공급자 객체를 사용할 수 있습니다.

다음 예는 토큰을 사용하여 내보내는 방법을 보여줍니다:

@@파일명()

```
const connectionFactory = {  
  provide: '연결',  
  useFactory: (optionsProvider: OptionsProvider) => {  
    const options = optionsProvider.get();  
    새 데이터베이스 연결(옵션)을 반환합니다;  
  },  
  주입합니다: [옵션 공급자],  
};
```

모듈({

공급자: [연결 팩토리],

```
수출: ['연결'],
})

내보내기 클래스 AppModule {}

@@switch
const connectionFactory = { provide:
  'CONNECTION', useFactory:
  (optionsProvider) => {
    const options = optionsProvider.get(); 반환
    새 데이터베이스 연결(옵션);
  },
  주입합니다: [옵션 공급자],
};

모듈({
  제공자: [connectionFactory], 내보
  내기: ['연결'],
}

내보내기 클래스 AppModule {}
```

또는 전체 공급자 개체를 사용하여 내보내세요:

@@파일명()

```
const connectionFactory = {  
  provide: '연결',  
  useFactory: (optionsProvider: OptionsProvider) => {  
    const options = optionsProvider.get();  
    새 데이터베이스 연결(옵션)을 반환합니다;  
  },  
  주입합니다: [옵션 공급자],  
};
```

모듈({

```
제공자: [연결 팩토리], 내보내기:  
[connectionFactory],  
})
```

```
내보내기 클래스 AppModule {}
```

@@switch

```
const connectionFactory = { provide:  
  'CONNECTION', useFactory:  
  (optionsProvider) => {  
    const options = optionsProvider.get(); 반환  
    새 데이터베이스 연결(옵션);  
  },  
  주입합니다: [옵션 공급자],  
};
```

모듈({

```
제공자: [연결 팩토리], 내보내기:  
[connectionFactory],  
})
```

```
내보내기 클래스 AppModule {}
```


순환 종속성

순환 종속성은 두 클래스가 서로 의존할 때 발생합니다. 예를 들어, 클래스 A는 클래스 B가 필요하고 클래스 B 역시 클래스 A가 필요합니다. Nest에서 순환 종속성은 모듈 간 또는 공급자 간에 발생할 수 있습니다.

순환 종속성은 가능한 한 피해야 하지만 항상 그렇게 할 수는 없습니다. 이러한 경우 Nest를 사용하면 두 가지 방법으로 프로바이더 간의 순환 종속성을 해결할 수 있습니다. 이 장에서는 한 가지 방법으로 정방향 참조를 사용하고, 다른 방법으로 ModuleRef 클래스를 사용하여 DI 컨테이너에서 공급자 인스턴스를 검색하는 방법을 설명합니다.

모듈 간의 순환 종속성 해결 방법도 설명합니다.

경고 "배럴 파일"/index.ts 파일을 사용하여 가져오기를 그룹화할 때 순환 종속성이 발생할 수도 있습니다.

모듈/프로바이더 클래스에 대해서는 배럴 파일을 생략해야 합니다. 예를 들어, 배럴 파일과 같은 디렉터리 내의 파일을 가져올 때는 배럴 파일을 사용해서는 안 됩니다. 즉, `cats/cats.controller`에서

`cats/cats.service` 파일을 가져오기 위해 `cats`를 가져와서는 안 됩니다. 자세한 내용은 [이 github 이슈를 참조하세요.](#)

앞으로 참조

정방향 참조를 사용하면 Nest가 `forwardRef()` 유틸리티 함수를 사용하여 아직 정의되지 않은 클래스를 참조할 수 있습니다. 예를 들어, CatsService와 CommonService가 서로 종속된 경우 관계의 양쪽에서 `@Inject()` 및 `forwardRef()` 유틸리티를 사용하여 순환 종속성을 해결할 수 있습니다.

그렇지 않으면 모든 필수 메타데이터를 사용할 수 없으므로 Nest에서 인스턴스화하지 않습니다. 다음은 예시입니다:

```
@@파일명(cats.service)
@Injectable()
내보내기 클래스 CatsService { 생
    성자(
        @Inject(forwardRef(() => CommonService))
        private commonService: CommonService,
    ) {}
}

@@스위치
@Injectable()
@Dependencies(forwardRef(() => CommonService))
export class CatsService {
    constructor(commonService) {
        this.commonService = commonService;
    }
}
```

정보 힌트 `forwardRef()` 함수는 `@nestjs/common` 패키지에서 가져온 것입니다.

이는 관계의 한 측면을 설명한 것입니다. 이제 `CommonService`에 대해서도 똑같이 해보겠습니다:

```

@@파일명(common.service)
@Injectable()
내보내기 클래스 CommonService {

생성자(
    @Inject(forwardRef(() => CatsService))
    private catsService: CatsService,
) {}

}

@@스위치
@Injectable()
@Dependencies(forwardRef(() => CatsService))
export class CommonService {
constructor(catsService) {
    this.catsService = catsService;
}
}

```

경고 경고 인스턴스화 순서는 불확실합니다. 코드가 어떤 생성자가 먼저 호출되는지에 따라 달라지지 않도록 하세요. [Scope.REQUEST](#)를 사용하는 공급자에 순환 종속성을 가지면 정의되지 않은 종속성이 발생할 수 있습니다. 자세한 정보는 [여기에서](#) 확인하세요.

ModuleRef 클래스 대체

[forwardRef\(\)](#)를 사용하는 대신 코드를 리팩터링하고 [ModuleRef](#) 클래스를 사용하여 순환 관계의 한쪽에서 공급자를 검색하는 방법을 사용할 수 있습니다. [여기에서](#) [ModuleRef](#) 유ти리티 클래스에 대해 자세히 알아보세요.

모듈 순방향 참조

모듈 간의 순환 종속성을 해결하려면 모듈 연결의 양쪽에서 동일한 [forwardRef\(\)](#) 유ти리티 함수를 사용하세요. 예를 들어

```

@@파일명(common.module)
@Module({
    임포트합니다: [forwardRef(() => CatsModule)],
})
내보내기 클래스 CommonModule {}

```

이것은 관계의 한 측면을 다룹니다. 이제 [CatsModule](#)에 대해서도 똑같이 해보겠습니다:

```
@@filename(cats.module)
@Module({
    임포트: [forwardRef(() => CommonModule)],
})
내보내기 클래스 CatsModule {}
```

모듈 참조

Nest는 내부 공급자 목록을 탐색하고 해당 주입 토큰을 조회 키로 사용하여 모든 공급자에 대한 참조를 얻을 수 있는 `ModuleRef` 클래스를 제공합니다. `ModuleRef` 클래스는 정적 및 범위가 지정된 프로바이더를 모두 동적으로 인스턴스화하는 방법도 제공합니다. `ModuleRef`는 일반적인 방법으로 클래스에 주입할 수 있습니다:

```
@@파일명(cats.service)
@Injectable()
내보내기 클래스 CatsService {
  constructor(private moduleRef: ModuleRef) {}
}

@@스위치
@Injectable()
@Dependencies(ModuleRef)
내보내기 클래스 CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }
}
```

정보 힌트 `ModuleRef` 클래스는 `@nestjs/core` 패키지에서 가져옵니다.

인스턴스 검색

`ModuleRef` 인스턴스(이하 모듈 레퍼런스라고 부릅니다)에는 `get()` 메서드가 있습니다. 이 메서드는 인젝션 토큰/클래스 이름을 사용하여 현재 모듈에 존재하는(인스턴스화된) 프로바이더, 컨트롤러 또는 인젝터블(예: 가드, 인터셉터 등)을 검색합니다.

```
@@파일명(cats.service)
@Injectable()
내보내기 클래스 CatsService 구현 OnModuleInit { private
    service: Service;
    constructor(private moduleRef: ModuleRef) {}

    onModuleInit() {
        this.service = this.moduleRef.get(Service);
    }
}

@@스위치
@Injectable()
@Dependencies(ModuleRef)
내보내기 클래스 CatsService {
    constructor(moduleRef) {
        this.moduleRef = moduleRef;
    }

    onModuleInit() {
        this.service = this.moduleRef.get(Service);
    }
}
```

```
    }  
}
```

경고 경고 범위가 지정된 공급자(일시적이거나 요청 범위가 지정된)는 `get()`으로 검색할 수 없습니다.

메서드를 사용할 수 없습니다. 대신 아래에 설명된 기술을 사용하세요. [여기에서](#) 범위를 제어하는 방법을 알아보세요.

글로벌 컨텍스트에서 공급자를 검색하려면(예: 공급자가 다른 모듈에 삽입된 경우) `{} '{} {}' strict: false` 옵션을 `get()`의 두 번째 인수로 전달합니다.

```
this.moduleRef.get(Service, { strict: false });
```

범위가 지정된 공급자 확인

범위가 지정된 공급자(일시적 또는 요청 범위)를 동적으로 확인하려면 공급자의 인젝션 토큰을 인수로 전달하여 `resolve()` 메서드를 사용합니다.

```
@@파일명(cats.service)  
@Injectable()  
export class CatsService implements OnModuleInit {  
  private transientService: TransientService;  
  constructor(private moduleRef: ModuleRef) {}  
  
  async onModuleInit() {  
    this.transientService = await  
this.moduleRef.resolve(TransientService);  
  }  
}  
@@스위치 @Injectable()  
@Dependencies(ModuleRef)  
내보내기 클래스 CatsService {  
  constructor(moduleRef) {  
    this.moduleRef = moduleRef;  
  }  
  
  async onModuleInit() {  
    this.transientService = await  
this.moduleRef.resolve(TransientService);  
  }  
}
```

`resolve()` 메서드는 자체 DI 컨테이너 하위 트리에서 공급자의 고유한 인스턴스를 반환합니다. 각 하위 트리에는 고유한 컨텍스트 식별자가 있습니다. 따라서 이 메서드를 두 번 이상 호출하고 인스턴스 참조를 비교하면 인스턴스 참조가 동일하지 않음을 알 수 있습니다.

```
@@파일명(cats.service)
@Injectable()
export class CatsService 구현 OnModuleInit { constructor(private
  moduleRef: ModuleRef) {}

  async onModuleInit() {
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService),
      this.moduleRef.resolve(TransientService),
    ]);
    console.log(transientServices[0] === transientServices[1]); // false
  }
}

@@스위치
@Injectable()
@Dependencies(ModuleRef)
내보내기 클래스 CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    const transientServices = await Promise.all([
      this.moduleRef.resolve(TransientService),
      this.moduleRef.resolve(TransientService),
    ]);
    console.log(transientServices[0] === transientServices[1]); // false
  }
}
```

여러 번의 `resolve()` 호출에 걸쳐 단일 인스턴스를 생성하고 생성된 동일한 DI 컨테이너 하위 트리를 공유하도록 하려면 `resolve()` 메서드에 컨텍스트 식별자를 전달하면 됩니다. 컨텍스트 식별자를 생성하려면 `ContextIdFactory` 클래스를 사용합니다. 이 클래스는 적절한 고유 식별자를 반환하는 `create()` 메서드를 제공합니다.

```
@@파일명(cats.service)
@Injectable()
export class CatsService 구현 OnModuleInit { constructor(private
  moduleRef: ModuleRef) {}

  async onModuleInit() {
    const contextId = ContextIdFactory.create();
    const transientServices = await
      Promise.all([
        this.moduleRef.resolve(TransientService, contextId),
        this.moduleRef.resolve(TransientService, contextId),
      ]);
    console.log(transientServices[0] === transientServices[1]); // true
  }
}

@@switch
```

주입 가능() @의존성

```
(ModuleRef) 내보내기 클래스
CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    const contextId = ContextIdFactory.create();
    const transientServices = await
      Promise.all([
        this.moduleRef.resolve(TransientService, contextId),
        this.moduleRef.resolve(TransientService, contextId),
      ]);
    console.log(transientServices[0] === transientServices[1]); // true
  }
}
```

정보 힌트 `ContextIdFactory` 클래스는 `@nestjs/core` 패키지에서 임포트됩니다.

요청 공급자 등록

수동으로 생성된 컨텍스트 식별자(`ContextIdFactory.create()`를 사용하여)는 네스트 종속성 주입 시스템에 의해 인스턴스화 및 관리되지 않기 때문에 요청 공급자가 정의되지 않은 DI 하위 트리를 나타냅니다.

수동으로 생성된 DI 하위 트리에 대한 사용자 지정 `REQUEST` 개체를 등록하려면

`ModuleRef#registerRequestByContextId()` 메서드를 다음과 같이 호출합니다:

```
const contextId = ContextIdFactory.create();
this.moduleRef.registerRequestByContextId(/* YOUR_REQUEST_OBJECT */,
contextId);
```

현재 하위 트리 가져오기

요청 컨텍스트 내에서 요청 범위가 지정된 프로바이더의 인스턴스를 확인해야 하는 경우가 있습니다.

`CatsService`가 요청 범위가 설정되어 있고 요청 범위가 설정된 공급자로 표시된 `CatsRepository` 인스턴스를 확인하려고 한다고 가정해 보겠습니다. 동일한 DI 컨테이너 하위 트리를 공유하려면 새 컨텍스트 식별자를 생성하는 대신 현재 컨텍스트 식별자를 가져와야 합니다(예: 위에 표시된 것처럼

`ContextIdFactory.create()` 함수 사용). 현재 컨텍스트 식별자를 얻으려면 `@Inject()` 데코레이터를 사용하여 요청 객체를 주입하는 것으로 시작하세요.

```
@@파일명(cats.service)
@Injectable()
내보내기 클래스 CatsService { 생
    성자(
        주입(요청) 비공개 요청: Record< 문자열, 알 수 없음>,
    ) {}
}
```

```

@@스위치
@Injectable()
@Dependencies(REQUEST) 내보내

기 클래스 CatsService {
  constructor(request) {
    this.request = request;
  }
}

```

정보 힌트 [여기에서](#) 요청 공급자에 대해 자세히 알아보세요.

이제 ContextIdFactory 클래스의 `getByRequest()` 메서드를 사용하여 요청 객체를 기반으로 컨텍스트 ID를 생성하고 이를 `resolve()` 호출에 전달합니다:

```

const contextId = ContextIdFactory.getByRequest(this.request);
const catsRepository = await this.moduleRef.resolve(CatsRepository,
contextId);

```

사용자 지정 클래스 동적으로 인스턴스화하기

이전에 프로바이더로 등록되지 않은 클래스를 동적으로 인스턴스화하려면 모듈 참조의 `create()` 메서드를 사용하세요.

```

@@파일명(cats.service)
@Injectable()
export class CatsService implements OnModuleInit {
  private catsFactory: CatsFactory;
  constructor(private moduleRef: ModuleRef) {}

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}

@@스위치 @Injectable()
@Dependencies(ModuleRef)
내보내기 클래스 CatsService {
  constructor(moduleRef) {
    this.moduleRef = moduleRef;
  }

  async onModuleInit() {
    this.catsFactory = await this.moduleRef.create(CatsFactory);
  }
}

```

이 기술을 사용하면 프레임워크 컨테이너 외부에서 다양한 클래스를 조건부로 인스턴스화할 수 있습니다.

지연 로딩 모듈

기본적으로 모듈은 열심히 로드되므로 애플리케이션이 로드되는 즉시 즉시 필요한지 여부에 관계없이 모든 모듈이 로드됩니다. 대부분의 애플리케이션에는 문제가 없지만, 시작 대기 시간('콜드 스타트')이 중요한 서비스 환경에서 실행되는 앱/워커의 경우 병목 현상이 발생할 수 있습니다.

지연 로딩은 특정 서비스 함수 호출에 필요한 모듈만 로드하여 부트스트랩 시간을 단축하는 데 도움이 될 수 있습니다. 또한 서비스 함수가 "워밍업"되면 다른 모듈을 비동기적으로 로드하여 후속 호출에 대한 부트스트랩 시간을 더욱 단축할 수도 있습니다(지연된 모듈 등록).

정보 힌트 Angular 프레임워크에 익숙하다면 "지연 로딩 모듈"이라는 용어를 본 적이 있을 것입니다. 이 기술은 Nest에서 기능적으로 다르므로 유사한 명명 규칙을 공유하는 완전히 다른 기능으로 생각하세요.

경고 경고 [라이프사이클 흐름](#) 메서드는 지연 로드된 모듈과 서비스에서는 호출되지 않는다는 점에 유의하세요.

시작하기

주문형 모듈을 로드하기 위해 Nest는 일반적인 방법으로 클래스에 주입할 수 있는 [LazyModuleLoader](#) 클래스를 제공합니다:

```
@@파일명(cats.service)
@Injectable()
내보내기 클래스 CatsService {
  constructor(private lazyModuleLoader: LazyModuleLoader) {}
}

@@스위치
@Injectable()
@Dependencies(LazyModuleLoader)
export class CatsService {
  constructor(lazyModuleLoader) {
    this.lazyModuleLoader = lazyModuleLoader;
  }
}
```

정보 힌트 [LazyModuleLoader](#) 클래스는 [@nestjs/core](#) 패키지에서 임포트됩니다.

또는 다음과 같이 애플리케이션 부트스트랩 파일([main.ts](#)) 내에서 [LazyModuleLoader](#) 공급자에 대한 참조를 얻을 수 있습니다:

```
// "app"은 Nest 애플리케이션 인스턴스를 나타냅니다 const  
lazyModuleLoader = app.get(LazyModuleLoader);
```

이제 다음 구성을 사용하여 모든 모듈을 로드할 수 있습니다:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);
```

정보 힌트 "지연 로드된" 모듈은 첫 번째 `LazyModuleLoader#load` 메서드 호출 시 캐시됩니다. 즉, 연속적으로 `LazyModule`을 로드하려고 시도할 때마다 매우 빠르게 모듈을 다시 로드하는 대신 캐시된 인스턴스를 반환합니다.

"`LazyModule`" 로드 시도: 1회:

2.379ms

"`LazyModule`" 로드 시도: 2회:

0.294ms

"`LazyModule`" 로드 시도: 3회:

0.303ms

또한 '지연 로드된' 모듈은 애플리케이션 부트스트랩에 열심히 로드된 모듈과 앱에 나중에 등록된 다른 지연 모듈과 동일한 모듈 그래프를 공유합니다.

여기서 `lazy.module.ts`는 일반 Nest 모듈을 내보내는 TypeScript 파일입니다(추가 변경이 필요하지 않음).

`LazyModuleLoader#load` 메서드는 내부 공급자 목록을 탐색하고 해당 주입 토큰을 조회 키로 사용하여 모든 공급자에 대한 참조를 얻을 수 있는 (`LazyModule`의) **모듈 참조를 반환합니다**.

예를 들어 다음과 같은 정의가 있는 `LazyModule`이 있다고 가정해 보겠습니다:

```
모듈({
  제공자: [LazyService], 수출:
  [LazyService],
})
내보내기 클래스 LazyModule {}
```

정보 힌트 지연 로드된 모듈은 글로벌 모듈로 등록할 수 없습니다(정적으로 등록된 모든 모듈이 이미 인스턴스화된 상태에서 온디맨드 방식으로 지연 등록되므로 의미가 없습니다). 마찬가지로 등록된 글로벌 인핸서(가드/인터셉터 등)도 제대로 작동하지 않습니다.

이를 통해 다음과 같이 `LazyService` 제공자에 대한 참조를 얻을 수 있습니다:

```
const { LazyModule } = await import('./lazy.module');
const moduleRef = await this.lazyModuleLoader.load(() => LazyModule);

const { LazyService } = await import('./lazy.service');
const lazyService = moduleRef.get(LazyService);
```

경고 Webpack을 사용하는 경우 `tsconfig.json` 파일을 업데이트해야 합니다.

`compilerOptions.module`을 "esnext"로 설정하고 "node"를 값으로 하여

`compilerOptions.moduleResolution` 속성을 추가해야 합니다:

```
{  
  "컴파일러옵션": { "module":  
    "esnext",  
    "moduleResolution": "노드",  
    ...  
  }  
}
```

이러한 옵션을 설정하면 [코드 분할](#) 기능을 활용할 수 있습니다.

지연 로딩 컨트롤러, 게이트웨이 및 리졸버

Nest의 컨트롤러(또는 GraphQL 애플리케이션의 리졸버)는 경로/경로/토픽(또는 쿼리/변이)의 집합을 나타내므로 [LazyModuleLoader](#) 클래스를 사용하여 지연 로드할 수 없습니다.

오류 경고 지연 로드된 모듈 내부에 등록된 컨트롤러, [리졸버](#) 및 [게이트웨이는](#) 예상대로 작동하지 않습니다. 마찬가지로 미들웨어 함수를 온디맨드 방식으로 등록할 수 없습니다([MiddlewareConsumer](#) 인터페이스를 구현하여).

예를 들어 Fastify 드라이버를 사용하여 REST API(HTTP 애플리케이션)를 구축한다고 가정해 보겠습니다 (@nestjs/platform-fastify 패키지 사용). Fastify는 애플리케이션이 준비되거나 메시지를 성공적으로 수신한 후에는 경로를 등록할 수 없습니다. 즉, 모듈의 컨트롤러에 등록된 경로 매핑을 분석하더라도 런타임에 등록할 방법이 없기 때문에 모든 지연 로드된 경로에 액세스할 수 없습니다.

마찬가지로, 트위터가 @nestjs/microservices 패키지의 일부로 제공하는 일부 전송 전략(Kafka, gRPC 또는 RabbitMQ 포함)은 연결이 설정되기 전에 특정 토픽/채널을 구독/청취해야 합니다. 애플리케이션이 메시지 수신을 시작하면 프레임워크가 새 토픽을 구독/수신할 수 없게 됩니다.

마지막으로, 코드 우선 접근 방식이 활성화된 @nestjs/graphql 패키지는 메타데이터를 기반으로 GraphQL 스키마를 즉석에서 자동으로 생성합니다. 즉, 모든 클래스를 미리 로드해야 합니다. 그렇지 않으면 적절하고 유효한 스키마를 생성할 수 없습니다.

일반적인 사용 사례

대부분의 경우, 작업자/크론 작업/람다 및 서비스 함수/웹훅이 입력 인수(경로 경로/날짜/쿼리 매개변수 등)에 따라 다른 서비스(다른 로직)를 트리거해야 하는 상황에서 자연 로드된 모듈을 볼 수 있습니다. 반면에 자연 로딩 모듈은 시작 시간이 다소 무관한 모놀리식 애플리케이션의 경우 그다지 의미가 없을 수 있습니다.

실행 컨텍스트

Nest는 여러 애플리케이션 컨텍스트(예: Nest HTTP 서버 기반, 마이크로서비스 및 웹소켓 애플리케이션 컨텍스트)에서 작동하는 애플리케이션을 쉽게 작성할 수 있도록 도와주는 여러 유ти리티 클래스를 제공합니다. 이러한 유ти리티는 현재 실행 컨텍스트에 대한 정보를 제공하여 광범위한 컨트롤러, 메서드 및 실행 컨텍스트에서 작동할 수 있는 일반 [가드](#), [필터](#) 및 [인터셉터](#)를 빌드하는 데 사용할 수 있습니다.

이 장에서는 이러한 클래스 두 가지를 다룹니다: [ArgumentsHost](#)와 [ExecutionContext](#)입니다.

ArgumentsHost 클래스

[ArgumentsHost](#) 클래스는 핸들러에 전달되는 인수를 검색하는 메서드를 제공합니다. 이 클래스를 사용하면 인수를 검색할 적절한 컨텍스트(예: HTTP, RPC(마이크로서비스) 또는 WebSockets)를 선택할 수 있습니다. 프레임워크는 일반적으로 [호스트](#) 매개변수로 참조되는 [ArgumentsHost](#)의 인스턴스를 사용자가 액세스하려는 위치에 제공합니다. 예를 들어 [예외 필터의 catch\(\)](#) 메서드는 [ArgumentsHost](#) 인스턴스와 함께 호출됩니다.

[ArgumentsHost](#)는 단순히 핸들러의 인수를 추상화하는 역할을 합니다. 예를 들어, HTTP 서버 애플리케이션(@nestjs/platform-express를 사용하는 경우)의 경우 [호스트](#) 객체는 Express의 [\[request, response, next\]](#) 배열을 캡슐화하며, 여기서 request는 요청 객체, response는 응답 객체, next는 애플리케이션의 요청-응답 사이클을 제어하는 함수입니다. 반면, GraphQL 애플리케이션의 경우 [호스트](#) 객체에는 [\[root, args, context, info\]](#) 배열이 포함됩니다.

현재 애플리케이션 컨텍스트

여러 애플리케이션 컨텍스트에서 실행되는 일반 [가드](#), [필터](#) 및 [인터셉터](#)를 빌드할 때는 메서드가 현재 실행 중인 애플리케이션 유형을 확인할 수 있는 방법이 필요합니다. 이 작업은 [ArgumentsHost](#)의 [getType\(\)](#) 메서드를 사용하여 수행합니다:

```
if (host.getType() === 'http') {  
    // 일반 HTTP 요청(REST)의 컨텍스트에서만 중요한 작업을 수행합니다.  
} else if (host.getType() === 'rpc') {  
    // 마이크로서비스 요청의 컨텍스트에서만 중요한 작업을 수행합니다.  
} else if (host.getType<GqlContextType>() === 'graphql') {  
    // GraphQL 요청의 컨텍스트에서만 중요한 작업을 수행합니다.  
}
```

정보 힌트 GqlContextType은 [@nestjs/graphql](#) 패키지에서 가져옵니다.

애플리케이션 유형을 사용할 수 있게 되면 아래와 같이 보다 일반적인 컴포넌트를 작성할 수 있습

니다. 호스트 핸들러 인수

핸들러에 전달되는 인자 배열을 검색하려면, 한 가지 방법은 호스트 객체의

`getArgs()` 메서드를 사용합니다.

```
const [req, res, next] = host.getArgs();
```

색인별로 특정 인수를 추출하려면 `getArgByIndex()` 메서드를 사용하면 됩니다:

```
const request = host.getArgByIndex(0);
const response = host.getArgByIndex(1);
```

이 예제에서는 인덱스로 요청 및 응답 객체를 검색했는데, 이는 애플리케이션을 특정 실행 컨텍스트에 연결하기 때문에 일반적으로 권장되지 않습니다. 대신 [호스트](#) 객체의 유ти리티 메서드 중 하나를 사용하여 애플리케이션에 적합한 애플리케이션 컨텍스트로 전환함으로써 코드를 보다 강력하고 재사용 가능하게 만들 수 있습니다. 컨텍스트 전환 유ти리티 메서드는 다음과 같습니다.

```
/**
 * 컨텍스트를 RPC로 전환합니다.
 */
switchToRpc(): RpcArgumentsHost;
/** 
 * 컨텍스트를 HTTP로 전환합니다.
 */
switchToHttp(): HttpArgumentsHost;
/** 
 * 컨텍스트를 웹소켓으로 전환합니다.
 */
switchToWs(): WsArgumentsHost;
```

`스위치투情商트()` 메서드를 사용하여 이전 예제를 다시 작성해 보겠습니다. `host.switchToHttp()` 헬퍼

호출은 HTTP 애플리케이션 컨텍스트에 적합한 `HttpArgumentsHost` 객체를 반환합니다.

`HttpArgumentsHost` 객체에는 원하는 객체를 추출하는 데 사용할 수 있는 두 가지 유용한 메서드가 있습니다. 또한 이 경우 Express 유형 어설션을 사용하여 기본 Express 유형 객체를 반환합니다:

```
const ctx = host.switchToHttp();
const request = ctx.getRequest<Request>();
const response = ctx.getResponse<Response>();
```

마찬가지로 `WsArgumentsHost`와 `RpcArgumentsHost`에는 마이크로서비스 및 웹 소켓 컨텍스트에서 적절한 객체를 반환하는 메서드가 있습니다. 다음은 `WsArgumentsHost`에 대한 메서드입니다:

```
내보내기 인터페이스 WsArgumentsHost {
```

```
/**
```

```
* 데이터 객체를 반환합니다.
```

```

    */
    getData<T>(): T;
    /**
     * 클라이언트 객체를 반환합니다.
     */
    getClient<T>(): T;
}

```

다음은 [RpcArgumentsHost](#)의 메서드입니다:

```

내보내기 인터페이스 RpcArgumentsHost {
    /**
     * 데이터 객체를 반환합니다.
     */
    getData<T>(): T;

    /**
     * 컨텍스트 객체를 반환합니다.
     */
    getContext<T>(): T;
}

```

실행 컨텍스트 클래스

[ExecutionContext](#)는 [ArgumentsHost](#)를 확장하여 현재 실행 프로세스에 대한 추가 세부 정보를 제공합니다. [ArgumentsHost](#)와 마찬가지로 Nest는 가드의 [canActivate\(\)](#) 메서드나 인터셉터의 [intercept\(\)](#) 메서드 등 필요할 수 있는 곳에 [ExecutionContext](#)의 인스턴스를 제공합니다. 다음과 같은 메서드를 제공합니다:

```

내보내기 인터페이스 ExecutionContext extends ArgumentsHost {
    /**
     * 현재 핸들러가 속한 컨트롤러 클래스의 유형을 반환합니다.
     */
    getClass<T>(): Type<T>;
    /**
     * 다음에 호출될 핸들러(메서드)에 대한 참조를 반환합니다.
     * 요청 파이프라인으로 이동합니다.
     */
    getHandler(): 함수;
}

```

`getHandler()` 메서드는 호출하려는 핸들러에 대한 참조를 반환합니다. `getClass()` 메서드는 이 특정 핸들러가 속한 `Controller` 클래스의 유형을 반환합니다. 예를 들어 HTTP 컨텍스트에서 현재 처리된 요청이 `POST` 요청인 경우, `create()` 메서드에 바인딩된

CatsController의 경우, getHandler()는 create() 메서드에 대한 참조를 반환하고 getClass()는 인스턴스가 아닌 CatsController 유형을 반환합니다.

```
const methodKey = ctx.getHandler().name; // "create" const  
className = ctx.getClass().name; // "CatsController"
```

현재 클래스와 핸들러 메서드 모두에 대한 참조에 액세스할 수 있는 기능은 뛰어난 유연성을 제공합니다. 가장 중요한 것은 가드 또는 인터셉터 내에서 Reflector#createDecorator를 통해 생성된 데코레이터 또는 내장된 @SetMetadata() 데코레이터를 통해 메타데이터 세트에 액세스할 수 있다는 점입니다. 이 사용 사례는 아래에서 다룹니다.

리플렉션 및 메타데이터

Nest는 Reflector#createDecorator 메서드를 통해 생성된 데코레이터와 내장된 @SetMetadata() 데코레이터를 통해 라우트 핸들러에 사용자 정의 메타데이터를 첨부할 수 있는 기능을 제공합니다. 이 섹션에서는 두 가지 접근 방식을 비교하고 가드 또는 인터셉터 내에서 메타데이터에 액세스하는 방법을 살펴보겠습니다.

Reflector#createDecorator를 사용하여 강력한 타입의 데코레이터를 만들려면 타입 인수를 지정해야 합니다. 예를 들어 문자열 배열을 인수로 받는 Roles 데코레이터를 만들어 보겠습니다.

```
@@파일명(역할.데코레이터)  
'@nestjs/core'에서 { Reflector }를 가져옵니다;  
  
export const Roles = Reflector.createDecorator<string[]>();
```

여기서 Roles 데코레이터는 문자열[] 타입의 단일 인수를 받는 함수입니다. 이제 이 데코레이터를 사용하려면 핸들러에 주석을 달기만 하면 됩니다:

```
@@파일명(cats.controller)
@Post()
@Roles(['admin'])
async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
}

@@스위치

@포스트()
@Roles(['admin'])
@Bind(Body())
async create(createCatDto) {
    this.catsService.create(createCatDto);
}
```

여기에서는 관리자 역할이 있는 사용자만 이 경로에 액세스하도록 허용해야 함을 나타내는 역할 데코레이터 메타데이터를 `create()` 메서드에 첨부했습니다.

경로의 역할(사용자 지정 메타데이터)에 액세스하려면 **Reflector** 헬퍼 클래스를 다시 사용하겠습니다.

Reflector

는 일반적인 방법으로 클래스에 주입할 수 있습니다:

```
@@파일명(roles.guard)
@Injectable()
내보내기 클래스 RolesGuard {
  constructor(private reflector: Reflector) {}
}

@@스위치

@Injectable()
@Dependencies(Reflector)
내보내기 클래스 CatsService {
  constructor(reflector) {
    this.reflector = reflector;
  }
}
```

정보 힌트 **리플렉터** 클래스는 `@nestjs/core` 패키지에서 가져옵니다.

이제 핸들러 메타데이터를 읽으려면 `get()` 메서드를 사용합니다:

```
const roles = this.reflector.get(Roles, context.getHandler());
```

`Reflector#get` 메서드를 사용하면 데코레이터 참조와 메타데이터를 검색할 컨텍스트(데코레이터 대상)의 두 가지 인수를 전달하여 메타데이터에 쉽게 액세스할 수 있습니다. 이 예에서 지정된 데코레이터는 **Roles**입니다 (위의 `roles.decorator.ts` 파일을 다시 참조하세요). 컨텍스트는 `context.getHandler()`를 호출하여 제공되며, 그 결과 현재 처리된 라우트 핸들러의 메타데이터를 추출합니다. `getHandler()`는 라우트 핸들러 함수에 대한 참조를 제공한다는 점을 기억하세요.

또는 컨트롤러 수준에서 메타데이터를 적용하여 컨트롤러 클래스의 모든 경로에 적용하여 컨트롤러를 구성할 수도 있습니다.

```
@@파일명(cats.controller)
@Roles(['admin'])
@Controller('cats')
내보내기 클래스 CatsController {}

@@switch
@Roles(['admin'])
@Controller('cats')
내보내기 클래스 CatsController {}
```

이 경우 컨트롤러 메타데이터를 추출하기 위해 `context.getHandler()` 대신 `context.getClass()`를 두 번째 인수로 전달합니다(메타데이터 추출을 위한 컨텍스트로 컨트롤러 클래스를 제공하기 위해):

```
@@파일명(roles.guard)
const roles = this.reflector.get(Roles, context.getClass());
```

여러 수준에서 메타데이터를 제공할 수 있으므로 여러 컨텍스트에서 메타데이터를 추출하고 병합해야 할 수도 있습니다. `Reflector` 클래스는 이를 지원하는 데 사용되는 두 가지 유ти리티 메서드를 제공합니다. 이 메서드들은 컨트롤러와 메서드 메타데이터를 한 번에 추출하고 서로 다른 방식으로 결합합니다.

두 수준 모두에서 역할 메타데이터를 제공한 다음 시나리오를 생각해 보세요.

```
@@파일명(cats.controller)
@Roles(['user'])
@Controller('cats')
내보내기 클래스 CatsController {
  @Post()
  @Roles(['admin'])
  async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
  }
}

스위치 @역할(['사용자']) @컨
트롤러('고양이')

export class CatsController {}
  @Post()
  @Roles(['admin'])
  @Bind(Body())
  async create(createCatDto) {
    this.catsService.create(createCatDto);
  }
}
```

기본 역할로 '사용자'를 지정하고 특정 메서드에 대해 선택적으로 재정의하려는 경우 `getAllAndOverride()` 메서드를 사용할 수 있습니다.

```
const roles = this.reflector.getAllAndOverride(Roles,
[context.getHandler(), context.getClass()]);
```

위의 메타데이터를 사용하여 `create()` 메서드의 컨텍스트에서 실행되는 이 코드가 포함된 가드는 `['admin']`을 포함하는 역할을 생성합니다.

둘 다에 대한 메타데이터를 가져와 병합하려면(이 메서드는 배열과 객체를 모두 병합합니다)

getAllAndMerge() 메서드를 사용합니다:

```
const roles = this.reflector.getAllAndMerge(Roles, [context.getHandler(),  
context.getClass()]);
```

이렇게 하면 `['user', 'admin']`을 포함하는 역할이 됩니다.

이 두 병합 메서드 모두 첫 번째 인자로 메타데이터 키를 전달하고 두 번째 인자로 메타데이터 대상 컨텍스트 배열(즉, `getHandler()` 및/또는 `getClass()` 메서드에 대한 호출)을 전달합니다.

낮은 수준의 접근 방식

앞서 언급했듯이 `Reflector#createDecorator`를 사용하는 대신 내장된

`SetMetadata()` 데코레이터를 사용하여 핸들러에 메타데이터를 첨부할 수 있습니다.

```
@@파일명(cats.controller)
@Post()
SetMetadata('roles', ['admin'])
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@스위치

@포스트()
SetMetadata('roles', ['admin'])
@Bind(Body())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

정보 힌트 `@SetMetadata()` 데코레이터는 `@nestjs/common` 패키지에서 가져온 것입니다.

위의 구조에서는 `roles` 메타데이터(`roles`는 메타데이터 키이고 `['admin']`은 연관된 값)를 `create()` 메서드에 첨부했습니다. 이렇게 해도 작동하지만 `@SetMetadata()`를 경로에 직접 사용하는 것은 좋은 방법이 아닙니다. 대신 아래와 같이 자체 데코레이터를 만들 수 있습니다:

```
@@파일명(역할.데코레이터)
'@nestjs/common'에서 { SetMetadata }를 가져옵니다;

export const Roles = (...roles: string[]) => SetMetadata('roles', roles);
@@switch
'@nestjs/common'에서 { SetMetadata }를 가져옵니다;

export const Roles = (...roles) => SetMetadata('roles', roles);
```

이 접근 방식은 훨씬 더 깔끔하고 가독성이 높으며 `Reflector#createDecorator` 접근 방식과 다소 유사합니다. 차이점은 `@SetMetadata`를 사용하면 메타데이터 키와 값을 더 많이 제어할 수 있고 둘 이상의 인수를 받는 데코

레이터를 만들 수도 있다는 점입니다.

이제 사용자 정의 `@Roles()` 데코레이터가 생겼으므로 이를 사용하여 `create()` 메서드를 데코레이션할 수 있습니다.

```

@@파일명(cats.controller)
@Post()
@Roles('admin')
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

스위치 @포스트()

@역할('관리자') @

바인드(본문())
async create(createCatDto) {
  this.catsService.create(createCatDto);
}

```

경로의 역할(사용자 지정 메타데이터)에 액세스하려면 `Reflector` 헬퍼 클래스를 다시 사용하겠습니다:

```

@@파일명(roles.guard)
@Injectable()
내보내기 클래스 RolesGuard {
  constructor(private reflector: Reflector) {}
}

@@스위치

@Injectable()
@Dependencies(Reflector)
내보내기 클래스 CatsService {
  constructor(reflector) {
    this.reflector = reflector;
  }
}

```

정보 힌트 `리플렉터` 클래스는 `@nestjs/core` 패키지에서 가져옵니다.

이제 핸들러 메타데이터를 읽으려면 `get()` 메서드를 사용합니다.

```
const roles = this.reflector.get<string[]>('roles', context.getHandler());
```

여기서는 데코레이터 참조를 전달하는 대신 메타데이터 키를 첫 번째 인수로 전달합니다(이 경우 '`roles`'). 다른 모든 것은 `Reflector#createDecorator` 예시와 동일하게 유지됩니다.

라이프사이클 이벤트

Nest 애플리케이션과 모든 애플리케이션 요소에는 Nest에서 관리하는 라이프사이클이 있습니다. Nest는 주요 수명 주기 이벤트에 대한 가시성을 제공하는 수명 주기 흙을 제공하며, 이벤트가 발생하면 모듈, 공급자 또는 컨트롤러에서 등록된 코드를 실행하는 기능을 제공합니다.

라이프사이클 순서

다음 다이어그램은 애플리케이션이 부트스트랩된 시점부터 노드 프로세스가 종료될 때까지의 주요 애플리케이션 수명 주기 이벤트의 순서를 보여줍니다. 전체 수명 주기는 초기화, 실행, 종료의 세 단계로 나눌 수 있습니다. 이 수명 주기를 사용하면 모듈과 서비스의 적절한 초기화를 계획하고, 활성 연결을 관리하고, 종료 신호를 수신할 때 애플리케이션을 정상적으로 종료할 수 있습니다.



라이프사이클 이벤트

라이프사이클 이벤트는 애플리케이션 부트스트랩과 종료 중에 발생합니다. Nest는 다음 각 수명 주기 이벤트에서 모듈, 공급자 및 컨트롤러에 등록된 수명 주기 흙 메서드를 호출합니다(아래 설명된 대로 셋다운 흙을 먼저 활성화해야 함). 위의 다이어그램에서 볼 수 있듯이 Nest는 적절한 기본 메서드를 호출하여 연결 수신을 시작하고 연결 수신을 중지하기도 합니다.

다음 표에서 `onModuleDestroy`, `beforeApplicationShutdown` 및 `onApplicationShutdown`은 명시적으로 `app.close()`를 호출하거나 프로세스가 특수 시스템 신호(예: SIGTERM)를 수신하고 애플리케이션 부트스트랩에서 `enableShutdownHook`을 올바르게 호출한 경우에만 발동됩니다(아래 애플리케이션 종료 부분 참조).

라이프사이클 후크 메서드	후크 메서드 호출을 트리거하는 라이프사이클 이벤트
<code>onModuleInit()</code>	호스트 모듈의 종속성이 다음과 같이 설정되면 호출됩니다.
<code>onApplicationBootstrap()</code>	<code>beforeApplicationShutdown()</code> *
<code>onModuleDestroy()</code> *	

해결되었습니다.

모든 모듈이 초기화되면 호출
되지만 연결을 수신 대기하
기 전에 호출됩니다.

종료 신호(예: SIGTERM)가 수신된 후 호출됩니다.

모든 `onModuleDestroy()` 핸들러가 완료된 후 호출됩니다(프로미스
가 해결되거나 거부된 경우);
완료되면(프로미스가 해결되거나 거부되면) 기존의 모든 연결이 닫힙
니다(`app.close()` 호출).

`onApplicationShutdown()`*

연결이 닫힌 후 호출됩니다(`app.close()` 해결).

* 이러한 이벤트의 경우 `app.close()`를 명시적으로 호출하지 않는 경우, SIGTERM과 같은 시스템 신호와 함께 작
동하도록 옵트인해야 합니다. 아래 [애플리케이션 종료](#)를 참조하세요.

경고 경고 위에 나열된 라이프사이클 혹은 요청 범위가 지정된 클래스에 대해 트리거되지 않습니다.
요청 범위 클래스는 애플리케이션 수명 주기에 묶여 있지 않으며 수명을 예측할 수 없습니다.
각 요청에 대해 전용으로 생성되며 응답이 전송된 후 자동으로 가비지 수집됩니다.

정보 힌트 `onModuleInit()` 및 `onApplicationBootstrap()`의 실행 순서는 이전 흑을 기다리는 모듈 가져오기 순서에 직접적으로 의존합니다.

사용법

각 라이프사이클 혹은 인터페이스로 표현됩니다. 인터페이스는 TypeScript 컴파일 이후에는 존재하지 않기 때문에 기술적으로는 선택 사항입니다. 그럼에도 불구하고 강력한 타이핑 및 편집기 도구의 이점을 활용하려면 인터페이스를 사용하는 것이 좋습니다. 라이프사이클 흑을 등록하려면 적절한 인터페이스를 구현하세요. 예를 들어 특정 클래스(예: 컨트롤러, 프로바이더 또는 모듈)에서 모듈 초기화 중에 호출할 메서드를 등록하려면 아래와 같이 [온모듈인잇\(\)](#) 메서드를 제공함으로써 [온모듈인잇](#) 인터페이스를 구현합니다:

```
@@파일명()
'@nestjs/common'에서 { Injectable, OnModuleInit }을 임포트합니다;

@Injectable()
내보내기 클래스 UserService 구현 온 모듈 초기화 { onModuleInit() {
    console.log(`모듈이 초기화되었습니다.`);
}
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트합니다;

@Injectable()
내보내기 클래스 UserService {
    onModuleInit() {
        console.log(`모듈이 초기화되었습니다.`);
    }
}
```

비동기 초기화

`OnModuleInit` 및 `OnApplicationBootstrap` 흑을 사용하면 애플리케이션 초기화 프로세스를 지연시킬 수 있습니다([프로미스](#)를 반환하거나 메서드를 비동기로 표시하고 메서드 본문에서 비동기 메서드 완료를 [기다림](#)).

```
@@파일명()
async onModuleInit(): Promise<void> {
    await this.fetch();
}

@@switch
async onModuleInit() {
```

```
    await this.fetch();
}
```

애플리케이션 종료

`onModuleDestroy()`, `beforeApplicationShutdown()` 및 `onApplicationShutdown()` 혹은 종료 단계에서 호출됩니다(`app.close()` 명시적 호출에 대한 응답 또는 옵트인한 경우 SIGTERM과 같은 시스템 신호를 수신할 때). 이 기능은 컨테이너의 라이프사이클을 관리하기 위해 [Kubernetes](#)와 함께 자주 사용되며, [Heroku](#)는 다이노 또는 이와 유사한 서비스를 위해 사용합니다.

셋다운 후크 리스너는 시스템 리소스를 소모하므로 기본적으로 비활성화되어 있습니다. 셋다운 흑을 사용하려면 `enableShutdownHooks()`를 호출하여 리스너를 활성화해야 합니다:

```
'@nestjs/core'에서 { NestFactory }를 임포트하고,
'./app.module'에서 { AppModule }을 임포트합니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule);

  // 셋다운 후크 수신 시작 app.enableShutdownHooks();

  await app.listen(3000);
}

부트스트랩();
```

경고 경고 고유한 플랫폼 제한으로 인해 NestJS는 Windows에서 애플리케이션 종료 후크에 대한 지원이 제한적입니다. SIGINT는 물론 SIGBREAK와 어느 정도 SIGHUP도 작동할 것으로 예상할 수 있습니다 - [자세히 보기](#). 그러나 작업 관리자에서 프로세스를 종료하는 것은 무조건적이기 때문에, 즉 애플리케이션이 이를 감지하거나 방지할 수 있는 방법이 없기 때문에 SIGTERM은 Windows에서 작동하지 않습니다.

Windows에서 SIGTERM, SIGBREAK 리스너를 어떻게 처리하는지를 자세히 알아보면 Node.js의 문제점에 초점을 맞춰 Node.js 앱을 실행하는 경우 (예: Jest로 개발한 Node.js 실행체를 참조하세요) Node.js에서 과도한 리스너 프로세스에 대해 불만을 제기할 수 있습니다. 이러한 이유로 `enableShutdownHooks()`는 기본적으로 활성화되지 않습니다. 단일 노드 프로세스에서 여러 인스턴스를 실행하는 경우 이 조건에 유의하세요.

애플리케이션이 종료 신호를 받으면 해당 신호를 첫 번째 매개변수로 사용하여 등록된 `onModuleDestroy()`,

`beforeApplicationShutdown()`, `onApplicationShutdown()` 메서드(위에 설명된 순서대로)를 호출합니다. 등록된 함수가 비동기 호출을 기다리는 경우(프로미스를 반환하는 경우) Nest는 프로미스가 해결되거나 거부될 때까지 시퀀스를 계속 진행하지 않습니다.

`@@파일명()`

`@Injectable()`

```
UserService 클래스는 OnApplicationShutdown을 구현합니다 {

    onApplicationShutdown(신호: 문자열) {
        console.log(signal); // 예: "SIGINT"
    }
}

@@스위치

@Injectable()

UserService 클래스는 OnApplicationShutdown을 구현합니다 {

    onApplicationShutdown(signal) {
        console.log(signal); // 예: "SIGINT"
    }
}
```

정보 `app.close()`를 호출해도 노드 프로세스가 종료되는 것이 아니라 `onModuleDestroy()`

및 `onApplicationShutdown()` 후크만 트리거되므로 일정 간격, 장기간 실행 중인 백그라운드

작업 등이 있는 경우 프로세스가 자동으로 종료되지 않습니다.

플랫폼 불가지론

Nest는 플랫폼에 구애받지 않는 프레임워크입니다. 즉, 다양한 유형의 애플리케이션에서 사용할 수 있는 재사용 가능한 논리적 부분을 개발할 수 있습니다. 예를 들어, 대부분의 컴포넌트는 서로 다른 기본 HTTP 서버 프레임워크(예: Express 및 Fastify)에서 변경 없이 재사용할 수 있으며, 심지어 서로 다른 유형의 애플리케이션(예: HTTP 서버 프레임워크, 전송 계층이 다른 마이크로서비스 및 웹 소켓)에서도 재사용할 수 있습니다.

한 번 구축하면 어디서나 사용 가능

이 문서의 개요 섹션에서는 주로 HTTP 서버 프레임워크를 사용하는 코딩 기법(예: REST API를 제공하는 앱 또는 MVC 스타일의 서버 측 렌더링 앱 제공)을 보여 줍니다. 그러나 이러한 모든 빌딩 블록은 다양한 전송 계층([마이크로서비스](#) 또는 [웹소켓](#)) 위에서 사용할 수 있습니다.

또한 Nest에는 전용 [GraphQL](#) 모듈이 함께 제공됩니다. GraphQL을 REST API를 제공하는 것과 동일하게 API 계층으로 사용할 수 있습니다.

또한 [애플리케이션 컨텍스트](#) 기능을 사용하면 Nest를 기반으로 CRON 작업 및 CLI 앱과 같은 모든 종류의 Node.js 애플리케이션을 만들 수 있습니다.

Nest는 애플리케이션에 더 높은 수준의 모듈성과 재사용성을 제공하는 Node.js 앱을 위한 본격적인 플랫폼이 되고자 합니다. 한 번 빌드하여 어디서나 사용하세요!

테스트

자동화된 테스트는 모든 진지한 소프트웨어 개발 노력의 필수적인 부분으로 간주됩니다. 자동화를 사용하면 개발 중에 개별 테스트 또는 테스트 스위트를 쉽고 빠르게 반복할 수 있습니다. 이를 통해 릴리스가 품질 및 성능 목표를 충족하도록 보장할 수 있습니다. 자동화는 커버리지를 늘리고 개발자에게 더 빠른 피드백 루프를 제공하는 데 도움이 됩니다. 자동화는 개별 개발자의 생산성을 높이고 소스 코드 제어 체크인, 기능 통합 및 버전 릴리스와 같은 중요한 개발 수명 주기 시점에 테스트를 실행할 수 있도록 합니다.

이러한 테스트는 단위 테스트, 엔드투엔드(e2e) 테스트, 통합 테스트 등 다양한 유형에 걸쳐 있는 경우가 많습니다. 이러한 테스트의 이점은 의심할 여지가 없지만, 설정하는 것이 지루할 수 있습니다. Nest는 효과적인 테스트를 포함한 개발 모범 사례를 장려하기 위해 노력하고 있으므로 개발자와 팀이 테스트를 빌드하고 자동화하는 데 도움이 되는 다음과 같은 기능이 포함되어 있습니다. Nest:

- 구성 요소에 대한 기본 단위 테스트와 애플리케이션에 대한 E2E 테스트를 자동으로 스캐폴드합니다.
- 기본 툴링(예: 격리된 모듈/애플리케이션 로더를 빌드하는 테스트 러너)을 제공합니다.
- 테스트 도구에 구애받지 않고 [Jest](#) 및 [Supertest](#)와의 통합을 즉시 제공하며 테스트 환경에서 Nest 종속성 주입 시스템을 사용하여 쉽게 모킹할 수 있습니다.

구성 요소

앞서 언급했듯이 Nest는 특정 툴을 강요하지 않으므로 원하는 테스트 프레임워크를 사용할 수 있습니다. 테스트 러너 등 필요한 요소만 교체하기만 하면 Nest의 기성 테스트 기능의 이점을 그대로 누릴 수 있습니다.

설치

시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save-dev @nestjs/testing
```

단위 테스트

다음 예제에서는 두 개의 클래스를 테스트합니다: `CatsController`와 `CatsService`를 테스트합니다. 앞서 언급했듯이 Jest는 기본 테스트 프레임워크로 제공됩니다. 테스트 실행자 역할을 하며 모킹, 스파이 등에 도움이 되는 어설트 함수와 테스트-더블 유ти리티도 제공합니다. 다음 기본 테스트에서는 이러한 클래스를 수동으로 인스턴

스화하고 컨트롤러와 서비스가 API 계약을 이행하는지 확인합니다.

```
@@파일명(cats.controller.spec)
'./cats.controller'에서 { CatsController }를 가져오고,
'./cats.service'에서 { CatsService }를 가져옵니다;

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

  beforeEach(() => {
```

```
catsService = 새로운 CatsService();
catsController = 새로운 CatsController(catsService);
});

describe('findAll', () => {
  it('should return an array of cats', async () => {
    const result = ['test'];
    jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

    기대(await catsController.findAll()).toBe(결과);
  });
});
});

@switch
'./cats.controller'에서 { CatsController }를 가져오고,
'./cats.service'에서 { CatsService }를 가져옵니다;

describe('CatsController', () => {
  let catsController;
  catsService;

  beforeEach(() => {
    catsService = 새로운 CatsService();
    catsController = 새로운 CatsController(catsService);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      기대(await catsController.findAll()).toBe(결과);
    });
  });
});
});
```

정보 힌트 테스트 파일은 테스트하는 클래스 근처에 보관하세요. 테스트 파일은 `.spec` 또는 `.test` 접미사.

위의 샘플은 사소한 것이기 때문에 Nest와 관련된 어떤 것도 테스트하고 있지 않습니다. 실제로 종속성 주입도 사용하지 않습니다(`CatsService`의 인스턴스를 `catsController`에 전달한 것을 주목하세요). 테스트 대상 클래스를 수동으로 인스턴스화하는 이러한 형태의 테스트는 프레임워크와 독립적이기 때문에 종종 격리 테스트라고 합니다. Nest 기능을 보다 광범위하게 사용하는 애플리케이션을 테스트하는 데 도움이 되는 몇 가지 고급 기능을 소개하겠습니다.

테스트 유ти리티

`nestjs/testing` 패키지는 보다 강력한 테스트 프로세스를 가능하게 하는 일련의 유ти리티를 제공합니다. 내장된 `Test` 클래스를 사용하여 이전 예제를 다시 작성해 보겠습니다:

```
@@파일명(cats.controller.spec)

'@nestjs/testing'에서 { Test }를 가져옵니다;

다;

'./cats.controller'에서 { CatsController }를 가져오고,
'./cats.service'에서 { CatsService }를 가져옵니다;

describe('CatsController', () => {
  let catsController: CatsController;
  let catsService: CatsService;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [CatsController],
      공급자: [CatsService],
    }).compile();

    catsService = moduleRef.get<CatsService>(CatsService);
    catsController = moduleRef.get<CatsController>(CatsController);
  });

  describe('findAll', () => {
    it('should return an array of cats', async () => {
      const result = ['test'];
      jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

      기대(await catsController.findAll()).toBe(결과);
    });
  });
});

@@switch

'@nestjs/testing'에서 { Test }를 가져옵니다;
'./cats.controller'에서 { CatsController }를 가져오고,
'./cats.service'에서 { CatsService }를 가져옵니다;

describe('CatsController', () => {
  let catsController;
  catsService;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [CatsController],
      공급자: [CatsService],
    }).compile();

    catsService = moduleRef.get(CatsService);
    catsController = moduleRef.get(CatsController);
  });

  기대(await catsController.findAll()).toBe(결과);
});
```

```
describe('findAll', () => {
  it('should return an array of cats', async () => {
    const result = ['test'];
    jest.spyOn(catsService, 'findAll').mockImplementation(() => result);

    기대(await catsController.findAll()).toBe(결과);
```

```
    });
  });
});
```

`Test` 클래스는 기본적으로 전체 Nest 런타임을 모킹하는 애플리케이션 실행 컨텍스트를 제공하는 데 유용하지만, 모킹 및 재정의 등 클래스 인스턴스를 쉽게 관리할 수 있는 툥을 제공합니다. `Test` 클래스에는 모듈 메타데이터 객체(`@Module()` 데코레이터에 전달한 객체와 동일한 객체)를 인자로 받는 `createTestingModule()` 메서드가 있습니다. 이 메서드는 몇 가지 메서드를 제공하는 `TestingModule` 인스턴스를 반환합니다. 단위 테스트의 경우 중요한 메서드는 `compile()` 메서드입니다. 이 메서드는 종속성이 있는 모듈을 부트스트랩하고(기존의 `main.ts` 파일에서 `NestFactory.create()`를 사용하여 애플리케이션을 부트스트랩하는 방식과 유사), 테스트할 준비가 된 모듈을 반환합니다.

정보 힌트 `compile()` 메서드는 비동기적이므로 기다려야 합니다. 모듈이 컴파일되면 `get()` 메서드를 사용하여 모듈이 선언한 정적 인스턴스(컨트롤러 및 프로바이더)를 검색할 수 있습니다.

`TestingModule`은 [모듈 참조](#) 클래스를 상속하므로 범위가 지정된 공급자(일시적이거나 요청 범위가 지정된)를 동적으로 확인할 수 있는 기능이 있습니다. 이 작업은 `resolve()` 메서드를 사용하여 수행합니다(`get()` 메서드는 정적 인스턴스만 검색할 수 있음).

```
const moduleRef = await Test.createTestingModule({
  controllers: [CatsController],
  공급자: [CatsService],
}).compile();

catsService = await moduleRef.resolve(CatsService);
```

경고 `resolve()` 메서드는 자체 DI 컨테이너 하위 트리에서 공급자의 고유한 인스턴스를 반환합니다. 각 하위 트리에는 고유한 컨텍스트 식별자가 있습니다. 따라서 이 메서드를 더 많이 호출하면 를 두 번 이상 클릭하고 인스턴스 참조를 비교하면 동일하지 않다는 것을 알 수 있습니다.

정보 힌트 [여기에서](#) 모듈 참조 기능에 대해 자세히 알아보세요.

프로덕션 버전의 제공업체를 사용하는 대신 테스트 목적으로 [사용자 지정 제공업체로](#) 재정의할 수 있습니다. 예를 들어 라이브 데이터베이스에 연결하는 대신 데이터베이스 서비스를 모의 테스트할 수 있습니다. 다음 섹션에서 오버라이드를 다루겠지만 단위 테스트에도 사용할 수 있습니다.

자동 모킹

Nest를 사용하면 누락된 모든 종속성에 적용할 모의 팩토리를 정의할 수도 있습니다. 이 기능은 클래스에 많은 종속성이 있고 모든 종속성을 모킹하는 데 오랜 시간과 많은 설정이 필요한 경우에 유용합니다. 이 기능을 사용하려면 `createTestingModule()`을 `useMocker()` 메서드와 체인으로 연결하여 의존성 모의에 대한 팩토리를 전달해야 합니다. 이 팩토리는 인스턴스 토큰인 선택적 토큰, 네스트 프로바이더에 유효한 모든 토큰을 받을 수 있으며 모의 구현을 반환합니다. 아래는 `jest-mock`을 사용하여 일반 모의 객체를 생성하는 예시와 `jest.fn()`을 사용하여 `CatsService`에 대한 특정 모의 객체를 생성하는 예시입니다.

```
// ...
import { ModuleMocker, MockFunctionMetadata } from 'jest-mock';

const moduleMocker = new ModuleMocker(global);

describe('CatsController', () => {
  let controller: CatsController;

  beforeEach(async () => {
    const moduleRef = await Test.createTestingModule({
      controllers: [CatsController],
    })
      .useMocker((токен) => {
        const results = ['test1', 'test2'];
        if (token === CatsService) {
          반환 { findAll: jest.fn().mockResolvedValue(results) };
        }
        if (typeof token === 'function') {
          const mockMetadata = moduleMocker.getMetadata(token) as
MockFunctionMetadata<any, any>;
          const Mock = moduleMocker.generateFromMetadata(mockMetadata);
          return new Mock();
        }
      })
      .compile();

    컨트롤러 = moduleRef.get(CatsController);
  });
});
});
```

또한 일반적으로 사용자 지정 공급자와 마찬가지로 테스트 컨테이너에서 이러한 모형을 검색할 수도 있습니다,
`moduleRef.get(CatsService)`.

정보 힌트 `@golevelup/ts-jest`의 `createMock`과 같은 일반적인 모의 팩토리를 직접 전달할 수도 있습니다.
정보 힌트 `REQUEST` 및 `INQUIRER` 공급자는 컨텍스트에서 이미 미리 정의되어 있으므로 자동 모의할 수 없습니다. 그러나 사용자 지정 공급자 구문을 사용하거나 `.overrideProvider` 메서드를 사용하여 덮어쓸 수 있습니다.

엔드투엔드 테스트

개별 모듈과 클래스에 초점을 맞추는 단위 테스트와 달리, 엔드투엔드(e2e) 테스트는 최종 사용자가 프로덕션 시스템과 상호 작용하는 방식에 더 가까운 보다 총체적인 수준에서 클래스와 모듈의 상호 작용을 다룹니다. 애플리케이션이 성장함에 따라 각 API 엔드포인트의 엔드투엔드 동작을 수동으로 테스트하는 것이 어려워집니

다. 자동화된 엔드투엔드 테스트는 시스템의 전반적인 동작이 정확하고 프로젝트 요구 사항을 충족하는지 확인하는 데 도움이 됩니다. e2e 테스트를 수행하기 위해 방금 단위 테스트에서 다룬 것과 유사한 구성을 사용합니다. 또한 Nest를 사용하면 [Supertest](#) 라이브러리를 사용하여 HTTP 요청을 쉽게 시뮬레이션할 수 있습니다.

```
@@파일명(cats.e2e-spec)

'supertest'에서 *를 요청으로 임포트하고,
'@nestjs/testing'에서 { Test }를 임포트합니다;

'../../src/cats/cats.module'에서 { CatsModule }을 임포트하고,
'../../src/cats/cats.service'에서 { CatsService }을 임포트하고,
'@nestjs/common'에서 { INestApplication }을 임포트합니다;

describe('Cats', () => {
  렛 앱: INestApplication;
  let catsService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
      .compile();

    app = moduleRef.createNestApplication();
    await app.init();
  });

  it(`GET /cats`, () => {
    반한 요청(app.getHttpServer())
      .get('/cats')
      .expect(200)
      .expect({
        데이터: catsService.findAll(),
      });
  });

  afterAll(async () => {
    await app.close();
  });
});

@@switch

'supertest'에서 *를 요청으로 임포트하고,
'@nestjs/testing'에서 { Test }를 임포트합니다;

'../../src/cats/cats.module'에서 { CatsModule }을 임포트하고,
'../../src/cats/cats.service'에서 { CatsService }을 임포트하고,
'@nestjs/common'에서 { INestApplication }을 임포트합니다;
```

```
describe('Cats', () => {
  렛 앱: INestApplication;
  let catsService = { findAll: () => ['test'] };

  beforeAll(async () => {
    const moduleRef = await Test.createTestingModule({
      imports: [CatsModule],
    })
      .overrideProvider(CatsService)
      .useValue(catsService)
```

```
.compile();

app = moduleRef.createNestApplication();
await app.init();
});

it(`/GET cats`, () => {
    반한 요청(app.getHttpServer())
        .get('/cats')
        .expect(200)
        .expect({
            데이터: catsService.findAll(),
        });
});

afterAll(async () => {
    await app.close();
});
});
```

정보 힌트 Fastify를 HTTP 어댑터로 사용하는 경우 약간 다른 구성이 필요하며 테스트 기능이 내장되어 있습니다:

```
렛 앱: NestFastifyApplication; beforeAll(async

() => {
    app = moduleRef.createNestApplication<NestFastifyApplication>(new
FastifyAdapter());

    await app.init();
    await app.getHttpAdapter().getInstance().ready();
});

it(`/GET cats`, () => {
    return app
        .inject({
            메서드: 'GET',
            url: '/cats',
        })
        .then((result) => {
            expect(result.statusCode).toEqual(200);
            expect(result.payload).toEqual(/* expectedPayload */);
        });
});

afterAll(async () => {
    await app.close();
});
```

이 예제에서는 앞서 설명한 몇 가지 개념을 기반으로 구축합니다. 앞에서 사용한 `compile()` 메서드에 더해 이제 `createNestApplication()` 메서드를 사용하여 전체 Nest 런타임 환경을 인스턴스화합니다. 실행 중인 앱에 대한 참조를 `app` 변수에 저장하여 HTTP 요청을 시뮬레이션하는 데 사용할 수 있습니다.

Supertest의 `request()` 함수를 사용하여 HTTP 테스트를 시뮬레이션합니다. 이러한 HTTP 요청이 실행 중인 Nest 앱으로 라우팅되기를 원하므로 `요청()` 함수에 Nest의 기반이 되는 HTTP 리스너에 대한 참조를 전달합니다(이 참조는 Express 플랫폼에서 제공될 수 있음). 따라서 `요청(app.getHttpServer())` 구조가 생성됩니다. `request()`를 호출하면 이제 Nest 앱에 연결된 래핑된 HTTP 서버가 전달되며, 이 서버는 실제 HTTP 요청을 시뮬레이션하는 메서드를 노출합니다. 예를 들어, `요청(...).get('/cats')`을 사용하면 네트워크를 통해 들어오는 `get '/cats'`와 같은 실제 HTTP 요청과 동일한 요청이 Nest 앱에 시작됩니다.

이 예제에서는 테스트할 수 있는 하드코딩된 값을 간단히 반환하는 `CatsService`의 대체(테스트-더블) 구현도 제공합니다. 이러한 대체 구현을 제공하려면 `overrideProvider()`를 사용하세요. 마찬가지로 Nest는 모듈, 가드, 인터셉터, 필터, 파이프를 재정의하는 메서드를 각각 `overrideModule()`, `overrideGuard()`, `overrideInterceptor()`, `overrideFilter()`, `overridePipe()` 메서드를 통해 제공합니다.

각 재정의 메서드(`overrideModule()` 제외)는 사용자 정의 공급자에 대해 설명된 메서드를 미러링하는 3개의 서로 다른 메서드가 있는 객체를 반환합니다:

- **사용 클래스**: 객체를 재정의할 인스턴스(공급자, 가드 등)를 제공하기 위해 인스턴스화할 클래스를 제공합니다.
- **사용값**: 객체를 재정의할 인스턴스를 제공합니다.
- **useFactory**: 객체를 재정의할 인스턴스를 반환하는 함수를 제공합니다.

반면에 `overrideModule()`은 다음과 같이 원래 모듈을 재정의할 모듈을 제공하는 데 사용할 수 있는 `useModule()` 메서드가 있는 객체를 반환합니다:

```
const moduleRef = await Test.createTestingModule({
  imports: [AppModule],
})
  .overrideModule(CatsModule)
  .useModule(AlternateCatsModule)
  .compile();
```

각 재정의 메서드 유형은 차례로 `TestingModule` 인스턴스를 반환하므로 유창한 스타일로 다른 메서드와 체인으로 연결할 수 있습니다. 이러한 체인의 끝에서 `compile()`을 사용하여 Nest가 모듈을 인스턴스화하고 초기화하도록 해야 합니다.

또한 테스트가 실행될 때(예: CI 서버에서) 사용자 정의 로거를 제공하고자 하는 경우도 있습니다. `setLogger()` 메서드를 사용하고 `LoggerService` 인터페이스를 충족하는 객체를 전달하여 테스트 중에 테스트 모듈 빌더에 로깅하는 방법을 지시하세요(기본적으로 "오류" 로그만 콘솔에 기록됩니다).

경고 경고 `@nestjs/core` 패키지는 글로벌 인핸서를 정의하는 데 도움이 되는 `APP_ 접두사가 있는 고유한 공급자 토큰을 노출합니다. 이러한 토큰은 다음을 나타낼 수 있으므로 재정의할 수 없습니다.`

여러 공급자를 사용할 수 있습니다. 따라서 `.overrideProvider(APP_GUARD)` 등을 사용할 수 없습니다

. 일부 글로벌 인핸서를 재정의하려면 [이 해결](#) 방법을 따르세요.

컴파일된 모듈에는 다음 표에 설명된 대로 몇 가지 유용한 메서드가 있습니다:

<code>createNestApplication()</code>	주어진 모듈을 기반으로 네스트 애플리케이션(INestApplication 인스턴스)을 생성하고 반환합니다. <code>init()</code> 메서드를 사용하여 애플리케이션을 수동으로 초기화해야 한다는 점에 유의하세요.
<code>createNestMicroservice()</code>	주어진 모듈을 기반으로 Nest 마이크로서비스(INestMicroservice 인스턴스)를 생성하고 반환합니다.
<code>get()</code>	애플리케이션 컨텍스트에서 사용할 수 있는 컨트롤러 또는 공급자(가드, 필터 등 포함)의 정적 인스턴스를 검색합니다. 모듈 참조 클래스에서 상속됩니다.
<code>resolve()</code>	애플리케이션 컨텍스트에서 사용할 수 있는 컨트롤러 또는 공급자(가드, 필터 등 포함)의 동적으로 생성된 범위 인스턴스(요청 또는 일시적)를 검색합니다. 모듈 참조 클래스에서 상속됩니다.
<code>select()</code>	모듈의 종속성 그래프를 탐색하고, 선택한 모듈에서 특정 인스턴스를 검색하는 데 사용할 수 있습니다(<code>get()</code> 메서드에서 엄격 모드 (<code>strict: true</code>)와 함께 사용).

정보 힌트 e2e 테스트 파일은 [테스트](#) 디렉터리 안에 보관하세요. 테스트 파일에는 `.e2e- spec` 접미사가 있어야 합니다.

전 세계적으로 등록된 인핸서 재정의하기

전 세계적으로 등록된 가드(또는 파이프, 인터셉터 또는 필터)가 있는 경우 해당 인핸서를 재정의하려면 몇 가지 단계를 더 수행해야 합니다. 원래 등록을 요약하면 다음과 같습니다:

```
제공자: [
  {
    제공: APP_GUARD, useClass:
      JwtAuthGuard,
  },
],
```

이는 **APP_*** 토큰을 통해 가드를 "멀티" 공급자로 등록하는 것입니다. 이 토큰은

JwtAuthGuard을 등록하려면 이 슬롯에 있는 기존 공급업체를 사용해야 합니다:

```
제공자: [
  {
    제공: APP_GUARD, useExisting:
      JwtAuthGuard,
    // ^^^^^^^^^ 'useClass' 대신 'useExisting'을 사용한 것을 확인할 수 있습니다.
  },
]
```

```
JwtAuthGuard,  
],
```

정보 힌트 Nest가 토큰 뒤에 인스턴스화하는 대신 등록된 공급자를 참조하도록 `useClass`를 `useExisting`으로 변경하세요.

이제 `JwtAuthGuard`는 Nest에 일반 공급자로 표시되며, 이 공급자는 `TestingModule`:

```
const moduleRef = await Test.createTestingModule({  
  imports: [AppModule],  
})  
  .overrideProvider(JwtAuthGuard)  
  .useClass(MockAuthGuard)  
  .compile();
```

이제 모든 테스트에서 모든 요청에 `MockAuthGuard`를 사용합니다. 요청 범위 인스턴스 테스트

요청 범위가 [지정된](#) 공급자는 들어오는 각 요청에 대해 고유하게 생성됩니다. 인스턴스는 요청 처리가 완료된 후 가비지 수집됩니다. 이는 테스트된 요청을 위해 특별히 생성된 의존성 주입 하위 트리에 액세스할 수 없기 때문에 문제가 됩니다.

위의 섹션을 통해 동적으로 인스턴스화된 클래스를 검색하는 데 `resolve()` 메서드를 사용할 수 있다는 것을 알고 있습니다. 또한 [여기에](#) 설명된 대로 고유한 컨텍스트 식별자를 전달하여 DI 컨테이너 하위 트리의 라이프사이클을 제어할 수 있다는 것도 알고 있습니다. 테스트 컨텍스트에서 이를 어떻게 활용할 수 있을까요?

전략은 컨텍스트 식별자를 미리 생성하고 Nest가 이 특정 ID를 사용하여 들어오는 모든 요청에 대한 하위 트리를 생성하도록 하는 것입니다. 이렇게 하면 테스트된 요청에 대해 생성된 인스턴스를 검색할 수 있습니다.

이를 위해 `ContextIdFactory`에서 `jest.spyOn()`을 사용합니다:

```
const contextId = ContextIdFactory.create();  
jest.spyOn(ContextIdFactory, 'getByRequest').mockImplementation(() =>  
  contextId);
```

이제 `contextId`를 사용하여 후속 요청에 대해 생성된 단일 DI 컨테이너 하위 트리에 액세스할 수 있습니다.

```
catsService = await moduleRef.resolve(CatsService, contextId);
```

구성

애플리케이션은 종종 서로 다른 환경에서 실행됩니다. 환경에 따라 다른 구성 설정을 사용해야 합니다. 예를 들어, 일반적으로 로컬 환경에서는 로컬 DB 인스턴스에만 유효한 특정 데이터베이스 자격 증명을 사용합니다. 프로덕션 환경에서는 별도의 DB 자격 증명 집합을 사용합니다. 구성 변수가 변경되므로 환경에 [구성 변수를 저장하는](#) 것이 가장 좋습니다.

외부에서 정의한 환경 변수는 [프로세스.env](#) 글로벌을 통해 Node.js 내부에서 볼 수 있습니다. 환경 변수를 각 환경마다 별도로 설정하여 여러 환경의 문제를 해결할 수 있습니다. 하지만 이는 특히 이러한 값을 쉽게 모킹하거나 변경해야 하는 개발 및 테스트 환경에서는 매우 번거로울 수 있습니다.

Node.js 애플리케이션에서는 각 환경을 나타내기 위해 각 키가 특정 값을 나타내는 키-값 쌍을 포함하는 [.env](#) 파일을 사용하는 것이 일반적입니다. 따라서 서로 다른 환경에서 앱을 실행하려면 올바른 [.env](#) 파일을 교체하기만 하면 됩니다.

Nest에서 이 기술을 사용하기 위한 좋은 접근 방식은 적절한 [.env](#) 파일을 로드하는 [ConfigService](#)를 노출하는 [ConfigModule](#)을 만드는 것입니다. 이러한 모듈을 직접 작성할 수도 있지만, 편의를 위해 Nest는 [@nestjs/config](#) 패키지를 기본으로 제공합니다. 이 패키지는 이번 장에서 다루겠습니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
$ npm i --save @nestjs/config
```

정보 힌트 [@nestjs/config](#) 패키지는 내부적으로 [dotenv](#)를 사용합니다.

경고 [@nestjs/config](#)에는 TypeScript 4.1 이상이 필요합니다.

시작하기

설치 프로세스가 완료되면 [컨피그모듈을](#) 임포트할 수 있습니다. 일반적으로 루트 [AppModule](#)로 임포트하고 [.forRoot\(\)](#) 정적 메서드를 사용하여 동작을 제어합니다. 이 단계에서는 환경 변수 키/값 쌍을 구문 분석하

고 해결합니다. 나중에 다른 기능 모듈에서 구성 모듈의 구성 서비스 클래스에 액세스하기 위한 몇 가지 옵션을 살펴볼 것입니다.

@@파일명(앱.모듈)

'@nestjs/common'에서 { Module }을 가져오고,
'@nestjs/config'에서 { ConfigModule }을 가져옵니다;
다;

모듈({

```
    import { ConfigModule } from '@nestjs/config';
    import { Module } from '@nestjs/common';

    @Module({
      imports: [ConfigModule.forRoot()],
      providers: []
    })
    export class AppModule {}
```

위의 코드는 기본 위치(프로젝트 루트 디렉토리)에서 `.env` 파일을 로드 및 구문 분석하고, `.env` 파일의 키/값 쌍을 `process.env`에 할당된 환경 변수와 병합한 다음, 그 결과를 `ConfigService`를 통해 액세스할 수 있는 비공개 구조에 저장합니다. `forRoot()` 메서드는 구문 분석/병합된 구성 변수를 읽기 위한 `get()` 메서드를 제공하는 `ConfigService` 공급자를 등록합니다. nestjs/config는 `dotenv`에 의존하기 때문에 환경 변수 이름의 충돌을 해결하기 위해 해당 패키지의 규칙을 사용합니다. 키가 런타임 환경에 환경 변수로 존재할 때(예: `export DATABASE_USER=test`와 같은 OS 셸 내보내기를 통해) 및 `.env` 파일에 모두 존재할 때 런타임 환경 변수가 우선합니다.

샘플 `.env` 파일은 다음과 같습니다:

```
DATABASE_USER=test  
DATABASE_PASSWORD=test
```

사용자 지정 환경 파일 경로

기본적으로 패키지는 애플리케이션의 루트 디렉터리에서 `.env` 파일을 찾습니다. `.env` 파일의 다른 경로를 지정하려면 다음과 같이 `forRoot()`에 전달하는 (선택적) 옵션 객체의 `envFilePath` 속성을 설정합니다:

```
ConfigModule.forRoot({  
  envFilePath: '.development.env',  
});
```

다음과 같이 `.env` 파일에 대해 여러 경로를 지정할 수도 있습니다:

```
ConfigModule.forRoot({  
  envFilePath: ['.env.development.local', '.env.development'],  
});
```

변수가 여러 파일에서 발견되면 첫 번째 파일이 우선합니다. 환경 변수로

딩 비활성화

`.env` 파일을 로드하지 않고 대신 다음에서 환경 변수에 간단히 액세스하려는 경우 런타임 환경(예: `DATABASE_USER=test` 내보내기와 같은 OS 셸 내보내기와 마찬가지로)에서 다음과 같이 옵션

개체의 `무시EnvFile` 속성을 `true`로 설정합니다:

```
ConfigModule.forRoot({  
  무시환경파일: true,  
});
```

모듈을 전 세계적으로 사용

다른 모듈에서 컨피그모듈을 사용하려면 모든 Nest 모듈의 표준처럼 컨피그모듈을 임포트해야 합니다. 또는 아래와 같이 옵션 객체의 `isGlobal` 속성을 `true`로 설정하여 [전역 모듈로](#) 선언할 수도 있습니다. 이 경우 루트 모듈(예: `AppModule`)에 로드된 후에는 다른 모듈에서 `ConfigModule`을 임포트할 필요가 없습니다.

```
ConfigModule.forRoot({
  isGlobal: true,
});
```

사용자 지정 구성 파일

더 복잡한 프로젝트의 경우 사용자 정의 설정 파일을 활용하여 중첩된 설정 객체를 반환할 수 있습니다. 이를 통해 관련 구성 설정을 기능별로 그룹화하고(예: 데이터베이스 관련 설정), 관련 설정을 개별 파일에 저장하여 독립적으로 관리할 수 있습니다.

사용자 정의 구성 파일은 구성 객체를 반환하는 팩토리 함수를 내보냅니다. 구성 객체는 임의로 중첩된 일반 JavaScript 객체일 수 있습니다. `process.env` 객체에는 완전히 해결된 환경 변수 키/값 쌍이 포함됩니다([위에서](#) 설명한 대로 `.env` 파일과 외부 정의 변수가 해결되고 병합됨). 반환된 구성 객체를 제어하므로 필요한 로직을 추가하여 값을 적절한 유형으로 캐스팅하고 기본값을 설정하는 등의 작업을 수행할 수 있습니다. 예를 들어

```
@@파일명(구성/설정) 내보내기 기본값 ()
=> ({
  port: parseInt(process.env.PORT, 10) || 3000,
  database: {
    호스트: process.env.DATABASE_HOST,
    port: parseInt(process.env.DATABASE_PORT, 10) || 5432
  }
});
```

`ConfigModule.forRoot()`에 전달하는 옵션 객체의 `load` 속성을 사용하여 이 파일을 로드합니다. 메서드를 사용합니다:

```
'./config/configuration'에서 구성 가져 오기; @Module({  
    임포트합니다: [  
        ConfigModule.forRoot({  
            로드합니다: [구성],  
            } ),  
        ],  
    })  
내보내기 클래스 AppModule {}
```

정보 로드 프로퍼티에 할당된 값은 배열이므로 여러 구성 파일을 로드할 수 있습니다(예: `[load: [databaseConfig, authConfig]]`).

사용자 지정 구성 파일을 사용하면 YAML 파일과 같은 사용자 지정 파일도 관리할 수 있습니다. 다음은 YAML 형식을 사용한 구성의 예입니다:

```
http:
  호스트: 'localhost'
  포트: 8080

db:
  postgres:
    url: 'localhost' 포
    트: 5432 데이터베이스:
    'yaml-db'

sqlite:
  데이터베이스: 'sqlite.db'
```

YAML 파일을 읽고 구문 분석하기 위해 `js-yaml` 패키지를 활용할 수 있습니다.

```
$ npm i js-yaml
$ npm i -D @타입스/js-yaml
```

패키지가 설치되면 `yaml#load` 함수를 사용하여 위에서 방금 만든 YAML 파일을 로드합니다.

```
@@파일이름(config/configuration)
import { readFileSync } from 'fs';
import * as yaml from 'js-yaml';
import { join } from 'path';

const YAML_CONFIG_FILENAME = 'config.yaml';

export default () => {
  반화 yaml.load(
    경고Nest CLI는 일드 프로세스 중에 "자산"(TS가 아닌 파일)을 dist 폴더로 자동으로 이동하지 않습니다.
    readFileSync(join(__dirname, YAML_CONFIG_FILENAME), 'utf8'),
  );
}

이를 지정해야 합니다. 예를 들어 config 폴더가 src 폴더와 같은 수준인 경우
compilerOptions#assets에 다음 값을 추가합니다.

"assets": [{ "include": "../config/*.yaml", "outDir": "./dist/config"}]. 여기에서 자세히 알아보세요.
```


컨피그서비스 사용

컨피그 서비스에서 구성 값에 액세스하려면 먼저 컨피그 서비스를 주입해야 합니다. 다른 프로바이더와 마찬가지로, 해당 프로바이더를 포함하는 모듈인 ConfigModule을 이를 사용할 모듈로 임포트해야 합니다 (`ConfigModule.forRoot()` 메서드에 전달된 옵션 객체의 `isGlobal` 속성을 `true`로 설정하지 않는 한). 아래와 같이 가능 모듈로 임포트합니다.

```
@@파일명(feature.module) @Module({
  imports: [구성 모듈],
  // ...
})
```

그런 다음 표준 생성자 주입을 사용하여 주입할 수 있습니다:

```
constructor(private configService: ConfigService) {}
```

정보 힌트 컨피그서비스는 `@nestjs/config` 패키지에서 가져옵니다.

그리고 수업에서 사용하세요:

```
// 환경 변수 가져오기
const dbUser = this.configService.get<string>('DATABASE_USER');

// 사용자 지정 구성 값 가져오기
const dbHost = this.configService.get<string>('database.host');
```

위와 같이 `configService.get()` 메서드를 사용하여 변수 이름을 전달하여 간단한 환경 변수를 가져옵니다. 위와 같이 유형을 전달하여 TypeScript 유형 힌트를 수행할 수 있습니다(예: `get<string>(...)`). 위의 두 번째 예에서와 같이 `get()` 메서드는 중첩된 사용자 지정 구성 객체([사용자 지정 구성 파일을 통해 생성됨](#))를 순회할 수도 있습니다.

인터페이스를 유형 힌트로 사용하여 중첩된 전체 사용자 지정 구성 객체를 가져올 수도 있습니다:

```
인터페이스 DatabaseConfig { 호
    스트: 문자열;
    포트: 숫자;
}

const dbConfig = this.configService.get<DatabaseConfig>('database');

// 이제 `dbConfig.port`와 `dbConfig.host`를 사용할 수 있습
니다;
```

또한 `get()` 메서드는 아래와 같이 키가 존재하지 않을 때 반환되는 기본값을 정의하는 선택적 두 번째 인수를 받습니다:

```
// "database.host"가 정의되지 않은 경우 "localhost" 사용
const dbHost = this.configService.get<string>('database.host',
  'localhost');
```

`ConfigService`에는 두 개의 선택적 제네릭(유형 인수)이 있습니다. 첫 번째는 존재하지 않는 구성 속성에 액세스하는 것을 방지하는 데 도움이 됩니다. 아래 그림과 같이 사용합니다:

```
인터페이스 EnvironmentVariables {
  PORT: 숫자;
  TIMEOUT: 문자열입니다;
}

// 코드 어딘가에
constructor(private configService: ConfigService<EnvironmentVariables>) {
  const port = this.configService.get('PORT', { infer: true });

  // 타입스크립트 오류: URL 속성이 환경변수에 정의되어 있지 않으므로 유효하지 않습니다.
  const url = this.configService.get('URL', { infer: true });
}
```

`infer` 속성을 `true`로 설정하면 `ConfigService#get` 메서드가 인터페이스를 기반으로 속성 유형을 자동으로 추론하므로, 예를 들어 `PORT`는 `EnvironmentVariables` 인터페이스에서 `숫자` 유형을 가지므로 `typeof port === "숫자"`(TypeScript에서 `strictNullChecks` 플래그를 사용하지 않는 경우)가 됩니다.

또한 `유추` 기능을 사용하면 점 표기법을 사용하는 경우에도 다음과 같이 중첩된 사용자 지정 구성 객체의 속성 유형을 유추할 수 있습니다:

```
constructor(private configService: ConfigService<{ database: { host:
  string } }>) {
  const dbHost = this.configService.get('database.host', { infer: true
})!;

  // dbHost 유형 === "문자열"
  //
  +--> 널이 아닌 어설션 연산자
}
```

두 번째 제네릭은 첫 번째 제네릭에 의존하여 정의되지 않은 모든 유형을 제거하는 유형 어설션으로 작동합니다.

ConfigService의 메서드는 strictNullChecks가 켜져 있을 때 반환할 수 있습니다. 예를 들어

```
// ...
생성자(private configService: ConfigService<{ PORT: number }, true>)
{
  //
  const port = this.configService.get('PORT', { infer: true });
  //^^^ 포트 유형이 '숫자'가 되므로 더 이상 TS 유형 어설션이 필요하지 않습니다.
}
```

구성 네임스페이스

위의 [사용자 지정 구성 파일](#)에 표시된 대로 여러 사용자 지정 구성 파일을 정의하고 로드할 수 있습니다. 해당 섹션에 표시된 것처럼 중첩된 구성 객체를 사용하여 복잡한 구성 객체 계층 구조를 관리할 수 있습니다. 또는 다음과 같이 `registerAs()` 함수를 사용하여 "네임스페이스" 구성 객체를 반환할 수 있습니다:

```
@@파일명(config/database.config)
export default registerAs('database', () => ({
  host: process.env.DATABASE_HOST,
  포트: process.env.DATABASE_PORT || 5432
}));
```

사용자 지정 설정 파일과 마찬가지로, `registerAs()` 팩토리 함수 내부의 `process.env` 객체에는 완전히 해결된 환경 변수 키/값 쌍이 포함됩니다(위에서 설명한 대로 `.env` 파일과 외부 정의 변수가 해결되고 병합된 상태).

정보 힌트 `registerAs` 함수는 [@nestjs/config](#) 패키지에서 내보냅니다.

사용자 지정 구성 파일을 로드하는 것과 같은 방식으로 `forRoot()` 메서드의 옵션 객체의 `load` 속성을 사용하여 네임스페이스 구성을 로드합니다:

```
'./config/database.config'에서 databaseConfig 가져오기;

@Module({
  임포트합니다: [
    ConfigModule.forRoot({
      로드합니다: [데이터베이스 구성],
    }),
  ],
})
내보내기 클래스 AppModule {}
```

이제 데이터베이스 네임스페이스에서 호스트 값을 가져오려면 점 표기법을 사용합니다. 네임스페이스 이름에 해당하는 속성 이름의 접두사로 'database'를 사용합니다(`registerAs()` 함수의 첫 번째 인수로 전달됨):

```
const dbHost = this.configService.get<string>('database.host');
```

합리적인 대안은 데이터베이스 네임스페이스를 직접 삽입하는 것입니다. 이렇게 하면 강력한 타이핑의 이점을 누릴 수 있습니다:

```
생성자( @Inject(databaseConfig.KEY)
  비공개 dbConfig: 구성 유형<데이터베이스 구성 유형>,
) {}
```

정보 힌트 구성 유형은 `@nestjs/config` 패키지에서 내보냅니다.

캐시 환경 변수

`process.env`에 액세스하는 속도가 느려질 수 있으므로, `ConfigModule.forRoot()`에 전달되는 옵션 객체의 캐시 속성을 설정하여 `process.env`에 저장된 변수에 대한 `ConfigService#get` 메서드의 성능을 향상시킬 수 있습니다.

```
ConfigModule.forRoot({ 캐시:
  true,
});
```

부분 등록

지금까지 루트 모듈(예: `AppModule`)에서 `forRoot()` 메서드를 사용하여 구성 파일을 처리했습니다. 기능별 구성 파일이 여러 다른 디렉터리에 있는 더 복잡한 프로젝트 구조를 가지고 있을 수도 있습니다. 이러한 모든 파일을 루트 모듈에 로드하는 대신 `@nestjs/config` 패키지는 각 기능 모듈과 관련된 구성 파일만 참조하는 부분 등록이라는 기능을 제공합니다. 이 부분 등록을 수행하려면 다음과 같이 기능 모듈 내에서 `forFeature()` 정적 메서드를 사용합니다:

'./config/database.config'에서 `databaseConfig` 가져오기;
 정보 경고 일부 상황에서는 생성자가 아닌 `onModuleInit()` 훅을 사용하여 부분 등록을 통해 로드된 `@Module({})` 프로퍼티에 액세스해야 할 수도 있습니다. 이는 모듈 초기화 중에 `forFeature()` 메서드가 실행되고 모듈 초기화 순서가 불확실하기 때문입니다. 다른 모듈에서 이러한 방식으로 로드한 값에 접근하는 경우, 데이터베이스 모듈 클래스 {} 내보내기
 생성자에서 모듈의

구성이 의존하는 모듈이 아직 초기화되지 않았을 수 있습니다. `onModuleInit()` 메서드는 종속된 모든

모듈이 초기화된 후에만 실행되므로 이 기법은 안전합니다.

스키마 유효성 검사

필수 환경 변수가 제공되지 않았거나 특정 유효성 검사 규칙을 충족하지 않는 경우 애플리케이션을 시작하는 동안 예외를 발생시키는 것이 표준 관행입니다. `nestjs/config` 패키지를 사용하면 두 가지 방법으로 이를 수행할 수 있습니다:

- `Joi` 내장 유효성 검사기. `Joi`를 사용하면 객체 스키마를 정의하고 이에 대해 JavaScript 객체의 유효성을 검사할 수 있습니다.
- 환경 변수를 입력으로 받는 사용자 정의 유효성 검사([\(\)](#)) 함수.

`Joi`를 사용하려면 `Joi` 패키지를 설치해야 합니다:

```
$ npm install --save joi
```

이제 `Joi` 유효성 검사 스키마를 정의하고 유효성 검사 스키마를

`forRoot()` 메서드의 옵션 객체를 아래와 같이 설정합니다:

`@@파일명`(`app.module`) 'joi'에서

*를 `Joi`로 가져옵니다;

```
모듈({ import: [
  ConfigModule.forRoot({
    validationSchema: Joi.object({
      NODE_ENV: Joi.string()
        .valid('개발', '프로덕션', '테스트', '프로비저닝')
        .default('development'),
      PORT: Joi.number().default(3000),
    }),
  ]),
}],  
}  
내보내기 클래스 AppModule {}
```

기본적으로 모든 스키마 키는 선택 사항으로 간주됩니다. 여기서는 환경(`.env` 파일 또는 프로세스 환경)에서 이러한 변수를 제공하지 않을 경우 사용되는 `NODE_ENV` 및 `PORT`에 대한 기본값을 설정합니다. 또는 `required()`

유효성 검사 메서드를 사용하여 환경(`.env` 파일 또는 프로세스 환경)에 값이 정의되어 있어야 함을 요구할 수 있습니다. 이 경우 유효성 검사 단계는 환경에 변수를 제공하지 않으면 예외를 발생시킵니다. 유효성 검사 스키마를 구성하는 방법에 대한 자세한 내용은 [Joi 유효성 검사 메서드를 참조하세요.](#)

기본적으로 알 수 없는 환경 변수(스키마에 키가 없는 환경 변수)는 허용되며 유효성 검사 예외를 트리거하지 않습니다. 기본적으로 모든 유효성 검사 오류가 보고됩니다. `forRoot()` 옵션 객체의 `validationOptions` 키를 통해 옵션 객체를 전달하여 이러한 동작을 변경할 수 있습니다. 이 옵션 객체에는 다음과 같은 표준 유효성 검사 옵션이 포함될 수 있습니다.

프로퍼티를 변경할 수 있습니다. 예를 들어 위의 두 설정을 반전시키려면 다음과 같은 옵션을 전달합니다:

```
@@파일명(app.module) 'joi'에서
*를 Joi로 가져옵니다;

모듈({ import: [
  ConfigModule.forRoot({
    validationSchema: Joi.object({
      NODE_ENV: Joi.string()
        .valid('개발', '프로덕션', '테스트', '프로비저닝')
        .default('development'),
      PORT: Joi.number().default(3000),
    }),
    유효성 검사 옵션: {
      allowUnknown: false,
      abortEarly: true,
    },
  }),
],
})

내보내기 클래스 AppModule {}
```

`nestjs/config` 패키지는 다음과 같은 기본 설정을 사용합니다:

- `allowUnknown`: 환경 변수에 알 수 없는 키를 허용할지 여부를 제어합니다. 기본값은 `true`입니다.
- `abortEarly`: 참이면 첫 번째 오류에서 유효성 검사를 중지하고, 거짓이면 모든 오류를 반환합니다. 기본값은 `false`입니다.

유효성 검사 옵션 객체를 전달하기로 결정한 후에는 명시적으로 전달하지 않은 모든 설정은 `@nestjs/config` 기본값이 아닌 `Joi` 표준 기본값으로 기본값이 설정된다는 점에 유의하세요. 예를 들어, 사용자 지정 유효성 검사 옵션 객체에서 `allowUnknowns`를 지정하지 않은 상태로 두면 `Joi` 기본값이 `false`로 설정됩니다. 따라서 사용자 지정 객체에서 이 두 가지 설정을 모두 지정하는 것이 가장 안전합니다.

사용자 지정 유효성 검사 기능

또는 환경 변수가 포함된 객체(환경 파일 및 프로세스에서)를 가져와서 필요한 경우 변환/변환할 수 있도록 유효성이 검사된 환경 변수가 포함된 객체를 반환하는 동기식 유효성 검사 함수를 지정할 수 있습니다. 이 함수가 오류를 발생시키면 애플리케이션이 부트스트랩되지 않습니다.

이 예제에서는 `클래스 트랜스포머`와 `클래스 유효성 검사기` 패키지로 진행하겠습니다. 먼저 정의해야 합니다:

- 유효성 검사 제약 조건이 있는 클래스입니다,
- `plainToInstance` 및 `validateSync` 함수를 사용하는 유효성 검사 함수입니다.

@@파일명(환경 유효성 검사)

'class-transformer'에서 { plainToInstance }를 가져옵니다;
 'class-validator'에서 { IsEnum, IsNumber, validateSync }를 가져옵니다;

```
열거형 환경 {
  개발 = "개발", 프로덕션 = "프로덕션",
  테스트 = "테스트",
  프로비저닝 = "프로비저닝",
}
```

환경 변수 클래스 { @IsEnum(환경)}

```
NODE_ENV: 환경;

IsNumber()
PORT: 숫자;
}

export 함수 validate(config: Record<string, unknown>) { const
  validatedConfig = plainToInstance(
    환경 변수, config,
    { enableImplicitConversion: true },
  );
  const errors = validateSync(validatedConfig, { skipMissingProperties: false });

  if (errors.length > 0) {
    새로운 오류(errors.toString())를 던집니다;
  }
  유효성 검사된 컨피그를 반환합니다;
}
```

이 설정이 완료되면 다음과 같이 유효성 검사 함수를 ConfigModule의 구성 옵션으로 사용합니다:

@@파일명(앱.모듈)

'./env.validation'에서 { validate }를 가져옵니다;

```
모듈({ import: [
  ConfigModule.forRoot({
    validate,
  }),
],
})

내보내기 클래스 AppModule {}
```

사용자 지정 게터 함수

ConfigService는 키별로 구성 값을 검색하는 일반 `get()` 메서드를 정의합니다. 좀 더 자연스러운 코딩 스타일을 구현하기 위해 `getter` 함수를 추가할 수도 있습니다:

```
@@파일명()
@Injectable()
내보내기 클래스 ApiConfigService {
    constructor(private configService: ConfigService) {}

    get isEnabled(): boolean {
        return this.configService.get('AUTH_ENABLED') === 'true';
    }
}

@@스위치 @종속성(컨피규레이션서비스) @인젝터를()
export class ApiConfigService {
    constructor(configService) {
        this.config서비스 = config서비스;
    }

    get isEnabled() {
        return this.configService.get('AUTH_ENABLED') === 'true';
    }
}
```

이제 다음과 같이 게터 함수를 사용할 수 있습니다:

```
@@파일명(app.service) @Injectable()
export class AppService {
    constructor(apiConfigService: ApiConfigService) {
        if (apiConfigService.isEnabled) {
            // 인증이 활성화되었습니다.
        }
    }
}

@@스위치 @종속성(ApiConfigService)
@Injectable()
export class AppService {
    constructor(apiConfigService) {
        if (apiConfigService.isEnabled) {
            // 인증이 활성화되었습니다.
        }
    }
}
```

환경 변수 로드 후크

모듈 구성이 환경 변수에 의존하고 이러한 변수는

`.env` 파일과 상호 작용하기 전에 파일이 로드되었는지 확인하기 위해 `ConfigModule.envVariablesLoaded` 흑을 사용할 수 있습니다(다음 예제 참조):

```
export async function getStorageModule() {
  await ConfigModule.envVariablesLoaded;
  반환 process.env.STORAGE === 'S3' ? S3StorageModule : 디폴트스토리지모듈;
}
```

이 구조는 `ConfigModule.envVariablesLoaded` 프로미스가 해결된 후 모든 구성 변수가 로드되도록 보장합니다.

확장 가능한 변수

`nestjs/config` 패키지는 환경 변수 확장을 지원합니다. 이 기술을 사용하면 한 변수가 다른 변수의 정의 내에서 참조되는 중첩 환경 변수를 만들 수 있습니다. 예를 들어

APP_URL=mywebsite.com 지원 이메일
=support@\${APP_URL}

이 구성을 사용하면 변수 SUPPORT_EMAIL이 'support@mywebsite.com'로 해석됩니다 '{ }...{{ }}' }]} 구문을 사용하여 SUPPORT_EMAIL 정의 내에서 변수 APP_URL의 값 확인을 트리거합니다.

정보 힌트 이 기능의 경우 `@nestis/config` 패키지는 내부적으로 `dotenv-expand`를 사용합니다.

아래 그림과 같이 `ConfigModule`의 `forRoot()` 메서드에 전달된 옵션 객체의 `expandVariables` 속성을 사용하여 환경 변수 확장을 활성화합니다:

```
@@파일명(app.module)
@Module({
    임포트합니다: [
        ConfigModule.forRoot({
            // ... 확장변수: true,
        }),
    ],
})
```

main.ts에서 사용

구성은 서비스에 저장되어 있지만 `메인.ts` 파일에서도 사용할 수 있습니다. 이렇게 하면 애플리케이션 포트나 CORS 호스트와 같은 변수를 저장하는 데 사용할 수 있습니다.

액세스하려면 `app.get()` 메서드 뒤에 서비스 참조를 사용해야 합니다:

```
const configService = app.get(ConfigService);
```

그런 다음 구성 키로 `get` 메서드를 호출하여 평소처럼 사용할 수 있습니다:

```
const port = configService.get('PORT');
```

데이터베이스

Nest는 데이터베이스에 구애받지 않으므로 모든 SQL 또는 NoSQL 데이터베이스와 쉽게 통합할 수 있습니다. 선호도에 따라 다양한 옵션을 사용할 수 있습니다. 가장 일반적인 수준에서 Nest를 데이터베이스에 연결하려면 Express 또는 Fastify를 사용할 때와 마찬가지로 데이터베이스에 적합한 Node.js 드라이버를 로드하기만 하면 됩니다.

또한, 더 높은 수준의 추상화에서 작동하기 위해 MikroORM(MikroORM 레시피 참조), Sequelize(Sequelize 통합 참조), Knex.js(Knex.js 튜토리얼 참조), TypeORM, Prisma(Prisma 레시피 참조) 등 범용 Node.js 데이터베이스 통합 라이브러리 또는 ORM을 직접 사용할 수도 있습니다.

편의를 위해 Nest는 이번 챕터에서 다룰 `@nestjs/typeorm` 및 `@nestjs/sequelize` 패키지를 통해 TypeORM 및 Sequelize와 즉시 사용할 수 있는 긴밀한 통합 기능을 제공하며, 이 챕터에서 다룰 `@nestjs/mongoose` 패키지를 통해 Mongoose와도 통합할 수 있습니다. 이러한 통합은 모델/리포지토리 주입, 테스트 가능성, 비동기 구성과 같은 추가적인 NestJS 전용 기능을 제공하여 선택한 데이터베이스에 더욱 쉽게 액세스할 수 있도록 해줍니다.

TypeORM 통합

Nest는 SQL 및 NoSQL 데이터베이스와의 통합을 위해 `@nestjs/typeorm` 패키지를 제공합니다. TypeORM은 TypeScript에서 사용할 수 있는 가장 성숙한 객체 관계형 매퍼(ORM)입니다. TypeScript로 작성되었기 때문에 Nest 프레임워크와 잘 통합됩니다.

사용을 시작하려면 먼저 필요한 종속성을 설치합니다. 이 장에서는 널리 사용되는 MySQL 관계형 DBMS를 사용하는 데모를 보여드리지만, TypeORM은 PostgreSQL, Oracle, Microsoft SQL Server, SQLite, 심지어 MongoDB와 같은 NoSQL 데이터베이스와 같은 많은 관계형 데이터베이스를 지원합니다. 이 장에서 안내하는 절차는 TypeORM에서 지원하는 모든 데이터베이스에 대해 동일합니다. 선택한 데이터베이스에 대한 관련 클라이언트 API 라이브러리를 설치하기만 하면 됩니다.

```
npm install --save @nestjs/typeorm typeorm mysql2
```

설치 프로세스가 완료되면 `TypeOrmModule`을 루트 앱모듈로 가져올 수 있습니다.

@@파일명(앱.모듈)

'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져옵니다;

```
모듈({ import: [
  TypeOrmModule.forRoot({
    type: 'mysql',
    호스트: 'localhost',
    포트: 3306, 사용자 이
    름: 'root', 비밀번호:
    'root', 데이터베이스:
    'test', entities:
  [] ,
```

```
        동기화: true,  
    } ),  
],  
}  
  
내보내기 클래스 AppModule {}
```

경고 **동기화** 설정: true는 프로덕션 환경에서 사용해서는 안 됩니다. 그렇지 않으면 프로덕션 데이터가 손실될 수 있습니다.

`forRoot()` 메서드는 TypeORM 패키지의 `DataSource` 생성자에 의해 노출되는 모든 구성 속성을 지원합니다. 또한 아래에 설명된 몇 가지 추가 구성 속성이 있습니다.

retryAttempts 데이터베이스에 연결하려는 시도 횟수(기본값: 10)

retryDelay 연결 재시도 시도 사이의 지연 시간(ms)(기본값: 3000)

autoLoadEntities 참이면 엔티티가 자동으로 로드됩니다(기본값: false).

정보 힌트 [여기에서](#) 데이터 원본 옵션에 대해 자세히 알아보세요.

이 작업이 완료되면 예를 들어 모듈을 임포트할 필요 없이 전체 프로젝트에 TypeORM `DataSource` 및 `EntityManager` 객체를 삽입할 수 있습니다:

@@파일명(앱.모듈)

'typeorm'에서 { DataSource }를 가져옵니다;

모듈({

 임포트: [TypeOrmModule.forRoot(), UsersModule],
 })

 내보내기 클래스 AppModule {
 constructor(private dataSource: DataSource) {}
 }

@switch

'typeorm'에서 { DataSource }를 가져옵니다;

 @Dependencies(DataSource)

 @Module({
 임포트: [TypeOrmModule.forRoot(), UsersModule],
 })

 export class AppModule {
 constructor(dataSource) {
 이 데이터 소스 = 데이터 소스;
 }
 }

리포지토리 패턴

TypeORM은 리포지토리 디자인 패턴을 지원하므로 각 엔티티에는 자체 리포지토리가 있습니다. 이러한 리포지토리는 데이터베이스 데이터 소스에서 가져올 수 있습니다.

예제를 계속하려면 엔티티가 하나 이상 필요합니다. [User](#) 엔티티를 정의해 보겠습니다.

@@파일명(사용자.엔티티)

'[typeorm](#)'에서 { Entity, Column, PrimaryGeneratedColumn }을 가져옵니다;

엔티티()

```
export class User {  
  @PrimaryGeneratedColumn()  
  id: number;
```

Column()

```
  firstName: string;
```

@Column() 성: 문자

열;

@Column({ 기본값: true })

정보 힌트 [TypeORM 문서](#)에서 엔티티에 대해 자세히 알아보세요.

사용자 엔티티 파일은 사용자 디렉터리에 있습니다. 이 디렉터리에는 [UsersModule](#)과 관련된 모든 파일이 들어 있습니다. 모델 파일을 어디에 보관할지 결정할 수 있지만 도메인 근처의 해당 모듈 디렉터리에 만드는 것이 좋습니다.

사용자 엔티티를 사용하려면 엔티티에 삽입하여 TypeORM에 해당 엔티티에 대해 알려야 합니다.

배열을 사용할 수 있습니다(정적 글로브 경로를 사용하지 않는 경우):

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,  
'./users/user.entity'에서 { User }를 가져옵니다;
```

```
모듈({ import: [  
    TypeOrmModule.forRoot({  
        type: 'mysql',  
        호스트: 'localhost',  
        포트: 3306, 사용자 이  
        름: 'root', 비밀번호:  
        'root', 데이터베이스:  
        'test', entities:  
        [ 사용자 ], 동기화:  
        true,  
    }),  
    ],  
})  
내보내기 클래스 AppModule {}
```

다음으로 `UsersModule`을 살펴보겠습니다:

```
@@파일명(users.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule } 가져오기,
'./users.service'에서 { UsersService } 가져오기,
'./users.controller'에서 { UsersController } 가져오기,
'./user.entity'에서 { User } 가져오기;

모듈({
  임포트합니다: [TypeOrmModule.forFeature([User])],
  providers: [UsersService],
  컨트롤러: [UsersController],
})
사용자 모듈 클래스 {} 내보내기
```

이 모듈은 `forFeature()` 메서드를 사용하여 현재 스코프에 등록된 리포지토리를 정의합니다. 이렇게 정의하면 `@InjectRepository()` 데코레이터를 사용하여 `UsersRepository`를 `UsersService`에 삽입할 수 있습니다:

```
@@파일명(users.service)
'@nestjs/common'에서 { Injectable } 임포트;
'@nestjs/typeorm'에서 { InjectRepository } 임포트;
'typeorm'에서 { Repository } 임포트;
'./user.entity'에서 { User }를 가져옵니다;

@Injectable()
내보내기 클래스 UsersService { 생성자(
    @InjectRepository(사용자)
    비공개 사용자리포지토리: 리포지토리<사용자>,
) {}

findAll(): Promise<User[]> {
    this.usersRepository.find()를 반환합니다;
}

findOne(id: 숫자): Promise<사용자 | null> { return
    this.usersRepository.findOneBy({ id });
}

async remove(id: 숫자): Promise<void> { await
    this.usersRepository.delete(id);
}
}

@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 가져오고,
'@nestjs/typeorm'에서 { getRepositoryToken }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
```

```

주입 가능()
@DependenciesgetRepositoryToken(User)
export class UsersService {
  constructor(usersRepository) {
    this.usersRepository = usersRepository;
  }

  findAll() {
    this.usersRepository.find()를 반환합니다;
  }

  findOne(id) {
    이.usersRepository.findOneBy({ id })를 반환합니다;
  }

  async remove(id) {
    await this.usersRepository.delete(id);
  }
}

```

경고 `UsersModule`을 루트 앱모듈로 임포트하는 것을 잊지 마세요.

`TypeOrmModule.forFeature`을 가져오는 모듈 외부에서 리포지토리를 사용하려면 해당 모듈에서 생성된 프로바이더를 다시 내보내야 합니다. 다음과 같이 전체 모듈을 내보내면 됩니다:

```

@@파일명(users.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;

```

```

모듈({
  임포트합니다:
  [TypeOrmModule.forFeature([User])], 내보내기:
  [TypeOrmModule]
})
사용자 모듈 클래스 {} 내보내기

```

이제 `UserHttpModule`에서 `UsersModule`을 가져오면 후자의 모듈의 공급자에 `@InjectRepository(User)`를 사용할 수 있습니다.

@@파일명(users-http.module)

```
'@nestjs/common'에서 { Module } 가져오기;  
'./users.module'에서 { UsersModule } 가져오기;  
'./users.service'에서 { UsersService } 가져오기;  
'./users.controller'에서 { UsersController }를 가져옵니다;
```

모듈({

```
  임포트: [UsersModule], 공급자:  
  [UsersService],
```

```
컨트롤러: [UsersController]  
}  
내보내기 클래스 UserHttpModule {}
```

관계

관계는 둘 이상의 테이블 간에 설정된 연결입니다. 관계는 각 테이블의 공통 필드를 기반으로 하며, 종종 기본 키와 외래 키가 포함됩니다.

관계에는 세 가지 유형이 있습니다:

일대일	주 테이블의 모든 행에는 외래 테이블에 연결된 행이 하나만 있습니다. 이 유형의 관계를 정의하려면 @OneToOne() 데코레이터를 사용합니다.
일대다 / 다대일	주 테이블의 모든 행에는 외래 테이블에 하나 이상의 관련 행이 있습니다. 주 테이블에서 이 유형의 관계를 정의하는 데 @OneToMany() 및 @ManyToOne() 데코레이터를 사용할 수
다대다	있습니다.

주 테이블의 모든 행은 외래 테이블에 많은 관련 행이 있고, 외래 테이블의 모든 레코드는 주 테이블에 많은 관련 행이 있습니다. 이러한 유형의 관계를 정의하려면 [@ManyToMany\(\)](#) 데코레이터를 사용합니다.

엔티티에서 관계를 정의하려면 해당 데코레이터를 사용합니다. 예를 들어 각 [사용자](#)는 여러 장의 사진을 가질 수 있으므로 [@OneToMany\(\)](#) 데코레이터를 사용합니다.

@@파일명 (사용자.엔티티)

'typeorm'에서 { Entity, Column, PrimaryGeneratedColumn, OneToMany }를 가져옵니다;
'.../사진/사진.엔티티'에서 { 사진 }을 가져옵니다;

엔티티()

```
export class User {  
  @PrimaryGeneratedColumn()  
  id: number;
```

Column()

```
  firstName: 문자열;
```

@Column() 성: 문자

열;

@Column({ 기본값: true })

```
  isActive: boolean;
```

원투다수(유형 => 사진, 사진 => 사진.사용자) 사진:

```
  Photo[];
```

```
}
```

정보 힌트 TypeORM의 관계에 대해 자세히 알아보려면 [TypeORM 문서를](#) 참조하세요.

엔티티 자동 로드

데이터 소스 옵션의 [엔티티](#) 배열에 엔티티를 수동으로 추가하는 작업은 번거로울 수 있습니다. 또한 루트 모듈에서 엔티티를 참조하면 애플리케이션 도메인 경계가 무너지고 애플리케이션의 다른 부분으로 구현 세부 정보가 유출될 수 있습니다. 이 문제를 해결하기 위해 대체 솔루션이 제공됩니다. 엔티티를 자동으로 로드하려면 아래 그림과 같이 구성 객체([forRoot\(\)](#) 메서드에 전달됨)의 [autoLoadEntities](#) 속성을 [true](#)로 설정합니다:

[@@파일명\(앱.모듈\)](#)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져옵니다;
```

```
모듈({ import: [  
    TypeOrmModule.forRoot({  
        ...  
        autoLoadEntities: true,  
    }),  
],  
})  
내보내기 클래스 AppModule {}
```

이 옵션을 지정하면 [forFeature\(\)](#) 메서드를 통해 등록된 모든 엔티티가 구성 객체의 [엔티티](#) 배열에 자동으로 추가됩니다.

경고 [forFeature\(\)](#) 메서드를 통해 등록되지 않고 (관계를 통해서만) 엔티티에서 참조되는 엔티티는 [자동로드 엔티티 설정을](#) 통해 포함되지 않는다는 점에 유의하세요.

엔티티 정의 분리

데코레이터를 사용하여 모델에서 바로 엔티티와 해당 열을 정의할 수 있습니다. 그러나 일부 사람들은 '[엔티티 스키마](#)'를 사용하여 별도의 파일 내에 엔티티와 해당 열을 정의하는 것을 선호합니다.

```
'typeorm'에서 { EntitySchema } 가져오기;
'./user.entity'에서 { User } 가져오기;

export const UserSchema = new EntitySchema<User>({
  name: 'User',
  대상: 사용자, 열: {
    id: {
      유형입니다: 숫자, 기본:
      true, 생성됨: true,
    },
    firstName: {
```

```

        유형: 문자열,
    },
    성: { type: 문자
        열,
    },
    isActive: {
        type: 부울, 기
        본값: true,
    },
},
관계: { photos: {
        유형: '일대다',
        대상: '사진', // 사진 스키마의 이름
    },
},
});

```

경고 오류 경고 대상 옵션을 제공하는 경우 이름 옵션 값은 대상 클래스의 이름과 동일해야 합니다. 대상을 제공하지 않으면 아무 이름이나 사용할 수 있습니다.

예를 들어 엔티티가 필요한 곳이면 어디에서나 Nest를 사용하여 엔티티 스키마 인스턴스를 사용할 수 있습니다:

```

'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,
'./user.schema'에서 { UserSchema }를 가져옵니다;
'./users.controller'에서 { UsersController }를 가져오고,
'./users.service'에서 { UserService }를 가져옵니다;

모듈({
    임포트합니다: [TypeOrmModule.forFeature([UserSchema])],
    제공자: [UserService],
    컨트롤러: [UsersController],
})
사용자 모듈 클래스 {} 내보내기

```

유형ORM 트랜잭션

데이터베이스 트랜잭션은 데이터베이스 관리 시스템 내에서 데이터베이스에 대해 수행되는 작업 단위를 의미하며, 다른 트랜잭션과 독립적으로 일관성 있고 신뢰할 수 있는 방식으로 처리됩니다. 트랜잭션은 일반적으로 데이터베

이스의 모든 변경 사항을 나타냅니다([자세히 알아보기](#)).

TypeORM 트랜잭션을 처리하는 데는 여러 가지 전략이 있습니다. 저희는 트랜잭션을 완전히 제어할 수 있기 때문입니다.

먼저, 일반적인 방법으로 데이터 소스 객체를 클래스에 주입해야 합니다:

```
@Injectable()  
사용자 서비스 클래스 내보내기 {
```

```
constructor(private dataSource: DataSource) {}
```

정보 힌트 `DataSource` 클래스는 `typeorm` 패키지에서 가져옵니다.

이제 이 객체를 사용하여 트랜잭션을 생성할 수 있습니다.

```
async createMany(users: User[]) {
  const queryRunner = this.dataSource.createQueryRunner();

  await queryRunner.connect();
  await queryRunner.startTransaction();
  try {
    await queryRunner.manager.save(users[0]);
    await queryRunner.manager.save(users[1]);

    await queryRunner.commitTransaction();
  } catch (err) {
    // 오류가 발생했으므로 변경 사항을 롤백하겠습니다 await
    queryRunner.rollbackTransaction();
  } finally {
    // 수동으로 인스턴스화된 쿼리 러너를 릴리스해야 합니다 await
    queryRunner.release();
  }
}
```

정보 힌트 데이터소스는 쿼리러너를 생성하는 데에만 사용된다는 점에 유의하세요. 그러나 이 클래스를 테스트하려면 여러 메서드를 노출하는 전체 `DataSource` 객체를 모킹해야 합니다. 따라서 헬퍼 팩토리 클래스(예: `QueryRunnerFactory`)를 사용하고 트랜잭션을 유지하는 데 필요한 제한된 메서드 집합으로 인터페이스를 정의하는 것이 좋습니다. 이 기법을 사용하면 이러한 메서드를 매우 간단하게 모킹할 수 있습니다.

또는 데이터 소스의 트랜잭션 메서드와 함께 콜백 스타일 접근 방식을 사용할 수 있습니다.

객체 ([자세히 읽기](#)).

```
async createMany(users: User[]) {
  await this.dataSource.transaction(async manager => {
    await manager.save(users[0]);
    await manager.save(users[1]);
  });
}
```

구독자

TypeORM [구독자를](#) 사용하면 특정 엔티티 이벤트를 수신할 수 있습니다.

```
가져 오기 {
  DataSource,
  엔티티 구독자 인터페이스, 이벤트
  구독자, 삽입 이벤트,
}를 'typeorm'에서 가져옵니다;
'./user.entity'에서 { User }를 가져옵니다;
```

이벤트 구독자()

```
export class UserSubscriber 구현 EntitySubscriberInterface<User> {
  constructor(dataSource: DataSource) {
    dataSource.subscribers.push(this);
  }

  listenTo() {
    return User;
  }

  beforeInsert(event: InsertEvent<사용자>) { console.log(`

  사용자가 삽입되기 전: `, event.entity);
  }
}
```

오류 경고 이벤트 구독자를 요청 범위로 지정할 수 없습니다.

이제 공급자 배열에 `UserSubscriber` 클래스를 추가합니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./users.controller'에서 { UsersController } 가져오기,
'./users.service'에서 { UsersService } 가져오기,
'./user.subscriber'에서 { UserSubscriber } 가져오기;
```

모듈({

임포트합니다: [TypeOrmModule.forFeature([User])], 제공자

: [UsersService, UserSubscriber], 컨트롤러:

정보 [한트법 인 구독자에 대한 자세한 내용은 여기를 참조하세요.](#)

})

사용자 모듈 클래스 {} 내보내기

マイグレーション

マイグレーション은 데이터베이스의 기존 데이터를 보존하면서 데이터베이스 스키마를 점진적으로 업데이트하여 애플리케이션의 데이터 모델과 동기화 상태를 유지할 수 있는 방법을 제공합니다. 마이그레이션을 생성, 실행 및 되돌리기 위해 TypeORM은 전용 [CLI](#)를 제공합니다.

マイグ레이션 클래스는 Nest アプリケーション ソース コードとは別個입니다. マイグレーション クラスの ライフサイクルは TypeORM CLI によって維持されます。したがって、マイグレーションを介して一貫性を保つことや、Nest に関する機能を活用するには、マイグレーションについて詳しく学ぶ必要があります。マイグレーションについて詳しく学ぶには、[TypeORM ドキュメントのガイド](#)を参照してください。

여러 데이터ベース

一部のプロジェクトでは複数のデータベース接続が必要になります。このモジュールを使用してこの作業を実行できます。複数の接続を作成するには、まず接続を定義します。この場合、データベース名を指定する必要があります。

자체データベースに格納された `Album` エンティティがあると仮定します。

```
const defaultOptions = {
  type: 'postgres',
  port: 5432,
  username: 'user',
  password: 'password',
  database: 'db',
  synchronize: true,
};

@Module({
  imports: [
    TypeOrmModule.forRoot({
      ...defaultOptions,
      host: 'user_db_host',
      entities: [User],
    }),
    TypeOrmModule.forRoot({
      ...basicOptions,
      name: 'albumsConnection',
      host: 'album_db_host',
      entities: [Album],
    }),
  ],
})
```

警告: `TypeOrmModule.forRoot()` の `name` を指定しない場合は、接続名が「`albumsConnection`」と設定されます。接続名が複数ある場合は、注意してください。

警告: `TypeOrmModule.forRootAsync()` を使用する場合、`useFactory` 外部でデータベース名を設定する必要があります。例を示すと次のようになります。

```
TypeOrmModule.forRootAsync({  
  name: 'albumsConnection',  
  useFactory: ...,
```

```
    주입: . . . ,  
}),
```

자세한 내용은 이번 호를 참조하세요.

이 시점에서 사용자 및 앨범 엔티티가 자체 데이터 소스에 등록되어 있습니다. 이 설정을 사용하면 `TypeOrmModule.forFeature()` 메서드와 `@InjectRepository()` 데코레이터에 어떤 데이터 소스를 사용해야 하는지 알려주어야 합니다. 데이터 소스 이름을 전달하지 않으면 기본 데이터 소스가 사용됩니다.

```
모듈({ import: [  
    TypeOrmModule.forFeature([User]),  
    TypeOrmModule.forFeature([Album], 'albumsConnection'),  
],  
})  
내보내기 클래스 AppModule {}
```

지정된 데이터 소스에 대한 `DataSource` 또는 `EntityManager`을 삽입할 수도 있습니다:

```
@Injectable()  
내보내기 클래스 앨범 서비스 { 생성자(  
    @InjectDataSource('albumsConnection') 비공개  
    데이터 소스: DataSource,  
    @InjectEntityManager('albumsConnection')  
    private entityManager: EntityManager,  
) {}  
}
```

데이터소스를 공급자에게 주입할 수도 있습니다:

```
모듈({ providers:  
[  
  {  
    제공 앨범 서비스,  
    useFactory: (albumsConnection: DataSource) => {  
      반환 새 앨범 서비스(albumsConnection);  
    },  
    주입합니다: [getDataSourceToken('albumsConnection')],  
  },  
],  
})
```

내보내기 클래스 앨범 모듈 {}

테스트

애플리케이션을 단위 테스트할 때 일반적으로 데이터베이스 연결을 피하고 테스트 스위트를 독립적으로 유지하며 실행 프로세스를 가능한 한 빠르게 유지하려고 합니다. 하지만 클래스가 데이터 소스(연결) 인스턴스에서 가져온 리포지토리에 의존할 수 있습니다. 이를 어떻게 처리할까요? 해결책은 모의 리포지토리를 만드는 것입니다. 이를 위해 사용자 지정 공급자를 설정합니다. 등록된 각 리포지토리는 자동으로 <EntityName> 리포지토리 토큰으로 표시되며, 여기서 EntityName은 엔티티 클래스의 이름입니다.

nestjs/typeorm 패키지는 주어진 엔티티를 기반으로 준비된 토큰을 반환하는 `getRepositoryToken()` 함수를 노출합니다.

```
모듈({ providers:
  [
    사용자 서비스,
    {
      제공: getRepositoryToken(User), 사용값: 모
      의 리포지토리,
    },
  ],
})
사용자 모듈 클래스 {} 내보내기
```

이제 대체 `mockRepository`가 `UsersRepository`로 사용됩니다. 어떤 클래스에서 `@InjectRepository()` 데코레이터를 사용하여 `UsersRepository`를 요청할 때마다 Nest는 등록된 `mockRepository` 객체를 사용합니다.

비동기 구성

리포지토리 모듈 옵션을 정적이 아닌 비동기적으로 전달하고 싶을 수도 있습니다. 이 경우 비동기 구성을 처리하는 여러 가지 방법을 제공하는 `forRootAsync()` 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
TypeOrmModule.forRootAsync({
  useFactory: () => ({
    유형: 'mysql', 호스
    트: 'localhost',
    포트: 3306, 사용자명
    : 'root', 비밀번호:
    'root', 데이터베이스
    : 'test',
    entities: [],
    synchronize: true,
  }),
});
```

저희 팩토리는 다른 [비동기 공급자처럼](#) 동작합니다(예: [비동기일](#) 수 있고, [주입을](#) 통해 종속성을 주입할 수 있습니다).

```
TypeOrmModule.forRootAsync({
  import: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    type: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    사용자           이름:
    configService.get('USERNAME'),   비밀번호:
    configService.get('PASSWORD'),   데이터베이
    스:           configService.get('DATABASE'),
    entities: [],
    동기화: true,
  }),
  주입합니다: [구성 서비스],
});
```

또는 [useClass](#) 구문을 사용할 수도 있습니다:

```
TypeOrmModule.forRootAsync({
  useClass: TypeOrmConfigService,
});
```

위의 구조는 [TypeOrmModule](#) 내부에 [TypeOrmConfigService](#)를 인스턴스화하고 이를 사용하여 [createTypeOrmOptions\(\)](#)를 호출하여 옵션 객체를 제공합니다. 이는 아래 그림과 같이 [TypeOrmConfigService](#)가 [TypeOrmOptionsFactory](#) 인터페이스를 구현해야 한다는 것을 의미합니다:

```
@Injectable()
export 클래스 TypeOrmConfigService 구현 TypeOrmOptionsFactory {
    createTypeOrmOptions(): TypeOrmModuleOptions {
        반환 {
            유형: 'mysql', 호스트
            : 'localhost', 포트
            : 3306, 사용자명:
            'root', 비밀번호:
            'root', 데이터베이스:
            'test', entities:
            [], synchronize:
            true,
        };
    }
}
```

TypeOrmModule 내부에 TypeOrmConfigService를 생성하지 않고 다른 모듈에서 가져온 프로바이더를 사용하려면 useExisting 구문을 사용할 수 있습니다.

```
TypeOrmModule.forRootAsync({  
    import: [ConfigModule],  
    useExisting: ConfigService,  
});
```

이 구조는 [사용클래스](#)와 동일하게 작동하지만 한 가지 중요한 차이점이 있습니다. TypeOrmModule은 가져온 모듈을 조회하여 새 [구성 서비스](#)를 인스턴스화하는 대신 기존 [구성 서비스](#)를 재사용합니다.

정보 힌트 이름 프로퍼티가 [사용팩토리](#), [사용클래스](#) 또는 [사용값](#) 프로퍼티와 동일한 수준에서 정의되었는지 확인하세요. 이렇게 하면 Nest가 적절한 주입 토큰 아래에 데이터 소스를 올바르게 등록할 수 있습니다.

맞춤형 데이터 소스 팩토리

[사용Factory](#), [사용Class](#) 또는 [사용Existing](#)을 사용하는 비동기 구성과 함께, 선택적으로 [데이터소스팩토리](#) 함수를 지정하여 TypeOrmModule이 데이터 소스를 생성하도록 허용하는 대신 자체 TypeORM 데이터 소스를 제공할 수 있도록 할 수 있습니다.

`dataSourceFactory`는 [useFactory](#), [useClass](#) 또는 [useExisting](#)을 사용하여 비동기 구성 중에 구성된 TypeORM 데이터 소스 옵션을 수신하고 TypeORM 데이터 소스를 확인하는 Promise를 반환합니다.

```
TypeOrmModule.forRootAsync({
  import: [ConfigModule],
  inject: [ConfigService],
  // 사용Factory, 사용Class 또는 사용Existing 사용
  //를 사용하여 데이터 소스 옵션을 구성합니다. 사용 팩토리:
  (config서비스: 구성 서비스) => ({
    유형: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    사용자           이름:
    configService.get('USERNAME'),   비밀번호:
    configService.get('PASSWORD'),   데이터베이
    스:           configService.get('DATABASE'),
    entities: [],
    동기화: true,
  }),
  // 데이터소스는 구성된 데이터소스옵션을 수신합니다.
  //를 호출하고 약속<데이터소스>를 반환합니다.

  dataSourceFactory: async (옵션) => {
    정보 한도 dataSource클래스는 await new ormconfig(옵션).initialize();
    return dataSource;
  },
});
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

시퀄라이즈 통합

TypeORM을 사용하는 대신 [@nestjs/sequelize](#) 패키지와 함께 [Sequelize](#) ORM을 사용하는 방법도 있습니다.

또한 엔티티를 선언적으로 정의하기 위해 추가 데코레이터 세트를 제공하는 [sequelize-typescript](#) 패키지를 활용합니다.

사용을 시작하려면 먼저 필요한 종속성을 설치합니다. 이 장에서는 널리 사용되는 [MySQL](#) 관계형 DBMS를 사용하는 데모를 보여드리지만, Sequelize는 PostgreSQL, MySQL, Microsoft SQL Server, SQLite 및 MariaDB와 같은 많은 관계형 데이터베이스를 지원합니다. 이 장에서 안내하는 절차는 Sequelize에서 지원하는 모든 데이터베이스에 대해 동일합니다. 선택한 데이터베이스에 대한 관련 클라이언트 API 라이브러리를 설치하기만 하면 됩니다.

```
$ npm install --save @nestjs/sequelize 시퀄라이즈 시퀄라이즈 --typescript  
mysql2  
$ npm install --save-dev @types/sequelize
```

설치 프로세스가 완료되면 [SequelizeModule](#)을 루트 앱 모듈로 가져올 수 있습니다.

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/sequelize'에서 { SequelizeModule }을 가져옵니다;
```

```
모듈({ import: [  
    SequelizeModule.forRoot({  
        dialect: 'mysql',  
        호스트: 'localhost',  
        포트: 3306, 사용자 이  
        름: 'root', 비밀번호:  
        'root', 데이터베이스:  
        'test', models:  
        [],  
    }),  
],  
})
```

```
내보내기 클래스 AppModule {}
```

`forRoot()` 메서드는 Sequelize 생성자에 의해 노출되는 모든 구성 속성을 지원합니다([자세히 보기](#)). 또한 아래에 설명된 몇 가지 추가 구성 프로퍼티가 있습니다.

<code>retryAttempts</code>	데이터베이스 연결 시도 횟수(기본값: 10)	<code>retryDelay</code>	연결 재시도 시도 사이의 지연 시간(ms)(기본값: 3000)	<code>autoLoadModels</code>	참이면 모델이 자동으로 로드됩니다(기본값: 거짓).

`keepConnectionAlive` 참이면 애플리케이션 종료 시 연결이 닫히지 않습니다(기본값):

거짓)

동기화 true인 경우 자동으로 로드된 모델이 동기화됩니다(기본값: true).

이 작업이 완료되면 예를 들어 모듈을 임포트할 필요 없이 전체 프로젝트에 Sequelize 오브젝트를 삽입할 수 있습니다:

@@파일명(앱.서비스)

```
'@nestjs/common'에서 { Injectable }을 임포트하고,  
'sequelize-typescript'에서 { Sequelize }를 임포트합  
니다;
```

@Injectable()

```
내보내기 클래스 AppService {  
    constructor(private sequelize: Sequelize) {}  
}  
@@switch  
'@nestjs/common'에서 { Injectable }을 임포트하고,  
'sequelize-typescript'에서 { Sequelize }를 임포트합  
니다;
```

종속성(시퀄라이즈) @인젝터블()

```
export class AppService {  
    constructor(sequelize) {  
        this.sequelize = 시퀄라이즈;  
    }  
}
```

모델

Sequelize는 활성 레코드 패턴을 구현합니다. 이 패턴을 사용하면 모델 클래스를 직접 사용하여 데이터베이스와 상호 작용합니다. 예제를 계속하려면 적어도 하나의 모델이 필요합니다. 사용자 모델을 정의해 보겠습니다.

@@파일명(사용자.모델)

'sequelize-typescript'에서 { Column, Model, Table }을 가져옵니다;

테이블

내보내기 클래스 User extends Model {

@Column

이름: 문자열입니다;

칼럼

성: 문자열입니다;

@Column({ defaultValue: true })

isActive: boolean;

}

정보 힌트 [여기에서](#) 사용 가능한 데코레이터에 대해 자세히 알아보세요.

사용자 모델 파일은 사용자 디렉터리에 있습니다. 이 디렉터리에는 `UsersModule`과 관련된 모든 파일이 들어 있습니다. 모델 파일을 어디에 보관할지 결정할 수 있지만 도메인 근처의 해당 모듈 디렉터리에 만드는 것이 좋습니다.

User 모델을 사용하려면 모듈의 `forRoot()` 메서드 옵션에 있는 모델 배열에 삽입하여 Sequelize에 해당 모델을 알려야 합니다:

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/sequelize'에서 { SequelizeModule }을 임포트하고,  
'./users/user.model'에서 { User }를 임포트합니다;
```

```
모듈({ import: [  
    SequelizeModule.forRoot({  
        dialect: 'mysql',  
        호스트: 'localhost',  
        포트: 3306, 사용자 이  
        름: 'root', 비밀번호:  
        'root', 데이터베이스:  
        'test', models: [사  
        용자],  
    }),  
    ],  
})  
내보내기 클래스 AppModule {}
```

다음으로 `UsersModule`을 살펴보겠습니다:

@@파일명 (users.module)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/sequelize'에서 { SequelizeModule }을 임포트하고,  
'./user.model'에서 { User }를 임포트합니다;  
'./users.controller'에서 { UsersController }를 가져오고,  
'./users.service'에서 { UserService }를 가져옵니다;
```

모듈({

```
    임포트합니다: [SequelizeModule.forFeature([User])],  
    providers: [UserService],  
    컨트롤러: [UsersController],  
})
```

사용자 모듈 클래스 {} 내보내기

이 모듈은 `forFeature()` 메서드를 사용하여 현재 스코프에 등록된 모델을 정의합니다. 이를 통해

`@InjectModel()` 데코레이터를 사용하여 사용자 모델을 사용자서비스에 삽입할 수 있습니다:

```
@@파일명(users.service)
'@nestjs/common'에서 { Injectable }을 임포트하고,
'@nestjs/sequelize'에서 { InjectModel }을 임포트하고,
'./user.model'에서 { User }를 임포트합니다;

@Injectable()
내보내기 클래스 UserService { 생성자(
    @InjectModel(사용자)
    비공개 사용자 모델: 사용자 유형,
) {}

비동기 findAll(): Promise<User[]> {
    return this.userModel.findAll();
}

findOne(id: 문자열): Promise<User> {
    return this.userModel.findOne({
        where: {
            id,
        },
    });
}

async remove(id: 문자열): Promise<void> {
    const user = await this.findOne(id);
    await user.destroy();
}
}

@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 임포트하고,
'@nestjs/sequelize'에서 { getModelToken }을 임포트합니다;
'./user.model'에서 { User }를 가져옵니다;
```

주입 가능()

```
@Dependencies(getModelToken(User)) 내보
내기 클래스 UserService {
    constructor(usersRepository) {
        this.usersRepository = usersRepository;
    }

    async findAll() {
        이.사용자모델.모두 찾기()를 반환합니다;
    }
}
```

```
findOne(id) {  
    return this.usermodel.findOne({  
        where: {  
            id,  
        },  
    });  
}
```

```
async remove(id) {
  const user = await this.findOne(id);
  await user.destroy();
}
}
```

경고 `UsersModule`을 루트 앱모듈로 임포트하는 것을 잊지 마세요.

`SequelizeModule.forFeature`를 임포트하는 모듈 외부에서 리포지토리를 사용하려면 모듈에서 생성된 프로바이더를 다시 내보내야 합니다. 다음과 같이 전체 모듈을 내보내면 됩니다:

```
@@파일명(users.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/sequelize'에서 { SequelizeModule }을 가져오고,
'./user.entity'에서 { User }을 가져옵니다;

모듈({
  임포트합니다: [SequelizeModule.forFeature([User])], 내보
  내기: [SequelizeModule]
})
사용자 모듈 클래스 {} 내보내기
```

이제 `UserHttpModule`에서 `UsersModule`을 가져오면, 후자의 모듈의 프로바이더에서 `@InjectModel(User)`를 사용할 수 있습니다.

```
@@파일명(users-http.module)
'@nestjs/common'에서 { Module } 가져오기;
'./users.module'에서 { UsersModule } 가져오기;
'./users.service'에서 { UsersController } 가져오기;
'./users.controller'에서 { UsersController }를 가져옵니다;

모듈({
  임포트합니다: [UsersModule], 공급
  자: [UsersService], 컨트롤러:
  [UsersController]
})
내보내기 클래스 UserHttpModule {}
```

관계

관계는 둘 이상의 테이블 간에 설정된 연결입니다. 관계는 각 테이블의 공통 필드를 기반으로 하며, 종종 기본 키와 외래 키가 포함됩니다.

관계에는 세 가지 유형이 있습니다:

일대일

기본 테이블의 모든 행에는 외래 테이블에 연결된 행이 하나만 있습니다.

테이블

일대다

/ 다대일

주 테이블의 모든 행에는 외래 테이블에 하나 이상의 관련 행이 있습니다.

다대다

주 테이블의 모든 행은 외래 테이블에 많은 관련 행이 있고, 외래 테이블의 모든 레코드는 주 테이블에 많은 관련 행이 있습니다.

모델에서 관계를 정의하려면 해당 데코레이터를 사용합니다. 예를 들어 각 사용자

는 여러 장의 사진을 가질 수 있으므로 `@HasMany()` 데코레이터를 사용합니다.

`@@파일명`(사용자.모델)

```
'sequelize-typescript'에서 { Column, Model, Table, HasMany } 가져오기; './
사진/사진.모델'에서 { Photo } 가져오기;
```

테이블

내보내기 클래스 `User extends Model {`

`@Column`

이름: 문자열입니다;

`칼럼`

성: 문자열입니다;

`@Column({ defaultValue: true })`
`isActive: boolean;`

`HasMany(() => 사진) 사진:`

정부 확인서 쿼리에서 연결에 대해 자세히 알아보려면 [이](#)장을 읽어보세요.

}

자동 로드 모델

연결 옵션의 `모델` 배열에 모델을 수동으로 추가하는 작업은 번거로울 수 있습니다. 또한 루트 모듈에서 모델을 참

조하면 애플리케이션 도메인 경계가 깨지고 애플리케이션의 다른 부분으로 구현 세부 정보가 유출될 수 있습니다.

이 문제를 해결하려면 아래 그림과 같이 `autoLoadModels`를 모두 설정하여 모델을 자동으로 로드하고 구성 객체의 `동기화` 속성(`forRoot()` 메서드에 전달)을 `true`로 설정하세요:

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/sequelize'에서 { SequelizeModule }을 가져옵니다;
```

```
모듈({ import: [  
    SequelizeModule.forRoot({  
        ...  
    })  
] })
```

```
    autoLoadModels: true,
    synchronize: true,
  }),
],
}

내보내기 클래스 AppModule {}
```

이 옵션을 지정하면 `forFeature()` 메서드를 통해 등록된 모든 모델이 구성 객체의 `모델` 배열에 자동으로 추가됩니다.

경고 `forFeature()` 메서드를 통해 등록되지 않고 모델에서 (연관을 통해서만) 참조되는 모델은 포함되지 않습니다.

거래 시퀄라이즈

데이터베이스 트랜잭션은 데이터베이스 관리 시스템 내에서 데이터베이스에 대해 수행되는 작업 단위를 의미하며, 다른 트랜잭션과 독립적으로 일관성 있고 신뢰할 수 있는 방식으로 처리됩니다. 트랜잭션은 일반적으로 데이터베이스의 모든 변경 사항을 나타냅니다([자세히 알아보기](#)).

[시퀄라이즈 트랜잭션을](#) 처리하는 전략에는 여러 가지가 있습니다. 다음은 관리되는 트랜잭션(자동 콜백)의 샘플 구현입니다.

먼저 일반적인 방법으로 `Sequelize` 객체를 클래스에 주입해야 합니다:

```
@Injectable()
export class UsersService {
  constructor(private sequelize: Sequelize) {}
}
```

정보 힌트 `시퀄라이즈` 클래스는 [시퀄라이즈-타입스크립트](#) 패키지에서 가져옵니다.

이제 이 객체를 사용하여 트랜잭션을 생성할 수 있습니다.

```
async createMany() {
  try {
    await this.sequelize.transaction(async t => {
      const transactionHost = { 트랜잭션: t };

      await this.usermodel.create(
        { firstName: '아브라함', lastName: '링컨' }, 트랜잭션호스트
        ,
      );
      await this.usermodel.create(
        { firstName: 'John', lastName: '부스' }, 트랜잭션호스트,
      );
    });
  } catch (err) {
    // 트랜잭션이 롤백되었습니다.
  }
}
```

```
// 오류는 트랜잭션 콜백에 반환된 프로미스 체인이 거부한 모든 것입니다.  
}  
}
```

정보 힌트 [Sequelize](#) 인스턴스는 트랜잭션을 시작할 때만 사용된다는 점에 유의하세요. 그러나 이 클래스를 테스트하려면 여러 메서드를 노출하는 전체 [Sequelize](#) 객체를 모킹해야 합니다. 따라서 헬퍼 팩토리 클래스(예: [TransactionRunner](#))를 사용하고 트랜잭션을 유지하는 데 필요한 제한된 메서드 집합으로 인터페이스를 정의하는 것이 좋습니다. 이 기법을 사용하면 이러한 메서드를 매우 간단하게 모킹 할 수 있습니다.

マイグ레이션

マイグ레이션은 데이터베이스의 기존 데이터를 보존하면서 데이터베이스 스키마를 점진적으로 업데이트하여 애플리케이션의 데이터 모델과 동기화 상태를 유지할 수 있는 방법을 제공합니다. 마이그레이션을 생성, 실행 및 되돌리기 위해 Sequelize는 전용 [CLI](#)를 제공합니다.

マイグ레이션 클래스는 Nest 애플리케이션 소스 코드와는 별개입니다. 마이그레이션 클래스의 수명 주기는 Sequelize CLI에 의해 유지 관리됩니다. 따라서 마이그레이션을 통해 종속성 주입 및 기타 Nest 특정 기능을 활용 할 수 없습니다. 마이그레이션에 대해 자세히 알아보려면 [Sequelize 설명서의 가이드](#)를 참조하세요.

여러 데이터베이스

일부 프로젝트에는 여러 데이터베이스 연결이 필요합니다. 이 모듈을 사용하면 이 작업도 수행할 수 있습니다. 여러 연결로 작업하려면 먼저 연결을 만듭니다. 이 경우 연결 이름 지정은 필수입니다.

자체 데이터베이스에 저장된 [앨범](#) 엔티티가 있다고 가정해 보겠습니다.

```
const defaultOptions = {
  dialect: 'postgres',
  port: 5432,
  사용자 이름: '사용자', 비밀번호
  : '비밀번호', 데이터베이스:
  'db', synchronize: true,
};

모듈({ import: [
  SequelizeModule.forRoot({
    ...defaultOptions,
    host: 'user_db_host',
    models: [사용자],
  }),
  SequelizeModule.forRoot({
    ...기본옵션,
    name: 'albumsConnection',
    host: 'album_db_host',
    models: [앨범],
  })
]})
```

```
    } ) ,  
    ] ,  
}  
  
내보내기 클래스 AppModule {}
```

경고 연결의 이름을 설정하지 않으면 해당 이름이 기본값으로 설정됩니다. 이름이 없거나 같은 이름을 가진 연결이 여러 개 있으면 재정의되므로 주의하세요.

이 시점에서 사용자 및 앨범 모델이 자체 연결로 등록되어 있습니다. 이 설정을 사용하면 SequelizeModule.forFeature() 메서드와 @InjectModel() 데코레이터에 어떤 연결을 사용해야 하는지 알려줘야 합니다. 연결 이름을 전달하지 않으면 기본 연결이 사용됩니다.

```
모듈({ import: [  
    SequelizeModule.forFeature([User]),  
    SequelizeModule.forFeature([Album], 'albumsConnection'),  
],  
})  
  
내보내기 클래스 AppModule {}
```

지정된 연결에 대해 Sequelize 인스턴스를 삽입할 수도 있습니다:

```
@Injectable()  
내보내기 클래스 앨범 서비스 { 생성자(  
    주입 연결('albumsConnection') 비공개 시퀄라이즈:  
        시퀄라이즈,  
    ) {}  
}
```

공급자에게 시퀄라이즈 인스턴스를 주입할 수도 있습니다:

```
모듈({  
  providers: [  
    {  
      제공 앨범 서비스,  
      useFactory: (albumsSequelize: Sequelize) => { 반환  
        새 앨범 서비스(albumsSequelize);  
      },  
      주입합니다: [getDataSourceToken('albumsConnection')],  
    },  
  ],  
})
```

내보내기 클래스 앨범 모듈 {}

테스트

애플리케이션을 단위 테스트할 때 일반적으로 데이터베이스 연결을 피하고 테스트 스위트를 독립적으로 유지하며 실행 프로세스를 가능한 한 빠르게 유지하려고 합니다. 하지만 클래스가 연결 인스턴스에서 가져온 모델에 의존할 수 있습니다. 이를 어떻게 처리할까요? 해결책은 모의 모델을 만드는 것입니다. 이를 위해 [사용자 지정 공급자를 설정합니다](#). 등록된 각 모델은 자동으로 <모델명>모델 토큰으로 표시되며, 여기서 모델명은 모델 클래스의 이름입니다.

`nestjs/sequelize` 패키지는 주어진 모델을 기반으로 준비된 토큰을 반환하는 `getModelToken()` 함수를 노출합니다.

```
모듈({ providers:
  [
    사용자 서비스,
    {
      제공: getModelToken(User), 사용값:
      mockModel,
    },
  ],
})
사용자 모듈 클래스 {} 내보내기
```

이제 대체 `mockModel`이 `UserModel`로 사용됩니다. 어떤 클래스가 `UserModel`을 요청할 때마다 [인젝트 모델\(\)](#) 데코레이터를 사용하면 Nest는 등록된 `mockModel` 객체를 사용합니다. 비

동기 구성

정적이 아닌 비동기적으로 `SequelizeModule` 옵션을 전달하고 싶을 수도 있습니다. 이 경우 비동기 구성을 처리하는 여러 가지 방법을 제공하는 `forRootAsync()` 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
SequelizeModule.forRootAsync({
  useFactory: () => ({
    방언을 사용합니다:
    'mysql', host:
    'localhost', port:
    3306, 사용자명:
    'root', 비밀번호:
    'root', 데이터베이스
    : 'test', models:
    [],
  }),
});
```

저희 팩토리는 다른 [비동기](#) 공급자와 마찬가지로 작동합니다(예: 비동기일 수 있고 [주입을](#) 통해 종속성을 주입할 수 있습니다).

```
SequelizeModule.forRootAsync({
  import: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    dialect: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    models: [],
  }),
  주입합니다: [구성 서비스],
});
```

또는 `useClass` 구문을 사용할 수도 있습니다:

```
SequelizeModule.forRootAsync({
  useClass: SequelizeConfigService,
});
```

위의 구조는 `SequelizeModule` 내부에 `SequelizeConfigService`를 인스턴스화하고 이를 사용하여 `createSequelizeOptions()`를 호출하여 옵션 객체를 제공합니다. 이는 아래 그림과 같이 `SequelizeConfigService`가 `SequelizeOptionsFactory` 인터페이스를 구현해야 한다는 것을 의미합니다:

```
@Injectable()
SequelizeConfigService 클래스는 SequelizeOptionsFactory {
  createSequelizeOptions() 를 구현합니다: SequelizeModuleOptions {
    반환 {
      방언을 사용합니다:
      'mysql', host:
      'localhost', port:
      3306, 사용자명:
      'root', 비밀번호:
      'root', 데이터베이스:
      'test', models:
      [],
    };
  }
}
```

`SequelizeModule` 내부에 `SequelizeConfigService`를 생성하지 않고 다른 모듈에서 가져온 프로바이더를 사용하려면 `useExisting` 구문을 사용하면 됩니다.

```
SequelizeModule.forRootAsync({  
  import: [ConfigModule],
```

```
    사용Existing: ConfigService,  
});
```

이 구조는 `useClass`와 동일하게 작동하지만 한 가지 중요한 차이점이 있습니다. `SequelizeModule`은 가져온 모듈을 조회하여 새 `구성 서비스`를 인스턴스화하는 대신 기존 `구성 서비스`를 재사용합니다.

예

작동 예제는 [여기에서](#) 확인할 수 있습니다.

몽고

Nest는 두 가지 방법으로 MongoDB 데이터베이스와 통합할 수 있습니다. 여기에 설명된 내장 TypeORM 모듈을 사용하거나, MongoDB용 커넥터가 있는 내장 TypeORM 모듈을 사용하거나, 가장 널리 사용되는 MongoDB 객체 모델링 도구인 Mongoose를 사용할 수 있습니다. 이 장에서는 전용 `@nestjs/mongoose` 패키지를 사용하여 후자를 설명하겠습니다.

필요한 종속성을 설치하는 것으로 시작하세요:

```
npm i @nestjs/mongoose 몽구스 몽구스
```

설치 프로세스가 완료되면 몽구스모듈을 루트 앱모듈로 가져올 수 있습니다.

`@@파일명(앱.모듈)`

'@nestjs/common'에서 `{ Module }`을 가져옵니다;

'@nestjs/mongoose'에서 `{ 몽구스모듈 }`을 가져옵니다;

`모듈({`

임포트합니다: [몽구스모듈.`forRoot('mongodb://localhost/nest')`],
})

내보내기 클래스 `AppModule {`

`forRoot()` 메서드는 여기에 설명된 대로 Mongoose 패키지의 `mongoose.connect()`와 동일한 구성 객체를 받아들입니다.

모델 주입

몽구스에서는 모든 것이 [스키마에서](#) 파생됩니다. 각 스키마는 MongoDB 컬렉션에 매핑되며 해당 컬렉션 내의 문서 모양을 정의합니다. 스키마는 [모델을](#) 정의하는 데 사용됩니다. 모델은 기본 MongoDB 데이터베이스에서 문서를 만들고 읽는 일을 담당합니다.

스키마는 NestJS 데코레이터를 사용하거나 몽구스 자체에서 수동으로 생성할 수 있습니다. 데코레이터를 사용하여 스키마를 생성하면 상용구가 크게 줄어들고 전반적인 코드 가독성이 향상됩니다.

`CatSchema`을 정의해 보겠습니다:

@@파일명(스키마/캣.스키마)

'@nestjs/mongoose'에서 { Prop, Schema, SchemaFactory }를 가져오고,
'mongoose'에서 { HydratedDocument }를 가져옵니다;

내보내기 유형 CatDocument = HydratedDocument<Cat>;

@Schema()

내보내기 클래스 Cat {
 @Prop()

 이름: 문자열;

```

@Prop()
나이: 숫자;

Prop() 품종: 문
자열;
}

export const CatSchema = SchemaFactory.createForClass(Cat);

```

정보 힌트 [nestjs/mongoose의 DefinitionsFactory](#) 클래스를 사용하여 원시 스키마 정의를 생성할 수도 있습니다. 이렇게 하면 제공한 메타데이터를 기반으로 생성된 스키마 정의를 수동으로 수정할 수 있습니다. 이는 데코레이터로 모든 것을 표현하기 어려울 수 있는 특정 에지 케이스에 유용합니다.

[스키마\(\)](#) 데코레이터는 클래스를 스키마 정의로 표시합니다. 이 데코레이터는 `Cat` 클래스를 같은 이름의 몽고 DB 컬렉션에 매핑하지만 끝에 "s"가 추가되므로 최종 몽고 컬렉션 이름은 `cats`가 됩니다. 이 데코레이터는 스키마 옵션 객체인 단일 선택적 인수를 받습니다. 이 객체는 일반적으로 `mongoose.Schema` 클래스 생성자의 두 번째 인수로 전달할 수 있는 객체라고 생각하면 됩니다(예: `new mongoose.Schema(_, options)`). 사용 가능한 스키마 옵션에 대해 자세히 알아보려면 [이](#) 장을 참조하세요.

[Prop\(\)](#) 데코레이터는 문서에서 속성을 정의합니다. 예를 들어, 위의 스키마 정의에서는 `이름`, `나이`, `품종`이라는 세 가지 속성을 정의했습니다. 이러한 속성의 [스키마 유형은](#) TypeScript 메타데이터(및 리플렉션) 기능 덕분에 자동으로 추론됩니다. 그러나 유형을 암시적으로 반영할 수 없는 더 복잡한 시나리오(예: 배열 또는 중첩된 객체 구조)에서는 다음과 같이 유형을 명시적으로 표시해야 합니다:

```

@Prop([문자열]) 태
그: 문자열[];

```

또는 [@Prop\(\)](#) 데코레이터는 옵션 객체 인수를 받습니다(사용 가능한 옵션에 대해 [자세히 알아보기](#)). 이를 통해 프로퍼티가 필요한지 여부를 표시하거나 기본값을 지정하거나 변경 불가능으로 표시할 수 있습니다. 예를 들어

```

@Prop({ 필수: true })
name: 문자열;

```

나중에 채우기 위해 다른 모델과의 관계를 지정하려는 경우 [@Prop\(\)](#) 데코레이터를 사용할 수도 있습니다. 예를

들어 Cat에 소유자라는 다른 컬렉션에 저장된 소유자가 있는 경우 프로퍼티에 유형과 참조가 있어야 합니다. 예를 들어

'몽구스'에서 *를 몽구스로 가져옵니다;

'./owners/schemas/owner.schema'에서 { Owner }를 가져옵니다;

```
// 클래스 정의 내부
Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'Owner' }) 소
유자: 소유자;
```

소유자가 여러 명인 경우, 숙소 구성은 다음과 같이 표시되어야 합니다:

```
Prop({ 유형: [{ 유형: mongoose.Schema.Types.ObjectId, ref: 'Owner' }] }) 소유자:
소유자[];
```

마지막으로, 원시 스키마 정의를 데코레이터에 전달할 수도 있습니다. 이는 예를 들어 프로퍼티가 클래스로 정의되지 않은 중첩된 객체를 나타낼 때 유용합니다. 이를 위해 다음과 같이 [@nestjs/mongoose](#) 패키지의 `raw()` 함수를 사용합니다:

```
@Prop(raw({
  firstName: { type: 문자열 }, 성 {
    유형: String
  })
})
세부 정보: 레코드<스트링, 입의>;
```

또는 데코레이터를 사용하지 않으려는 경우 스키마를 수동으로 정의할 수 있습니다. 예를 들어

```
export const CatSchema = new mongoose.Schema({
  name: String,
  나이: 숫자, 품종
  : 문자열,
});
```

`cat.schema` 파일은 `cats` 디렉터리의 폴더에 있으며, 이 폴더에서 `CatsModule`도 정의합니다. 스키마 파일은 원하는 위치에 저장할 수 있지만, 관련 도메인 객체 근처의 적절한 모듈 디렉터리에 저장하는 것이 좋습니다.

`CatsModule`을 살펴봅시다:

@@파일명 (cats.module)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/mongoose'에서 { 몽구스모듈 } 임포트;  
'./cats.controller'에서 { 캣츠컨트롤러 } 임포트;  
'./cats.service'에서 { 캣츠서비스 } 임포트;  
'./schemas/cat.schema'에서 { Cat, CatSchema }를 가져옵니다;
```

모듈 ({

```
  임포트합니다: [몽구스모듈.forFeature([{ 이름: Cat.name, 스키마: CatSchema  
  }])],  
  컨트롤러: [CatsController], 제공자:  
  [CatsService],
```

```
})
내보내기 클래스 CatsModule {}
```

몽구스모듈은 현재 범위에 등록할 모델을 정의하는 등 모듈을 구성할 수 있는 `forFeature()` 메서드를 제공합니다. 다른 모듈에서도 모델을 사용하려면 `CatsModule`의 `내보내기` 섹션에 몽구스모듈을 추가하고 다른 모듈에서 `CatsModule`을 가져오면 됩니다.

스키마를 등록한 후에는 다음을 사용하여 `Cat` 모델을 `CatsService`에 주입할 수 있습니다.

`@InjectModel()` 데코레이터:

```
@@파일명(cats.service)
'몽구스'에서 { 모델 }을 가져옵니다;
'@nestjs/common'에서 { Injectable }을 가져오고,
'@nestjs/mongoose'에서 { InjectModel }을 가져오고,
'./schemas/cat.schema'에서 { Cat }을 가져옵니다;
'./dto/create-cat.dto'에서 { CreateCatDto }를 가져옵니다;

@Injectable()
내보내기 클래스 CatsService {
  생성자(@InjectModel(Cat.name) private catModel: Model<Cat>) {}

  async create(createCatDto: CreateCatDto): Promise<Cat> {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  비동기 findAll(): Promise<Cat[]> {
    return this.catModel.find().exec();
  }
}
@@switch
'몽구스'에서 { 모델 }을 가져옵니다;
'@nestjs/common'에서 { Injectable, Dependencies }를 가져오고,
'@nestjs/mongoose'에서 { getModelToken }을 가져옵니다;
'./schemas/cat.schema'에서 { Cat }을 가져옵니다;

주입 가능()
@Dependencies(getModelToken(Cat.name))
내보내기 클래스 CatsService {
  constructor(catModel) {
```

```
    this.catModel = catModel;
}

async create(createCatDto) {
  const createdCat = new this.catModel(createCatDto);
  return createdCat.save();
}

async findAll() {
  이.catModel.find().exec()를 반환합니다;
```

```
    }  
}
```

연결

때로는 네이티브 [몽구스 연결](#) 객체에 액세스해야 할 수도 있습니다. 예를 들어 연결 객체에서 네이티브 API 호출을 하고 싶을 수 있습니다. 다음과 같이 `@InjectConnection()` 데코레이터를 사용하여 몽구스 커넥션을 삽입할 수 있습니다:

```
'@nestjs/common'에서 { Injectable }을 임포트합니다;  
'@nestjs/mongoose'에서 { InjectConnection }을 임포트하고,  
'mongoose'에서 { Connection }을 임포트합니다;  
  
@Injectable()  
내보내기 클래스 CatsService {  
  생성자(@InjectConnection() 비공개 연결: Connection) {}  
}  
}
```

여러 데이터베이스

일부 프로젝트에는 여러 데이터베이스 연결이 필요합니다. 이 모듈을 사용하면 이 작업도 수행할 수 있습니다. 여러 연결로 작업하려면 먼저 연결을 만듭니다. 이 경우 연결 이름 지정은 필수입니다.

```
@@파일명(앱.모듈)  
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/mongoose'에서 { 몽구스모듈 }을 가져옵니다;  
  
모듈({ import: [  
  몽구스모듈.forRoot('mongodb://localhost/test', {  
    connectionName: 'cats',  
  }),  
  MongooseModule.forRoot('mongodb://localhost/users', {  
    connectionName: 'users',  
  }),  
],  
})
```

```
내보내기 클래스 AppModule {}
```

경고 주의 이름이 없거나 같은 이름의 연결이 여러 개 있으면 재정의됩니다.

이 설정을 사용하면 `MongooseModule.forFeature()` 함수에 어떤 연결을 사용해야 하는지 알려줘야 합니다

```
모듈({ import: [
    몽구스모듈.forFeature([{ 이름: Cat.name, 스키마: CatSchema }], 'cats'),
  ],
})
내보내기 클래스 CatsModule {}
```

지정된 연결에 대한 연결을 삽입할 수도 있습니다:

```
'@nestjs/common'에서 { Injectable }을 임포트합니다;
'@nestjs/mongoose'에서 { InjectConnection }을 임포트하고,
'mongoose'에서 { Connection }을 임포트합니다;

@Injectable()
내보내기 클래스 CatsService {
  생성자(@InjectConnection('cats') 비공개 연결: Connection) {}
}
```

지정된 연결을 사용자 지정 공급자(예: 팩토리 공급자)에 주입하려면

`getConnectionToken()` 함수에 연결 이름을 인수로 전달합니다.

```
{
  제공하세요: CatsService,
  useFactory: (catsConnection: Connection) => {
    return new CatsService(catsConnection);
  },
  주입합니다: [getConnectionToken('cats')],
}
```

명명된 데이터베이스에서 모델을 주입하려는 경우 연결 이름을 `@InjectModel()` 데코레이터의 두 번째 매개 변수로 사용할 수 있습니다.

```
@@파일명(cats.service)
@Injectable()
내보내기 클래스 CatsService {
  constructor(@InjectModel(Cat.name, 'cats') private catModel: Model<Cat>)
{}
}

@@스위치 @Injectable()
@Dependencies(getModelToken(Cat.name, 'cats'))
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
```

```
    }  
}
```

후크(미들웨어)

미들웨어(프리훅 및 포스트훅이라고도 함)는 비동기 함수를 실행하는 동안 제어권을 전달하는 함수입니다. 미들웨어는 스키마 수준에서 지정되며 플러그인([소스](#)) 작성에 유용합니다. 몽구스에서는 모델을 컴파일한 후 [pre\(\)](#) 또는 [post\(\)](#)를 호출하면 작동하지 않습니다. 모델 등록 전에 훅을 등록하려면 팩토리 프로바이더(예: [useFactory](#))와 함께 [MongooseModule](#)의 [forFeatureAsync\(\)](#) 메서드를 사용합니다. 이 기법을 사용하면 스키마 객체에 액세스한 다음 [pre\(\)](#) 또는 [post\(\)](#) 메서드를 사용하여 해당 스키마에 훅을 등록할 수 있습니다. 아래 예시를 참조하세요:

```
모듈({ import: [  
  몽구스모듈.forFeatureAsync([[  
    {  
      name: Cat.name,  
      useFactory: () => {  
        const schema = CatsSchema;  
        schema.pre('save', function () {  
          console.log('안녕하세요 저장 전입니다');  
        });  
        반환 스키마;  
      },  
      ],  
    ],  
  ]  
})  
내보내기 클래스 AppModule {}
```

다른 [팩토리](#) 공급자와 마찬가지로 팩토리 함수는 [비동기화될](#) 수 있으며 다음을 통해 종속성을 주입할 수 있습니다. 주입합니다.

```
모듈({ import: [
    몽구스모듈.forFeatureAsync([
        {
            이름: Cat.name,
            임포트: [구성 모듈],
            useFactory: (configService: ConfigService) => {
                const schema = CatsSchema; schema.pre('save',
                    function() {
                        콘솔 로그(
                            `${configService.get('APP_NAME')}: 안녕하세요, 사전 저장에서`,
                        ),
                    });
                반환 스키마;
            },
        ],
    ],
}]);
```

```

        주입합니다: [구성 서비스],
    },
  ],
},
})

내보내기 클래스 AppModule {}

```

플러그인

주어진 스키마에 대한 [플러그인을 등록](#)하려면 `forFeatureAsync()` 메서드를 사용합니다.

```

모듈({ import: [
  몽구스모듈.forFeatureAsync([
    {
      name: Cat.name,
      useFactory: () => {
        const schema = CatsSchema;
        schema.plugin(require('mongoose-autopopulate'));
        return schema;
      },
    },
  ]),
],
})

내보내기 클래스 AppModule {}

```

모든 스키마에 대한 플러그인을 한 번에 등록하려면 `Connection` 객체의 `.plugin()` 메서드를 호출합니다. 모델을 생성하기 전에 연결에 액세스해야 하며, 이를 위해 `connectionFactory`를 사용합니다:

@@파일명(앱.모듈)

'@nestjs/common'에서 `{ Module }`을 가져옵니다;

'@nestjs/mongoose'에서 `{ 몽구스모듈 }`을 가져옵니다;

```

모듈({ import: [
  MongooseModule.forRoot('mongodb://localhost/test', {
    connectionFactory: (connection) => {
      connection.plugin(require('mongoose-autopopulate')); 반환
      연결;
    }
  }),
],
})

내보내기 클래스 AppModule {}

```

차별 주의자

판별자는 스키마 상속 메커니즘입니다. 이를 통해 동일한 기본 MongoDB 컬렉션 위에 스키마가 겹치는 여러 모델을 가질 수 있습니다.

단일 컬렉션에서 다양한 유형의 이벤트를 추적하고 싶다고 가정해 보겠습니다. 모든 이벤트에는 타임스탬프가 있습니다.

```
@@파일명(이벤트.스키마)
스키마({ 판별자 키: '종류' }) 내보내기 클래스 이
벤트 {
  @Prop({
    유형입니다: 문자열,
    필수: true,
    열거형: [ClickedLinkEvent.name, SignUpEvent.name],
  })
  종류: 문자열;

  Prop({ 유형: 날짜, 필수: 참 }) 시간: 날짜;
}
```

정보 힌트 몽구스가 따른 판별자 모델 간의 차이를 구분하는 방법은 판별자 키를 사용하는 것입니다.

입니다. 몽구스는 스키마에 문자열 경로인 _____라는 문자열 경로를 스키마에 추가하여 이 문서가 어떤 판별자의 인스턴스인지 추적하는 데 사용합니다. 판별자 키 옵션을 사용하여 판별 경로를 정의할 수도 있습니다.

SignedUpEvent 및 ClickedLinkEvent 인스턴스는 일반 이벤트와 동일한 컬렉션에 저장됩니다.

이제 다음과 같이 ClickedLinkEvent 클래스를 정의해 보겠습니다:

```
@@파일명(클릭-링크-이벤트.스키마) @Schema()  
내보내기 클래스 ClickedLinkEvent { 종  
  류: 문자열;  
  시간: 날짜;  
  
  @Prop({ 유형: 문자열, 필수: true }) url: 문자열  
;  
}  
  
export const ClickedLinkEventSchema =  
SchemaFactory.createForClass(ClickedLinkEvent);
```

그리고 SignUpEvent 클래스:

```

@@파일명(가입-이벤트.스키마) @Schema()

내보내기 클래스 SignUpEvent {
    종류: 문자열;
    시간: 날짜;

    @Prop({ 유형: 문자열, 필수: true }) 사용자: 문자
    열;
}

export const SignUpEventSchema =
SchemaFactory.createForClass(SignUpEvent);

```

이렇게 하면 판별자 옵션을 사용하여 주어진 스키마에 대한 판별자를 등록할 수 있습니다. 이 옵션은 [몽구스모듈](#) `.forFeature`와 [몽구스모듈](#) `.forFeatureAsync` 모두에서 작동합니다:

```

@@파일명(event.module)

'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/mongoose'에서 { 몽구스모듈 }을 가져옵니다;

모듈({ import: [
    몽구스모듈.forFeature([
        {
            이름: Event.name, 스키마
            : EventSchema, 판별자:
            [
                { 이름: ClickedLinkEvent.name, 스키마: ClickedLinkEventSchema },
                { 이름: SignUpEvent.name, 스키마: SignUpEventSchema },
            ],
            },
        ],
    ]),
} )
}

이벤트 모듈 클래스 {} 내보내기

```

테스트

애플리케이션을 단위 테스트할 때 일반적으로 데이터베이스 연결을 피하여 테스트 스위트를 더 간단하게 설정하고 실행 속도를 높이고자 합니다. 하지만 클래스는 연결 인스턴스에서 가져온 모델에 의존할 수 있습니다. 이러한 클

래스를 어떻게 해결할 수 있을까요? 해결책은 모의 모델을 만드는 것입니다.

이 작업을 더 쉽게 하기 위해 [@nestjs/mongoose](#) 패키지는 토큰 이름에 따라 준비된 [인젝션 토큰을](#) 반환하는 `getModelToken()` 함수를 노출합니다. 이 토큰을 사용하면 [사용 클래스](#), [사용 값](#), [사용 팩토리](#) 등 표준 [사용자](#) 정의 [공급자](#) 기법을 사용하여 모의 구현을 쉽게 제공할 수 있습니다. 예를 들어

```
모듈({ providers:
  [
    CatsService,
    {
      제공: getModelToken(Cat.name), 사용값:
      catModel,
    },
  ],
})

내보내기 클래스 CatsModule {}
```

이 예제에서는 소비자가 객체 인스턴스를 주입할 때마다 하드코딩된 `catModel`(객체 인스턴스)이 제공됩니다.

`Model<Cat>`에 `@InjectModel()` 데코레이터를 사용

합니다. 비동기 구성

모듈 옵션을 정적이 아닌 비동기적으로 전달해야 하는 경우, `forRootAsync()`

메서드를 사용합니다. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성을 처리하는 몇 가지 기술을 제공합니

다. 한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
MongooseModule.forRootAsync({
  useFactory: () => ({
    uri: 'mongodb://localhost/nest',
  }),
});
```

다른 [팩토리](#) 공급자와 마찬가지로 팩토리 함수는 [비동기화될](#) 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.
주입합니다.

```
MongooseModule.forRootAsync({
  import: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    uri: configService.get<string>('MONGODB_URI'),
  }),
  주입합니다: [구성 서비스],
});
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 [몽구스](#) 모듈을 구성할 수도 있습니다:

```
MongooseModule.forRootAsync({  
  useClass: 몽구스컨피그서비스,  
});
```

위의 구성은 `MongooseModule` 내부에 `MongooseConfigService`를 인스턴스화하여 이를 사용하여 필요 한 옵션 객체를 생성합니다. 이 예제에서 `MongooseConfigService`는 아래와 같이 `MongooseOptionsFactory` 인터페이스를 구현해야 한다는 점에 유의하세요. 몽구스모듈은 제공된 클래스의 인스턴스화된 객체에서 `createMongooseOptions()` 메서드를 호출합니다.

```
@Injectable()
```

```
export 클래스 몽구스컨피그서비스는 몽구스옵션팩토리를 구현합니다 { createMongoose옵션():  
    MongooseModuleOptions {  
        반환 {  
            uri: 'mongodb://localhost/nest',  
        };  
    }  
}
```

내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려는 경우 몽구스모듈에 `사용Existing` 구문을 사용합니다.

```
MongooseModule.forRootAsync({  
    import: [ConfigModule],  
    useExisting: ConfigService,  
});
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

유효성 검사

웹 애플리케이션으로 전송되는 모든 데이터의 정확성을 검증하는 것이 가장 좋습니다. 들어오는 요청의 유효성을 자동으로 검사하기 위해 Nest는 바로 사용할 수 있는 여러 파이프를 제공합니다:

- ValidationPipe
- ParseIntPipe
- ParseBoolPipe
- ParseArrayPipe
- ParseUUIDPipe

유효성 검사 파이프는 강력한 [클래스](#) 유효성 검사기 패키지와 선언적 유효성 검사 데코레이터를 사용합니다.

ValidationPipe는 들어오는 모든 클라이언트 페이로드에 대해 유효성 검사 규칙을 적용하는 편리한 접근 방식을 제공하며, 특정 규칙은 각 모듈의 로컬 클래스/DTO 선언에 간단한 어노테이션으로 선언됩니다.

개요

파이프 챕터에서는 간단한 파이프를 빌드하고 컨트롤러, 메서드 또는 글로벌 앱에 바인딩하는 과정을 통해 프로세스가 어떻게 작동하는지 보여드렸습니다. 이 장의 주제를 가장 잘 이해하려면 해당장을 반드시 복습하세요. 여기에서는 ValidationPipe의 다양한 실제 사용 사례에 초점을 맞추고 고급 사용자 정의 기능 중 일부를 사용하는 방법을 보여드리겠습니다.

기본 제공 ValidationPipe 사용

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
$ npm i --save class-validator class-transformer
```

정보 힌트 ValidationPipe는 [@nestjs/common](#) 패키지에서 내보냅니다.

이 파이프는 [클래스 유효성](#) 검사기 및 [클래스 트랜스포머](#) 라이브러리를 사용하므로 사용할 수 있는 옵션이 많습니다. 파이프에 전달된 구성 개체를 통해 이러한 설정을 구성합니다. 다음은 기본 제공 옵션입니다:

```
내보내기 인터페이스 ValidationPipeOptions extends ValidatorOptions {  
    transform?: boolean;  
    disableErrorMessages?: boolean;  
    예외 팩토리?: (오류: 유효성 검사 오류[]) => any;  
}
```

이 외에도 모든 클래스 유효성 검사기 옵션(유효성 검사기 옵션 인터페이스에서 상속됨)을 사용할 수 있습니다:

옵션	유형	설명
----	----	----

<code>enableDebugMessage</code>	부울	<code>true</code> 로 설정하면, 유효성 검사기가 추가 경고 메시지를 인쇄합니다.
		를 콘솔로 전송하세요.
<code>skipUndefinedProperties</code>	부울	<code>true</code> 로 설정하면 유효성 검사기는 모든 유효성 검사를 건너뜁니다.
		유효성 검사 객체에 정의되지 않은 프로퍼티가 있습니다.
<code>skipNullProperties</code>	<code>boolean</code>	<code>true</code> 로 설정하면 유효성 검사기는 유효성 검사 대상에서 null인 모든 프로퍼티의 유효성을 검사를 건너뜁니다.
<code>skipMissingProperties</code>	<code>boolean</code>	<code>true</code> 로 설정하면 유효성 검사기는 유효성 검사 객체에서 null이거나 정의되지 않은 모든 프로퍼티의 유효성을 검사를 건너뜁니다.
화이트리스트	부울	true로 설정하면 유효성 검사기는 유효성 검사 데코레이터를 사용하지 않는 모든 프로퍼티에서 유효성 검사된(반환된) 객체를 제거합니다.
<code>forbidNonWhitelisted</code>	부울	<code>true</code> 로 설정하면 유효성 검사기는 유효성 검사 데코레이터를 사용하지 않는 모든 프로퍼티에서 유효성 검사된(반환된) 객체를 제거합니다.
<code>forbidUnknownValues</code>	부울	<code>true</code> 로 설정하면 화이트리스트에 없는 프로퍼티를 제거하는 대신 유효성 검사기가 예외를 던집니다.
<code>disableErrorMessage</code>	부울	<code>true</code> 로 설정하면 알 수 없는 객체의 유효성 검사 시도가 즉시 실패합니다.
<code>errorHttpStatusCode</code>	숫자	<code>true</code> 로 설정하면 유효성 검사 오류가 클라이언트에 반환되지 않습니다.
예외 팩토리	함수	이 설정을 사용하면 오류 발생 시 어떤 예외 유형을 사용할지 지정할 수 있습니다. 기본적으로 <code>BadRequestException</code> 을 던집니다.
		유효성 검사 오류의 배열을 받아 예외 객체를 반환합니다.
<code>groups</code>	문자열[]	객체의 유효성 검사 중에 사용할 그룹입니다.
항상	<code>boolean</code>	데코레이터의 항상 옵션을 기본값으로 설정합니다. 기본값은 다음과 같습니다.

데코레이터 옵션에서 재정의할 수 있습니다.

`strictGroups` 부울 그룹이 지정되지 않았거나 비어 있으면 그룹이 하나 이상 있는 데코레이터를 무시합니다. `dismissDefaultMessages` 부울 참으로 설정하면 유효성 검사에서 기본 메시지를 사용하지 않습니다. 명시적으로 설정되지 않은 경우 오류 메시지는 항상 정의되지 않습니다. `validationError.target boolean` 타겟을 유효성 검사 오류에 노출할지 여부를 나타냅니다. `validationError.value boolean` 유효성 검사 값을 유효성 검사 오류에 노출할지 여부를 나타냅니다. `stopAtFirstError boolean true`로 설정하면 첫 번째 오류가 발생한 후 지정된 프로퍼티의 유효성 검사가 중지됩니다. 기본값은 `false`입니다.

정보 공지 [클래스 유효성 검사](#)기 패키지에 대한 자세한 내용은 해당 [저장소에서](#) 확인하세요.

자동 유효성 검사

애플리케이션 수준에서 `ValidationPipe`를 바인딩하여 모든 엔드포인트가 잘못된 데이터를 수신하지 않도록 보호하는 것부터 시작하겠습니다.

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalPipes(new ValidationPipe());
  await app.listen(3000);
}

부트스트랩();
```

파이프를 테스트하기 위해 기본 엔드포인트를 만들어 보겠습니다.

```
@Post()
create(@Body() createUserDto: CreateUserDto) {
  return '이 작업은 새 사용자를 추가합니다';
}
```

정보 힌트 TypeScript는 제네릭이나 인터페이스에 대한 메타데이터를 저장하지 않으므로 DTO에서 제네릭이나 인터페이스를 사용할 경우 ValidationPipe가 들어오는 데이터의 유효성을 제대로 검사하지 못 할 수 있습니다. 따라서 DTO에 규칙점에 클래처를 사용하는 경우 가체호거를 사용할 수 없습니다. 즉, `{{ '{' }}`로 가져와야 합니다. `CreateUserDto {{ '{' }}` 유형 대신 `{{ '{' }}`을 임포트해야 합니다. `CreateUserDto {{ '{' }}`.

이제 `CreateUserDto`에 몇 가지 유효성 검사 규칙을 추가할 수 있습니다. 여기에 자세히 설명된 [클래스 유효성 검사기](#) 패키지에서 제공하는 데코레이터를 사용하여 이 작업을 수행합니다. 이렇게 하면 `CreateUserDto`를 사용하는 모든 경로에 이러한 유효성 검사 규칙이 자동으로 적용됩니다.

```
'class-validator'에서 { IsEmail, IsNotEmpty } import;

export class CreateUserDto {
  @IsEmail() 이메일: 문자열;

  @IsNotEmpty() 비밀번호: 문자열;
}
```

이러한 규칙을 적용하면 요청 본문에 잘못된 `이메일` 속성이 포함된 요청이 엔드포인트에 도달하면 애플리케이션이 자동으로 400 잘못된 요청 코드와 함께 다음 응답 본문으로 응답합니다:

```
{  
  "statusCode": 400, "오류":  
    "잘못된 요청",  
    "메시지": [ "이메일은 이메일이어야 합니다" ]  
}
```

요청 본문의 유효성을 검사하는 것 외에도, 다른 요청 객체 프로퍼티에도 `ValidationPipe`를 사용할 수 있습니다. 엔드포인트 경로에 `:id`를 허용하고 싶다고 가정해 보겠습니다. 이 요청 매개변수에 숫자만 허용되도록 하기 위해 다음 구문을 사용할 수 있습니다:

```
@Get(':id')
findOne(@Param() params: FindOneParams) {
    return '이 액션은 사용자를 반환합니다';
}
```

`FindOneParams`는 DTO와 마찬가지로 [클래스 유효성 검사기를](#) 사용하여 유효성 검사 규칙을 정의하는 클래스일 뿐입니다. 다음과 같이 보일 것입니다:

```
'class-validator'에서 { IsNumberString } 가져오기; 내보내기

클래스 FindOneParams {
    IsNumberString()
    id: 숫자;
}
```

세부 오류 비활성화

오류 메시지는 요청에 무엇이 잘못되었는지 설명하는 데 도움이 될 수 있습니다. 그러나 일부 프로덕션 환경에서는 자세한 오류를 비활성화하는 것을 선호합니다. 이렇게 하려면 옵션 객체를 `ValidationPipe`에 전달하면 됩니다 :

```
app.useGlobalPipes(
    new ValidationPipe({
        disableErrorMessages: true,
    }),
);
```

따라서 응답 본문에 자세한 오류 메시지가 표시되지 않습니다. 속성 스트리핑

또한 메서드 핸들러가 수신해서는 안 되는 프로퍼티를 필터링할 수 있는 `ValidationPipe`를 사용할 수 있습니다. `In`

이 경우 허용 가능한 속성을 화이트리스트에 추가할 수 있으며, 화이트리스트에 포함되지 않은 속성은 결과 개

체에서 자동으로 제거됩니다. 예를 들어, 처리기에서 [이메일](#) 및 [비밀번호](#) 속성을 기대하지만 요청에 [나이](#) 속성도 포함된 경우 이 속성은 결과 DTO에서 자동으로 제거될 수 있습니다. 이러한 동작을 사용하려면 [화이트리스트](#)를 [true](#)로 설정하세요.

```
app.useGlobalPipes(  
  new ValidationPipe({
```

```
    화이트리스트: true,  
  },  
);
```

true로 설정하면 화이트리스트에 없는 프로퍼티(유효성 검사 클래스에 데코레이터가 없는 프로퍼티)가 자동으로 제거됩니다.

또는 화이트리스트에 없는 속성이 있는 경우 요청 처리를 중지하고 사용자에게 오류 응답을 반환할 수 있습니다. 이 기능을 사용하려면 화이트리스트를 true로 설정하는 것과 함께 forbidNonWhitelisted 옵션 속성을 true로 설정하세요.

페이지로드 객체 변환

네트워크를 통해 들어오는 페이지는 일반 JavaScript 객체입니다. 유효성 검사 파이프는 페이지를 DTO 클래스에 따라 입력된 객체로 자동 변환할 수 있습니다. 자동 변환을 활성화하려면 transform을 true로 설정합니다. 이 작업은 메서드 수준에서 수행할 수 있습니다:

```
@@파일명(cats.controller)  
@Post()  
UsePipes(new ValidationPipe({ transform: true }))  
async create(@Body() createCatDto: CreateCatDto) {  
  this.catsService.create(createCatDto);  
}
```

이 동작을 전역적으로 사용하려면 전역 파이프에서 옵션을 설정합니다:

```
app.useGlobalPipes(  
  new ValidationPipe({  
    transform: true,  
  }),  
);
```

자동 변환 옵션을 활성화하면 ValidationPipe는 기본 유형 변환도 수행합니다. 다음 예제에서 findOne() 메서드는 추출된 id 경로 매개변수를 나타내는 하나의 인수를 받습니다:

```
@Get(':id')
findOne(@Param('id') id: number) {
  console.log(typeof id === 'number'); // true
  반환 '이 작업은 사용자를 반환합니다';
}
```

기본적으로 모든 경로 매개변수와 쿼리 매개변수는 네트워크를 통해 문자열로 전달됩니다. 위의 예에서는 메서드 서명에서 ID 유형을 숫자로 지정했습니다. 따라서

ValidationPipe는 문자열 식별자를 숫자로 자동 변환하려고 시도합니다. 명시적 변환

위 섹션에서는 ValidationPipe가 쿼리와 경로를 암시적으로 변환하는 방법을 보여드렸습니다. 매개변수를 사용할 수 있습니다. 그러나 이 기능을 사용하려면 자동 변환을 사용하도록 설정해야 합니다.

또는 (자동 변환을 비활성화한 상태에서) ParseIntPipe 또는 ParseBoolPipe를 사용하여 명시적으로 값을 캐스팅할 수 있습니다(앞서 언급했듯이 모든 경로 매개변수와 쿼리 매개변수는 기본적으로 네트워크를 통해 문자열로 제공되므로 ParseStringPipe는 필요하지 않음).

```
@Get(':id')
findOne(
  Param('id', ParseIntPipe) id: 숫자,
  @Query('sort', ParseBoolPipe) sort: 부울,
) {
  console.log(typeof id === '숫자'); // 참
  console.log(typeof sort === '부울'); // 참 반환 '이 작업
  은 사용자를 반환합니다';
```

정보 힌트 ParseIntPipe와 ParseBoolPipe는 [@nestjs/common](#)에서 내보냅니다.

패키지입니다.

매핑된 유형

CRUD(만들기/읽기/업데이트/삭제)와 같은 기능을 구축할 때 기본 엔티티 유형에서 변형을 구성하는 것이 유용할 때가 많습니다. Nest는 유형 변환을 수행하는 여러 유ти리티 함수를 제공하여 이 작업을 더욱 편리하게 만듭니다.

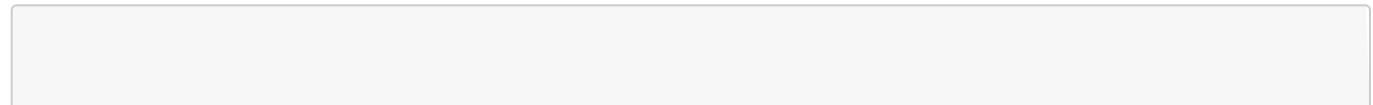
경고 애플리케이션에서 [@nestjs/swagger](#) 패키지를 사용하는 경우 [이 장](#)에서 매핑된 유형에 대한 자세한 내용을 참조하세요. 마찬가지로 [@nestjs/graphql](#) 패키지를 사용하는 경우 [이 장](#)을 참조하세요. 두 패키지 모두 타입에 크게 의존하므로 사용하려면 다른 임포트가 필요합니다. 따라서 앱 유형에 따라 적절한 [@nestjs/mapped-types](#) 대신 [@nestjs/swagger](#) 또는 [@nestjs/graphql](#)을 사용하는 경우 문서화되지 않은 다양한 부작용이 발생할 수 있습니다.

입력 유효성 검사 유형(DTO라고도 함)을 구축할 때 동일한 유형에 대해 생성 및 업데이트 변형을 구축하는 것이 유용한 경우가 많습니다. 예를 들어, 만들기 변형은 모든 필드를 필수로 설정하고 업데이트 변형은 모든 필드를 선

택 사항으로 설정할 수 있습니다.

Nest는 이 작업을 더 쉽게 수행하고 상용구를 최소화하기 위해 `PartialType()` 유ти리티 함수를 제공합니다.

`PartialType()` 함수는 입력 유형의 모든 속성이 선택 사항으로 설정된 유형(클래스)을 반환합니다. 예를 들어 다음과 같은 `create` 유형이 있다고 가정해 보겠습니다:



```
export 클래스 CreateCatDto {  
    이름: 문자열;  
    나이: 숫자; 품종  
    : 문자열;  
}
```

기본적으로 이러한 필드는 모두 필수입니다. 필드는 동일하지만 각 필드가 선택 사항인 유형을 만들려면 클래스 참조(`CreateCatDto`)를 인수로 전달하는 `PartialType()`을 사용합니다:

```
export class UpdateCatDto extends PartialType(CreateCatDto) {} {}
```

정보 힌트 `PartialType()` 함수는 `@nestjs/mapped-types` 패키지에서 가져온 것입니다.

`PickType()` 함수는 입력 유형에서 속성 집합을 선택하여 새 유형(클래스)을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

```
export 클래스 CreateCatDto { 이름  
    : 문자열;  
    나이: 숫자; 품종: 문  
    자열;  
}
```

이 클래스에서 `PickType()` 유ти리티 함수를 사용하여 프로퍼티 집합을 선택할 수 있습니다:

```
export class UpdateCatAgeDto extends PickType(CreateCatDto, [ 'age' ] as  
const) {} {}
```

정보 힌트 `PickType()` 함수는 `@nestjs/mapped-types` 패키지에서 가져온 것입니다.

`OmitType()` 함수는 입력 유형에서 모든 속성을 선택한 다음 특정 키 집합을 제거하여 유형을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

```
export 클래스 CreateCatDto {  
    이름: 문자열;  
    나이: 숫자; 품종  
    : 문자열;  
}
```

아래와 같이 이름을 제외한 모든 프로퍼티를 가진 파생 유형을 생성할 수 있습니다. 이 구조체에서 omitType의 두 번째 인수는 프로퍼티 이름의 배열입니다.

```
export class UpdateCatDto extends OmitType(CreateCatDto, [ 'name' ] as const) {} {}
```

정보 힌트 `OmitType()` 함수는 `@nestjs/mapped-types` 패키지에서 가져온 것입니다.

`IntersectionType()` 함수는 두 유형을 하나의 새로운 유형(클래스)으로 결합합니다. 예를 들어 다음과 같은 두 가지 유형으로 시작한다고 가정해 보겠습니다:

```
export 클래스 CreateCatDto {
    이름: 문자열;
    품종: 문자열;
}

내보내기 클래스 AdditionalCatInfo {
    색상: 문자열;
}
```

두 유형의 모든 속성을 결합한 새로운 유형을 생성할 수 있습니다.

```
내보내기 클래스 UpdateCatDto extends IntersectionType(
    CreateCatDto,
    AdditionalCatInfo,
) {}
```

정보 힌트 `IntersectionType()` 함수는 `@nestjs/mapped-types`에서 가져온 것입니다.

패키지입니다.

유형 매핑 유틸리티 함수는 컴포지션이 가능합니다. 예를 들어 다음은 `이름을 제외한 CreateCatDto` 유형의 모든 속성을 가진 유형(클래스)을 생성하며, 해당 속성은 선택 사항으로 설정됩니다:

```
내보내기 클래스 UpdateCatDto extends PartialType(
    OmitType(CreateCatDto, [ 'name' ] as const),
) {}
```

배열 구문 분석 및 유효성 검사

TypeScript는 제네릭이나 인터페이스에 대한 메타데이터를 저장하지 않으므로 DTO에서 제네릭이나 인터페이스

를 사용할 때 ValidationPipe가 들어오는 데이터의 유효성을 제대로 검사하지 못할 수 있습니다. 예를 들어, 다음 코드에서는 createUserDtos의 유효성이 올바르게 검사되지 않습니다.

```
@Post()  
createBulk(@Body() createUserDtos: CreateUserDto[]) {
```

```
'이 작업은 새 사용자를 추가합니다'를 반환합니다;
}
```

배열의 유효성을 검사하려면 배열을 감싸는 프로퍼티가 포함된 전용 클래스를 만들거나 [ParseArrayPipe](#).

```
@Post()
createBulk(
  Body(new ParseArrayPipe({ items: CreateUserDto })),
  createUserDtos: CreateUserDto[],
) {
  '이 작업은 새 사용자를 추가합니다'를 반환합니다;
}
```

또한 쿼리 매개변수를 구문 분석할 때 [ParseArrayPipe](#)가 유용하게 사용될 수 있습니다. 예를 들어 쿼리 매개변수로 전달된 식별자를 기반으로 사용자를 반환하는 [findByIds\(\)](#) 메서드입니다.

```
@Get()
findByIds(
  쿼리('ids', new ParseArrayPipe({ 항목: 숫자, 구분자: ',', })), ids: 숫자[],
) {
  반환 '이 액션은 아이디별로 사용자를 반환합니다';
}
```

이 구조는 다음과 같이 HTTP [GET](#) 요청에서 들어오는 쿼리 매개변수의 유효성을 검사합니다:

```
GET /?ids=1,2,3
```

웹소켓 및 마이크로서비스

이 장에서는 HTTP 스타일 애플리케이션(예: Express 또는 Fastify)을 사용한 예제를 보여드리지만, [ValidationPipe](#)는 사용되는 전송 방법에 관계없이 웹소켓 및 마이크로서비스에 대해 동일하게 작동합니다.

자세히 알아보기

[여기에서](#) [클래스 유효성](#) 검사기 패키지에서 제공하는 사용자 지정 유효성 검사기, 오류 메시지 및 사용 가능한 데코레이터에 대해 자세히 알아보세요.

캐싱

캐싱은 앱의 성능을 향상시키는 데 도움이 되는 훌륭하고 간단한 기술입니다. 캐싱은 고성능 데이터 액세스를 제공하는 임시 데이터 저장소 역할을 합니다.

설치

먼저 필요한 패키지를 설치합니다:

```
npm 설치 @nestjs/cache-manager 캐시-관리자
```

경고 캐시 관리자 버전 4는 TTL(Time-To-Live)에 초를 사용합니다. 현재 버전의 캐시 관리자(v5)는 대신 밀리초를 사용하도록 전환되었습니다. NestJS는 값을 변환하지 않고 단순히 사용자가 제공한 TTL을 라이브러리에 전달합니다. 다시 말해

- 캐시 관리자 v4를 사용하는 경우, ttl을 초 단위로 입력합니다.
- 캐시 관리자 v5를 사용하는 경우, ttl을 밀리초 단위로 입력합니다.

NestJS는 캐시 관리자 버전 4를 대상으로 출시되었으므로 문서에서는 초를 기준으로 합니다.

인메모리 캐시

Nest는 다양한 캐시 스토리지 제공업체를 위한 통합 API를 제공합니다. 기본 제공되는 것은 인메모리 데이터 저장소입니다. 그러나 Redis와 같은 보다 포괄적인 솔루션으로 쉽게 전환할 수 있습니다.

캐싱을 활성화하려면 캐시 모듈을 임포트하고 등록() 메서드를 호출합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/cache-manager'에서 { CacheModule }을 임포트하고,  
'./app.controller'에서 { AppController }를 임포트합니다;
```

```
모듈({  
  임포트합니다: [CacheModule.register()], 컨트롤러:  
  [AppController],  
})
```

캐시 스토어와 상호 작용하기

캐시 관리자 인스턴스와 상호 작용하려면 다음과 같이 **캐시 관리자** 토큰을 사용하여 클래스에 주입하세요:

```
생성자(@Inject(CACHE_MANAGER) private cacheManager: Cache) {}
```

정보 힌트 캐시 클래스는 캐시 매니저에서 가져오고, 캐시_매니저 토큰은 [@nestjs/cache-manager](#)

패키지에서 가져옵니다.

캐시 인스턴스(캐시 관리자 패키지의 캐시 인스턴스)의 `get` 메서드는 캐시에서 항목을 검색하는 데 사용됩니다.

항목이 캐시에 존재하지 않으면 `null`이 반환됩니다.

```
const value = await this.cacheManager.get('key');
```

캐시에 항목을 추가하려면 `set` 메서드를 사용합니다:

```
await this.cacheManager.set('key', 'value');
```

캐시의 기본 만료 시간은 5초입니다.

다음과 같이 이 특정 키에 대한 TTL(초 단위의 만료 시간)을 수동으로 지정할 수 있습니다:

```
await this.cacheManager.set('key', 'value', 1000);
```

캐시 만료를 비활성화하려면 `ttl` 구성 속성을 `0`으로 설정합니다:

```
await this.cacheManager.set('key', 'value', 0);
```

캐시에서 항목을 제거하려면 `del` 메서드를 사용합니다:

```
await this.cacheManager.del('key');
```

전체 캐시를 지우려면 `reset` 방법을 사용하세요:

```
await this.cacheManager.reset();
```

응답 자동 캐싱

경고 GraphQL 애플리케이션에서 인터셉터는 각 필드 리졸버에 대해 별도로 실행됩니다. 따라서 인터셉터를 사용하여 응답을 캐시하는 `attache`제대로 작동하지 않습니다.

응답 자동 캐싱을 사용하려면 데이터를 캐시하려는 위치에 `CacheInterceptor`를 연결하기만 하면 됩니다.

컨트롤러() @사용인터셉터(캐시인터셉터)

```
내보내기 클래스 AppController {
  @Get()
  findAll(): string[] {
    return [];
  }
}
```

경고 경고 `GET` 엔드포인트만 캐시됩니다. 또한 네이티브 응답 객체(`@Res()`)를 삽입하는 HTTP 서버 경로는 캐시 인터셉터를 사용할 수 없습니다. 자세한 내용은 [응답 매핑을 참조하세요.](#)

필요한 상용구의 양을 줄이려면 모든 엔드포인트에 [캐시인터셉터를](#) 전역적으로 바인딩하면 됩니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/cache-manager'에서 { CacheModule, CacheInterceptor }를 임포트하고,
'./app.controller'에서 { AppController }를 임포트합니다;
'@nestjs/core'에서 { APP_INTERCEPTOR }를 임포트합니다;

모듈({
  임포트합니다: [CacheModule.register()],
  컨트롤러: [AppController], providers: [
    {
      제공: APP_INTERCEPTOR,
      useClass: 캐시인터셉터,
    },
  ],
})
내보내기 클래스 AppModule {}
```

캐싱 사용자 지정

모든 캐시된 데이터에는 고유한 만료 시간([TTL](#))이 있습니다. 기본값을 사용자 정의하려면 옵션 객체를 `register()` 메서드에 전달합니다.

```
CacheModule.register({
  ttl: 5, // 초
  최대: 10, // 캐시 내 최대 항목 수
});
```

모듈을 전 세계적으로 사용

다른 모듈에서 캐시 모듈을 사용하려면 모든 Nest 모듈의 표준처럼 캐시 모듈을 임포트해야 합니다. 또는 아래와 같이 옵션 객체의 `isGlobal` 속성을 `true`로 설정하여 전역 모듈로 선언할 수도 있습니다. 이 경우 루트 모듈(예: `AppModule`)에서 `CacheModule`을 로드한 후에는 다른 모듈에서 임포트할 필요가 없습니다.

```
CacheModule.register({
  isGlobal: true,
});
```

글로벌 캐시 재정의

글로벌 캐시가 활성화되어 있는 동안 캐시 항목은 경로 경로에 따라 자동 생성되는 [캐시키](#) 아래에 저장됩니다. 메소드별로 특정 캐시 설정([@CacheKey\(\)](#) 및 [@CacheTTL\(\)](#))을 재정의할 수 있으므로 개별 컨트롤러 메소드에 대한 맞춤형 캐시 전략을 사용할 수 있습니다. 이는 [서로 다른 캐시 저장소를](#) 사용할 때 가장 적절할 수 있습니다

컨트롤러()

```
export class AppController {
  @CacheKey('custom_key')
  @CacheTTL(20)
  findAll(): string[] {
    return [];
  }
}
```

정보 힌트 [@CacheKey\(\)](#) 및 [@CacheTTL\(\)](#) 데코레이터는

[nestjs/cache-manager](#) 패키지.

[캐시키\(\)](#) 데코레이터는 해당 [@CacheTTL\(\)](#) 데코레이터와 함께 또는 없이 사용할 수 있으며, 그 반대의 경우도 마찬가지입니다. [캐시키\(\)](#) 데코레이터만 재정의하거나 [캐시TTL\(\)](#) 데코레이터만 재정의하도록 선택할 수 있습니다. 데코레이터로 재정의하지 않은 설정은 전역에 등록된 기본값을 사용합니다([캐싱 사용자 정의](#) 참조).

웹소켓 및 마이크로서비스

사용 중인 전송 방법에 관계없이 마이크로서비스의 패턴뿐만 아니라 웹소켓 구독자에게도 [캐시인터셉터를](#) 적용할 수 있습니다.

```
@@파일명()
@CacheKey('events')
사용 인터셉터(캐시인터셉터) @SubscribeMessage('이벤트')
handleEvent(client: Client, data: string[]): Observable<string[]> {
    return [];
}

@@스위치
@CacheKey('events')
UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client, data) {
    반환 [];
}
```

그러나 캐시된 데이터를 나중에 저장하고 검색하는 데 사용되는 키를 지정하려면 추가 `@CacheKey()` 데코레이터가 필요합니다. 또한 모든 것을 캐시해서는 안 된다는 점에 유의하세요. 단순히 데이터를 쿼리하는 것이 아니라 일부 비즈니스 작업을 수행하는 액션은 절대 캐시해서는 안 됩니다.

또한 `@CacheTTL()` 데코레이터를 사용하여 캐시 만료 시간(TTL)을 지정할 수 있으며, 이 경우 글로벌 기본 TTL 값이 재정의됩니다.

```
@@파일명()
@CacheTTL(10)
사용 인터셉터(캐시인터셉터)

@SubscribeMessage('이벤트')
handleEvent(client: Client, data: string[]): Observable<string[]> {
    return [];
}
@@switch
@CacheTTL(10)
UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client, data) {
    반환 [];
}
```

정보 힌트 `@CacheTTL()` 데코레이터는 대응하는 `@CacheKey()`와 함께 또는 없이 사용할 수 있습니다.
데코레이터.

추적 조정

기본적으로 Nest는 요청 URL(HTTP 앱의 경우) 또는 캐시 키(웹소켓 및 마이크로서비스 앱의 경우 `@CacheKey()` 데코레이터를 통해 설정됨)를 사용하여 캐시 레코드를 엔드포인트와 연결합니다.

그럼에도 불구하고 때때로 다른 요소(예: [프로필](#) 엔드포인트를 올바르게 식별하기 위한 [권한 부여](#))를 사용하는 등 다양한 요소에 따라 추적을 설정하고 싶을 수 있습니다.

이를 위해 `CacheInterceptor`의 서브클래스를 생성하고 `trackBy()` 메서드를 재정의합니다.
메서드를 사용합니다.

```
@Injectable()  
클래스 HttpCacheInterceptor extends CacheInterceptor {  
    trackBy(context: ExecutionContext): string | undefined {  
        '키'를 반환합니다;  
    }  
}
```

다른 매장

이 서비스는 내부적으로 [캐시 매니저를](#) 활용합니다. [캐시 관리자](#) 패키지는 [Redis 스토어와](#) 같은 다양한 유용한 스토어를 지원합니다. 지원되는 저장소의 전체 목록은 [여기에서](#) 확인할 수 있습니다. 설정하려면

에서 해당 옵션과 함께 패키지를 등록() 함수에 전달하기만 하면 됩니다.

메서드를 사용합니다.

```
'redis'에서 { RedisClientOptions } 유형을 가져옵니다;
'cache-manager-redis-store'에서 redisStore로 * 임포트;
'@nestjs/common'에서 { Module }을 임포트합니다;
'@nestjs/cache-manager'에서 { CacheModule }을 임포트하고,
'./app.controller'에서 { AppController }를 임포트합니다;
```

```
모듈({ import: [
  CacheModule.register<RedisClientOptions>({
    store: redisStore,
    // 스토어별 구성: host: 'localhost',
    // 포트: 6379,
  }),
],
컨트롤러: [앱 컨트롤러],
})
```

경고 경고 메시지: AppModule 스토어는 레디스 v4를 지원하지 않습니다. ClientOpts 인터페이스가 존재하고 올바르게 작동하려면 최신 redis 3.x.x 주 릴리스를 설치해야 합니다. 이 업그레이드의 진행 상황을 추적하려면 이 [문제를](#) 참조하세요.

비동기 구성

컴파일 시 정적으로 전달하는 대신 모듈 옵션을 비동기적으로 전달하고 싶을 수 있습니다. 이 경우 비동기 구성 을 처리하는 여러 가지 방법을 제공하는 registerAsync() 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
CacheModule.registerAsync({
  useFactory: () => ({
    ttl: 5,
  }),
});
```

저희 팩토리는 다른 모든 비동기 모듈 팩토리처럼 작동합니다(비동기일 수 있고 인젝트를 통해 종속성을 주입할 수

있습니다).

```
CacheModule.registerAsync({  
    import: [ConfigModule],  
    사용 팩토리: 비동기 (config서비스: 구성 서비스) => ({
```

```
    ttl: configService.get('CACHE_TTL'),
}),
주입합니다: [구성 서비스],
});
```

또는 `useClass` 메서드를 사용할 수 있습니다:

```
CacheModule.registerAsync({
  useClass: CacheConfigService,
});
```

위의 구조는 `CacheModule` 내부에 `CacheConfigService`를 인스턴스화하고 이를 사용해 옵션 객체를 가져옵니다. `CacheConfigService`는 구성 옵션을 제공하기 위해 `CacheOptionsFactory` 인터페이스를 구현해야 합니다:

```
@Injectable()
CacheConfigService 클래스는 CacheOptionsFactory { createCacheOptions() 를 구현합니다: CacheModuleOptions {
  반환 { ttl:
    5,
  };
}
```

다른 모듈에서 가져온 기존 구성 공급자를 사용하려면

`사용기존` 구문:

```
CacheModule.registerAsync({
  import: [ConfigModule],
  useExisting: ConfigService,
});
```

한 가지 중요한 차이점을 제외하고는 `사용클래스`와 동일하게 작동하지만, 캐시모듈은 가져온 모듈을 조회하여 자체적으로 인스턴스화하지 않고 이미 생성된 `컨피그서비스`를 재사용합니다.

정보 힌트 `CacheModule#register` 및 `CacheModule#registerAsync`와 `CacheOptionsFactory`에는 스토어별 구성 옵션의 범위를 좁힐 수 있는 선택적 제네릭(유형 인수)이 있으므로 유형 안전합니다.

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

직렬화

직렬화는 네트워크 응답에서 객체가 반환되기 전에 발생하는 프로세스입니다. 클라이언트에 반환할 데이터를 변환하고 살균하기 위한 규칙을 제공하기에 적합한 곳입니다. 예를 들어 비밀번호와 같은 민감한 데이터는 항상 응답에서 제외해야 합니다. 또는 엔티티 속성의 하위 집합만 전송하는 등 특정 속성에 추가 변환이 필요할 수도 있습니다. 이러한 변환을 수동으로 수행하면 지루하고 오류가 발생하기 쉬우며 모든 경우를 처리했는지 확실하지 않을 수 있습니다.

개요

Nest는 이러한 작업을 간단한 방법으로 수행할 수 있도록 지원하는 내장 기능을 제공합니다.

[ClassSerializerInterceptor](#) 인터셉터는 강력한 [클래스 트랜스포머](#) 패키지를 사용하여 선언적이고 확장 가능한 객체 변환 방법을 제공합니다. 이 인터셉터가 수행하는 기본 작업은 메서드 핸들러가 반환한 값을 가져와 [클래스 트랜스포머](#)의 `instanceToPlain()` 함수를 적용하는 것입니다. 이렇게 함으로써 아래 설명된 대로 엔티티/DTO 클래스에 [클래스 트랜스포머](#) 데코레이터로 표현된 규칙을 적용할 수 있습니다.

정보 힌트 직렬화는 [StreamableFile](#) 응답에는 적용되지 않습니다.

속성 제외

사용자 엔티티에서 [비밀번호](#) 속성을 자동으로 제외하려고 한다고 가정해 보겠습니다. 다음과 같이 엔티티에 주석을 추가합니다:

```
import { Exclude } from 'class-transformer';

export class UserEntity {
    id: 숫자; 이름: 문자
    열; 성: 문자열;

    @Exclude() 비밀번호:
    문자열;

    constructor(partial: Partial<UserEntity>) {
        Object.assign(this, partial);
    }
}
```

이제 이 클래스의 인스턴스를 반환하는 메서드 핸들러가 있는 컨트롤러를 생각해 보겠습니다.

사용 인터셉터(클래스시리얼라이저인터셉터) `@Get()`

```
findOne(): UserEntity {  
    return new UserEntity({  
        id: 1,  
    })  
}
```

```

    이름: '카밀', 성 '미슬리비츠
    ', 비밀번호: '비밀번호',
  });
}

```

경고 클래스의 인스턴스를 반환해야 한다는 점에 유의하세요. 예를 들어 `{} '{ }' user: new UserEntity() {} '`와 같이 반환하면 객체가 제대로 직렬화되지 않습니다.

정보 힌트 `ClassSerializerInterceptor`는 [@nestjs/common](#)에서 가져옵니다.

이 엔드포인트가 요청되면 클라이언트는 다음과 같은 응답을 받습니다:

```
{
  "id": 1,
  "이름": "카밀", "성": "미슬리
  비츠"
}
```

인터셉터는 애플리케이션 전체에 적용될 수 있습니다([여기서](#) 설명한 대로). 인터셉터와 엔티티 클래스 선언을 조합하면 `UserEntity`를 반환하는 모든 메서드에서 `비밀번호` 속성을 제거할 수 있습니다. 이를 통해 이 비즈니스 규칙을 중앙 집중식으로 시행할 수 있습니다.

속성 노출

`노출()` 데코레이터를 사용하여 아래와 같이 프로퍼티의 별칭 이름을 제공하거나 프로퍼티 값을 계산하는 함수(계 터 함수와 유사)를 실행할 수 있습니다.

```

@Expose()
get fullName(): 문자열 {
  반환 `${이름} ${이름}`을 반환합니다;
}

```

변환

`Transform()` 데코레이터를 사용하여 추가 데이터 변환을 수행할 수 있습니다. 예를 들어 다음 구문은 전체 객체를 반환하는 대신 `역할` 엔티티의 이름 속성을 반환합니다.

```
@Transform(({ 값 }) => value.name) 역할:  
RoleEntity;
```

패스 옵션

변환 함수의 기본 동작을 수정하고 싶을 수 있습니다. 기본 설정을 재정의하려면 옵션 객체에

@SerializeOptions() 데코레이터를 사용하여 해당 설정을 전달합니다.

```
@SerializeOptions({ 제외 접두사:  
  [ '_'],  
})  
@Get()  
findOne(): UserEntity { 반환  
  새 UserEntity();  
}
```

정보 힌트 @SerializeOptions() 데코레이터는 [@nestjs/common](#)에서 가져온 것입니다.

SerializeOptions()를 통해 전달된 옵션은 기본 [인스턴스ToPlain\(\)](#) 함수의 두 번째 인수로 전달됩니다. 이 예제에서는 _ 접두사로 시작하는 모든 프로퍼티를 자동으로 제외합니다.

예

작동 예제는 [여기에서](#) 확인할 수 있습

니다. 웹소켓 및 마이크로서비스

이 장에서는 HTTP 스타일 애플리케이션(예: Express 또는 Fastify)을 사용하는 예제를 보여 주지만, 이 장에서는 ClassSerializerInterceptor는 사용되는 전송 방법에 관계없이 웹소켓과 마이크로서비스에서 동일하게 작동합니다.

자세히 알아보기

클래스 트랜스포머 패키지에서 제공하는 데코레이터 및 옵션에 대한 자세한 내용은 [여기](#)를 참조하세요.

버전 관리

정보 힌트 이 장은 HTTP 기반 애플리케이션에만 해당됩니다.

버전 관리를 사용하면 동일한 애플리케이션 내에서 서로 다른 버전의 컨트롤러 또는 개별 경로를 실행할 수 있습니다. 애플리케이션은 매우 자주 변경되며, 이전 버전의 애플리케이션을 계속 지원하면서 변경해야 하는 경우가 드물지 않습니다.

지원되는 버전 관리 유형은 4가지입니다:

URI

버전 관리

버전은 요청의 URI 내에서 전달됩니다(기본값) 사용자 지정 요청 헤

헤더 버전 관

리 더에 버전이 지정됩니다.

미디어 유형

버전 관리

요청의 **Accept** 헤더에는 버전이 지정됩니다.

사용자 지정

버전 관리

요청의 모든 측면을 사용하여 버전을 지정할 수 있습니다. 해당 버전을 추출하기 위한 사용자 정의 함수가 제공됩니다.

URI 버전 관리 유형

URI 버전 관리에서는 다음과 같이 요청의 URI 내에 전달된 버전을 사용합니다.

`https://example.com/v1/route` 및 `https://example.com/v2/route`.

경고 URI 버전 지정 시 버전은 **글로벌 경로 접두사**(있는 경우) 뒤와 컨트롤러 또는 경로 경로 앞에 자동으로 URI에 추가됩니다.

애플리케이션에 대해 URI 버전 관리를 사용 설정하려면 다음과 같이 하세요:

@@파일명 (메인)

```
const app = await NestFactory.create(AppModule);
// 또는 "app.enableVersioning()"
app.enableVersioning({
  유형입니다: 버전 관리 유형.URI,
});
await app.listen(3000);
```

경고 URI의 버전은 기본적으로 자동으로 접두사 앞에 `v`가 붙지만 `접두사` 키를 원하는 접두사로 설정하거나 비활성화하려는 경우 `false`로 설정하여 접두사 값을 구성할 수 있습니다.

정보 힌트 유형 속성에 사용할 수 있는 `VersioningType` 열거형은 `@nestjs/common` 패키지에서 가져옵니다.

헤더 버전 관리 유형

헤더 버전 관리에서는 사용자 지정 사용자 지정 요청 헤더를 사용하여 헤더 값이 요청에 사용할 버전이 되는 버전을 지정합니다.

헤더 버전 관리를 위한 HTTP 요청 예시:

애플리케이션에 헤더 버전 관리를 사용 설정하려면 다음과 같이 하세요:

@@파일명(메인)

```
const app = await NestFactory.create(AppModule);
app.enableVersioning({
    유형: VersioningType.HEADER, 헤더
    : '사용자 지정 헤더',
});
await app.listen(3000);
```

헤더 속성은 요청의 버전을 포함할 헤더의 이름이어야 합니다.

정보 힌트 유형 속성에 사용할 수 있는 **VersioningType** 열거형은 [@nestjs/common](#) 패키지에서 가져옵니다.

미디어 유형 버전 관리 유형

미디어 유형 버전 지정은 요청의 **Accept** 헤더를 사용하여 버전을 지정합니다.

Accept 헤더 내에서 버전은 세미콜론(;)으로 미디어 유형과 구분됩니다. 그런 다음 요청에 사용할 버전을 나타내는 키-값 쌍을 포함해야 합니다(예: **Accept: application/json;v=2**). 키와 구분 기호를 포함하도록 구성할 버전을 결정할 때 키는 접두사로 더 많이 취급됩니다.

애플리케이션에 미디어 유형 버전 관리를 사용 설정하려면 다음을 수행합니다:

@@파일명(메인)

```
const app = await NestFactory.create(AppModule);
app.enableVersioning({
    유형: 버전관리유형.MEDIA_TYPE, 키:
    'v=',
});
await app.listen(3000);
```

키 속성은 버전을 포함하는 키-값 쌍의 키와 구분 기호여야 합니다. **Accept: application/json;v=2** 예제에

서 키 속성은 v=로 설정됩니다.

정보 힌트 유형 속성에 사용할 수 있는 `VersioningType` 열거형은 `@nestjs/common` 패키지에서 가져옵니다.

사용자 지정 버전 관리 유형

사용자 정의 버전 관리에서는 요청의 모든 측면을 사용하여 버전(또는 버전)을 지정합니다. 들어오는 요청은 문자열 또는 문자열 배열을 반환하는 [추출기](#) 함수를 사용하여 분석됩니다.

요청자가 여러 버전을 제공한 경우 추출 함수는 가장 큰/가장 높은 버전에서 가장 작은/가장 낮은 버전 순으로 정렬된 문자열 배열을 반환할 수 있습니다. 버전은 가장 높은 버전에서 가장 낮은 버전 순으로 경로에 매칭됩니다.

[추출기](#)에서 빈 문자열 또는 배열이 반환되면 일치하는 경로가 없으며 404가 반환됩니다. 예를 들어, 들어오는 요청이 버전 1, 2, 3을 지원한다고 지정하는 경우 [추출기는](#) 반드시 [3, 2, 1]. 이렇게 하면 가능한 가장 높은 경로 버전이 먼저 선택됩니다.

버전 [3, 2, 1]이 추출되었지만 경로가 버전 2와 1에 대해서만 존재하는 경우 버전 2와 일치하는 경로가 선택됩니다(버전 3은 자동으로 무시됨).

경고 주의 추출기에서 반환된 배열을 기준으로 가장 일치하는 버전을 선택하는 것은 설계상의 제한으로 인해 Express 어댑터에서 안정적으로 작동하지 않습니다. 단일 버전(문자열 또는 요소 1개로 구성된 배열)은 Express에서 정상적으로 작동합니다. Fastify는 가장 일치하는 버전 선택과 단일 버전 선택을 모두 올바르게 지원합니다.

애플리케이션에 사용자 지정 버전 관리를 사용하려면 다음과 같이 [추출기](#) 함수를 만들어 애플리케이션에 전달합니다:

@@파일명 (메인)

```
// 사용자 정의 헤더에서 버전 목록을 가져와 정렬된 배열로 변환하는 추출기 예제입니다.  
// 이 예에서는 Fastify를 사용하지만 Express 요청도 비슷한 방식으로 처리할 수 있습니다.  
  
const extractor = (request: FastifyRequest): string | string[] =>  
  [request.headers['custom-versioning-field'] ?? '']  
    .flatMap(v => v.split(','))  
    .filter(v => !!v)  
    .sort()  
    .reverse()  
  
const app = await NestFactory.create(AppModule);  
app.enableVersioning({  
  유형: 버전 관리 유형, 추출기,  
});  
await app.listen(3000);
```

사용법

버전 관리를 사용하면 컨트롤러와 개별 경로를 버전 관리할 수 있으며 특정 리소스가 버전 관리를 거부할 수 있는 방법도 제공합니다. 버전 관리의 사용법은 애플리케이션에서 사용하는 버전 관리 유형에 관계없이 동일합니다

경고 애플리케이션에 버전 관리가 활성화되어 있지만 컨트롤러 또는 경로가 버전을 지정하지 않은 경우 해당 컨트롤러/경로에 대한 모든 요청은 [404](#) 응답 상태로 반환됩니다.

마찬가지로 해당 컨트롤러나 경로가 없는 버전이 포함된 요청이 수신되면 **404** 응답 상태도 반환됩니다.

컨트롤러 버전

컨트롤러에 버전을 적용하여 컨트롤러 내의 모든 경로에 대한 버전을 설정할 수 있습니다. 컨트롤러

에 버전을 추가하려면 다음과 같이 하세요:

```
@@파일명(cats.controller)
@Controller({
    버전: '1',
})
내보내기 클래스 CatsControllerV1 {
    @Get('cats')
    findAll(): 문자열 {
        반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다';
    }
}
@@스위치 @Controller({
    버전: '1',
})
내보내기 클래스 CatsControllerV1 {
    @Get('cats')
    findAll() {
        반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다';
    }
}
```

경로 버전

버전은 개별 경로에 적용할 수 있습니다. 이 버전은 컨트롤러 버전과 같이 경로에 영향을 미치는 다른 모든 버전보다 우선합니다.

개별 경로에 버전을 추가하려면 다음과 같이 하세요:

@@파일명(cats.controller)

'@nestjs/common'에서 { Controller, Get, Version }을 가져옵니다;

컨트롤러()

내보내기 클래스 CatsController {

 @Version('1')

 @Get('cats')

 findAllV1(): string {

 반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다;

 }

버전('2')

 @Get('cats')

```
findAllV2(): 문자열 {
    반환 '이 작업은 버전 2의 모든 고양이를 반환합니다;
}
}
@@switch
'@nestjs/common'에서 { Controller, Get, Version }을 가져옵니다;

컨트롤러()

내보내기 클래스 CatsController {
    @Version('1')
    @Get('cats')
    findAllV1() {
        반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다;
    }
}

버전('2')
@Get('cats')
findAllV2() {
    반환 '이 작업은 버전 2의 모든 고양이를 반환합니다;
}
}
```

여러 버전

컨트롤러 또는 경로에 여러 버전을 적용할 수 있습니다. 여러 버전을 사용하려면 버전을 배열로 설정합니다.

여러 버전을 추가하려면 다음과 같이 하세요:

```
@@파일명(cats.controller)
@Controller({
    버전입니다: ['1', '2'],
})

내보내기 클래스 CatsController {
    @Get('cats')
    findAll(): 문자열 {
        반환 '이 작업은 버전 1 또는 2에 대한 모든 고양이를 반환합니다';
    }
}

@@스위치 @Controller({
    버전입니다: ['1', '2'],
})

내보내기 클래스 CatsController {
    @Get('cats')
    findAll() {
        반환 '이 작업은 버전 1 또는 2에 대한 모든 고양이를 반환합니다';
    }
}
```

버전 "중립"

일부 컨트롤러나 라우트는 버전에 상관하지 않고 버전에 관계없이 동일한 기능을 사용할 수 있습니다. 이를 위해 버전을 **버전_중립** 기호로 설정할 수 있습니다.

들어오는 요청은 요청에 버전이 전혀 포함되어 있지 않은 경우뿐만 아니라 요청에 전송된 버전과 관계없이 **VERSION_NEUTRAL** 컨트롤러 또는 경로에 매핑됩니다.

경고 URI 버전 관리의 경우, **버전_중립** 리소스는 URI에 버전이 존재하지 않습니다.

버전 중립 컨트롤러 또는 경로를 추가하려면 다음과 같이 하세요:

```
@@파일명(cats.controller)
'@nestjs/common'에서 { Controller, Get, VERSION_NEUTRAL }을 가져옵니다;

컨트롤러({
  버전: 버전_중립,
})
내보내기 클래스 CatsController {
  @Get('cats')
  findAll(): 문자열 {
    반환 '이 작업은 버전에 관계없이 모든 고양이를 반환합니다';
  }
}
@@switch
'@nestjs/common'에서 { Controller, Get, VERSION_NEUTRAL }을 가져옵니다;

컨트롤러({
  버전: 버전_중립,
})
내보내기 클래스 CatsController {
  @Get('cats')
  findAll() {
    반환 '이 작업은 버전에 관계없이 모든 고양이를 반환합니다';
  }
}
```

글로벌 기본 버전

각 컨트롤러/개별 경로에 대한 버전을 제공하지 않거나 특정 버전을 지정하지 않은 모든 컨트롤러/경로에 대해

특정 버전을 기본 버전으로 설정하려는 경우 다음과 같이 `defaultVersion`을 설정할 수 있습니다:

```
@@filename(main)
app.enableVersioning({
  // ...
  defaultVersion: '1'
  // 또는
```

```
defaultVersion: ['1', '2']
// 또는
기본 버전: 버전_중립
});
```

미들웨어 버전 관리

미들웨어는 버전 관리 메타데이터를 사용하여 특정 경로의 버전에 맞게 미들웨어를 구성할 수도 있습니다. 이렇게 하려면 `MiddlewareConsumer.forRoutes()` 메서드의 매개변수 중 하나로 버전 번호를 제공하면 됩니다:

@@파일명(앱.모듈)

```
'@nestjs/common'에서 { Module, NestModule, MiddlewareConsumer }를 임포트하고,
'./common/middleware/logger.middleware'에서 { LoggerMiddleware }를 임포트하고,
'./cats/cats.module'에서 { CatsModule }을 임포트합니다;
'./cats/cats.controller'에서 { CatsController }를 가져옵니다;
```

모듈({

```
  수입: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    소비자
    .apply(LoggerMiddleware)
    .forRoutes({ path: '/cats', method: RequestMethod.GET, version: '2' })
  }
}
```

위의 코드를 사용하면 `LoggerMiddleware`는 `/cats` 엔드포인트의 버전 '2'에만 적용됩니다.

정보 공지 미들웨어는 이 섹션에 설명된 모든 버전 관리 유형에서 작동합니다: `URI`, `헤더`, `미디어 유형` 또는 `사용자 정의`.

작업 예약

작업 예약을 사용하면 임의의 코드(메서드/함수)가 정해진 날짜/시간에, 반복되는 간격으로 또는 지정된 간격 후에 한 번 실행되도록 예약할 수 있습니다. Linux 환경에서는 종종 OS 수준에서 [cron](#)과 같은 패키지로 이 작업을 처리합니다. Node.js 앱의 경우 크론과 유사한 기능을 에뮬레이트하는 여러 패키지가 있습니다. Nest는 널리 사용되는 Node.js [cron](#) 패키지와 통합되는 [@nestjs/schedule](#) 패키지를 제공합니다. 이 패키지는 이번 장에서 다루겠습니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
npm install --save @nestjs/schedule
```

작업 스케줄링을 활성화하려면 [스케줄](#) 모듈을 루트 앱 모듈로 가져와서 [forRoot\(\)](#)를 실행합니다.

정적 메서드를 사용합니다:

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/schedule'에서 { ScheduleModule }을 가져옵니다;

모듈({ import: [
    ScheduleModule.forRoot()
],
})

내보내기 클래스 AppModule {}
```

.[forRoot\(\)](#) 호출은 스케줄러를 초기화하고 앱 내에 존재하는 모든 선언적 [크론 작업](#), [타임아웃](#) 및 [간격](#)을 등록합니다. 등록은 [onApplicationBootstrap](#) 수명 주기 후크가 발생할 때 발생하며, 모든 모듈이 예약된 작업을 로드하고 선언했는지 확인합니다.

선언적 크론 작업

크론 작업은 임의의 함수(메서드 호출)가 자동으로 실행되도록 예약합니다. 크론 작업이 실행될 수 있습니다:

- 지정된 날짜/시간에 한 번.
- 반복 작업은 지정된 간격(예: 한 시간에 한 번, 일주일에 한 번, 5분에 한 번) 내에서 지정된 순간에 실행할 수 있습니다.

다음과 같이 실행할 코드가 포함된 메서드 정의 앞에 `@Cron()` 데코레이터를 사용하여 크론 작업을 선언합니다:

```
'@nestjs/common'에서 { Injectable, Logger }를 가져오고,  
'@nestjs/schedule'에서 { Cron }을 가져옵니다;
```

```

@Injectable()
내보내기 클래스 TasksService {
    비공개 읽기 전용 로거 = 새로운 로거(TasksService.name);

    @Cron('45 * * * *')
    handleCron() {
        this.logger.debug('현재 초가 45일 때 호출됩니다');
    }
}

```

이 예제에서는 현재 초가 45가 될 때마다 `handleCron()` 메서드가 호출됩니다. 즉, 메서드는 1분에 한 번, 45초가 될 때마다 실행됩니다.

`Cron()` 데코레이터는 모든 표준 [크론 패턴을 지원합니다](#):

- 별표(예: *)
- 범위(예: 1-3,5)• 걸음 수(예: */2)

위의 예에서는 데코레이터에 `45 * * * * *`를 전달했습니다. 다음 키는 크론 패턴 문자열의 각 위치가 어떻게 해석되는지 보여줍니다:

*	*	*	*	*	*
				요일	
			개월		
					월의 요일
		시간			
	분				

초(선택 사항)

몇 가지 샘플 크론 패턴은 다음과 같습니다:

* * * * * 매초

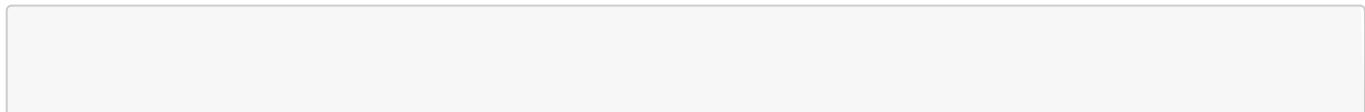
45 * * * * 매분, 45초마다

0 10 * * * 매시간, 10분이 시작될 때

0 */30 9-17 * * * 오전 9시에서 오후 5시 사이 30분마다

0 30 11 * * 1-5 월요일부터 금요일 오전 11시 30분

`nestjs/schedule` 패키지는 일반적으로 사용되는 크론 패턴이 포함된 편리한 열거형을 제공합니다. 이 열거형은 다음과 같이 사용할 수 있습니다:



```
'@nestjs/common'에서 { Injectable, Logger }를 임포트하고,  
'@nestjs/schedule'에서 { Cron, CronExpression }을 임포트합니다  
;  
  
@Injectable()  
내보내기 클래스 TasksService {  
  
    비공개 읽기 전용 로거 = 새로운 로거(TasksService.name);  
  
    @Cron(CronExpression.EVERY_30_SECONDS)  
    handleCron() {  
        this.logger.debug('30초마다 호출');  
    }  
}
```

이 예제에서는 30초마다 `handleCron()` 메서드가 호출됩니다.

또는 `@Cron()` 데코레이터에 JavaScript `Date` 객체를 제공할 수도 있습니다. 이렇게 하면 지정된 날짜에 작업이 정확히 한 번 실행됩니다.

정보 힌트 자바스크립트 날짜 산술을 사용하여 현재 날짜를 기준으로 작업을 예약합니다. 예를 들어

`@Cron(new Date(Date.now() + 10 * 1000))`을 사용하여 앱이 시작된 후 10초 후에 실행되도록 작업을 예약할 수 있습니다.

또한 `@Cron()` 데코레이터에 두 번째 매개변수로 추가 옵션을 제공할 수도 있습니다.

이름 크론 작업이 선언된 후 크론 작업에 액세스하고 제어하는 데 유용합니다.

실행할 시간대를 지정합니다. 이렇게 하면 실제 시간이 사용자의

시간대 시간대입니다. 시간대가 유효하지 않은 경우 오류가 발생합니다. 사용 가능한 모든 표준

시간대는 [모멘트 표준 시간대 웹사이트](#)에서 확인할 수 있습니다.

utcOffset 이렇게 하면 `시간대`를 사용하는 대신 표준 시간대의 오프셋을 지정할 수 있습니다.

매개변수.

disabled 작업이 전혀 실행되지 않는지 여부를 나타냅니다.

```
'@nestjs/common'에서 { Injectable }을 가져옵니다;  
'@nestjs/schedule'에서 { Cron, CronExpression }을 가져옵니다;
```

```
@Injectable()  
export class NotificationService {  
  @Cron('* * 0 * * *', {  
    이름: '알림', 시간대: '유럽/파리',  
  })  
  triggerNotifications() {}  
}
```

크론 작업이 선언된 후 크론 작업에 액세스하여 제어하거나, [동적 API](#)를 사용하여 크론 작업(크론 패턴이 런타임에 정의되는 경우)을 동적으로 만들 수 있습니다. API를 통해 선언적 크론 작업에 액세스하려면 다음과 같이 하세요.

는 데코레이터의 두 번째 인수로 선택적 옵션 객체의 **이름** 속성을 전달하여 작업을 이름과 연관시켜야 합니다.

선언적 간격

메서드가 (반복적으로) 지정된 간격으로 실행되어야 한다고 선언하려면 메서드 정의 앞에 **@Interval()** 데코레이터를 붙입니다. 아래와 같이 간격 값을 밀리초 단위의 숫자로 데코레이터에 전달합니다:

```
@Interval(10000)
handleInterval() {
    this.logger.debug('10초마다 호출');
}
```

정보 힌트 이 메커니즘은 내부적으로 JavaScript **setInterval()** 함수를 사용합니다. 크론 작업을 활용하여 반복 작업을 예약할 수도 있습니다.

동적 API를 통해 선언 클래스 외부에서 선언 간격을 제어하려면 다음 구문을 사용하여 간격을 이름과 연결하세요

:

```
@Interval('알림', 2500)
handleInterval() {}
```

또한 **동적 API**를 사용하면 런타임에 간격의 속성을 정의하는 동적 간격을 생성하고 이를 나열 및 삭제할 수 있습니다.

선언적 시간 초과

지정된 시간 초과 시 메서드가 (한 번) 실행되도록 선언하려면 메서드 정의 앞에 **@Timeout()** 데코레이터를 붙입니다. 아래와 같이 애플리케이션 시작부터 상대적인 시간 오프셋(밀리초 단위)을 데코레이터에 전달합니다:

```
@Timeout(5000)
handleTimeout() {
    this.logger.debug('5초 후 한 번 호출됨');
}
```

정보 힌트 이 메커니즘은 자바스크립트 **setTimeout()** 함수를 내부적으로 사용합니다.

동적 API를 통해 선언 클래스 외부에서 선언적 타임아웃을 제어하려면 다음 구문을 사용하여 타임아웃을 이름과 연결하세요:

```
@Timeout('알림', 2500)  
handleTimeout() {}
```

또한 동적 API를 사용하면 런타임에 타임아웃의 속성을 정의하는 동적 타임아웃을 생성하고 이를 나열 및 삭제할 수 있습니다.

동적 일정 모듈 API

nestjs/schedule 모듈은 선언적 [크론 작업](#), [시간](#) 초과 및 [간격](#)을 관리할 수 있는 동적 API를 제공합니다. 이 API를 사용하면 런타임에 속성이 정의되는 동적 크론 작업, 시간 초과 및 간격을 생성하고 관리할 수도 있습니다.

동적 크론 작업

코드의 어느 곳에서나 이름으로 [CronJob](#) 인스턴스에 대한 참조를 가져옵니다.

[SchedulerRegistry](#) API를 사용합니다. 먼저 표준 생성자 주입을 사용하여 [SchedulerRegistry](#)를 주입합니다

:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

정보 힌트 @nestjs/schedule 패키지에서 [SchedulerRegistry](#)를 가져옵니다.

그런 다음 다음과 같이 클래스에서 사용합니다. 다음 선언을 사용하여 크론 작업이 생성되었다고 가정합니다:

```
@Cron('* * 8 * * *', {
  name: '알림',
})
triggerNotifications()
```

다음을 사용하여 이 작업에 액세스합니다:

```
const job = this.schedulerRegistry.getCronJob('알림'); job.stop();
console.log(job.lastDate());
```

[getCronJob\(\)](#) 메서드는 명명된 크론 작업을 반환합니다. 반환된 [CronJob](#) 객체에는 다음과 같은 메서드가 있습니다:

- [stop\(\)](#) - 실행이 예약된 작업을 중지합니다.

- ◆ `start()` - 중지된 작업을 다시 시작합니다.
- ◆ `setTime(time: CronTime)` - 작업을 중지하고 새 시간을 설정한 다음 작업을 시작합니다.
- ◆
- ◆ `lastDate()` - 작업이 실행된 마지막 날짜의 문자열 표현을 반환합니다. `nextDates(count: 숫자)` - 예정된 작업 실행 날짜를 나타내는 모멘트 객체의 배열(크기 카운트)을 반환합니다.

정보 힌트 사람이 읽을 수 있는 형태로 렌더링하려면 모멘트 객체에서 `toDate()`를 사용하세요.

다음과 같이 `SchedulerRegistry#addCronJob` 메서드를 사용하여 새 크론 작업을 동적으로 생성합니다:

```
addCronJob(name: 문자열, seconds: 문자열) {  
    const job = new CronJob(`#${초} * * * *`, () => { this.logger.warn(`시간 (${초}) 동안 ${이름} 작업이 실행됩니다!`);  
});  
  
    this.schedulerRegistry.addCronJob(name, job);  
    job.start();  
  
    this.logger.warn(  
        매분 ${초}마다 ${이름} 작업이 추가되었습니다!  
    );  
}
```

이 코드에서는 `크론` 패키지의 `크론` 잡 객체를 사용하여 크론 잡을 생성합니다. `CronJob` 생성자는 첫 번째 인자로 (`@Cron()` 데코레이터와 마찬가지로) 크론 패턴을, 두 번째 인자로 크론 타이머가 실행될 때 실행될 콜백을 받습니다. `SchedulerRegistry#addCronJob` 메서드는 두 개의 인수를 받습니다. `CronJob`의 이름과 `CronJob` 자체입니다.

경고 경고 `SchedulerRegistry`에 액세스하기 전에 반드시 인젝션해야 합니다. Import
크론 패키지의 `크론잡`.

다음과 같이 `SchedulerRegistry#deleteCronJob` 메서드를 사용하여 명명된 크론 작업을 삭제합니다:

```
deleteCron(name: 문자열) {  
    this.schedulerRegistry.deleteCronJob(name);  
    this.logger.warn(`job ${name} deleted!`);  
}
```

다음과 같이 `SchedulerRegistry#getCronJobs` 메서드를 사용하여 모든 크론 작업을 나열합니다:

```
getCrongs() {
    const jobs = this.schedulerRegistry.getCronJobs();
    jobs.forEach((value, key, map) => {{
        let next;
        try {
            next = value.nextDates().toDate();
        } catch (e) {
            다음 = '오류: 다음 발사 날짜가 과거입니다!';
        }
        this.logger.log(`job: ${key} -> next: ${next}`);
    });
}
```

`getCronJobs()` 메서드는 맵을 반환합니다. 이 코드에서는 맵을 반복하여 각 `CronJob`의 `nextDates()` 메서드에 액세스하려고 시도합니다. `CronJob` API에서는 작업이 이미 실행되었고 향후 실행 날짜가 없는 경우 예외를 던집니다.

동적 간격

`SchedulerRegistry#getInterval` 메서드를 사용하여 간격에 대한 참조를 얻습니다. 위와 같이 표준 생성자 주입을 사용하여 [스케줄러 레지스트리](#)를 생성합니다:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

그리고 다음과 같이 사용하세요:

```
const interval = this.schedulerRegistry.getInterval('notifications');
clearInterval(interval);
```

다음과 같이 `SchedulerRegistry#addInterval` 메서드를 사용하여 새 간격을 동적으로 생성합니다:

```
addInterval(name: 문자열, milliseconds: 숫자) {
  const
    callback = () => {
      this.logger.warn(`시간 (${milliseconds})에 ${name} 간격으로 실행 중!`);
    };

  const interval = setInterval(callback, milliseconds);
  this.schedulerRegistry.addInterval(name, interval);
}
```

이 코드에서는 표준 자바스크립트 간격을 생성한 다음 이를 `SchedulerRegistry#addInterval` 메서드에 전달합니다. 이 메서드에는 간격의 이름과 간격 자체의 두 가지 인수가 필요합니다.

다음과 같이 `SchedulerRegistry#deleteInterval` 메서드를 사용하여 명명된 간격을 삭제합니다:

```
deleteInterval(name: string) {
  this.schedulerRegistry.deleteInterval(name);
  this.logger.warn(`Interval ${name} deleted!`);
}
```

다음과 같이 `SchedulerRegistry#getIntervals` 메서드를 사용하여 모든 인터벌을 나열합니다:

```
    intervals.forEach(key => this.logger.log(`Interval: ${key}`));
}
```

동적 시간 초과

SchedulerRegistry#getTimeout 메서드를 사용하여 타임아웃에 대한 참조를 얻습니다. 위와 같이 표준 생성자 주입을 사용하여 [스케줄러 레지스트리](#)를 생성합니다:

```
constructor(private readonly schedulerRegistry: SchedulerRegistry) {}
```

그리고 다음과 같이 사용하세요:

```
const timeout = this.schedulerRegistry.getTimeout('알림');
clearTimeout(timeout);
```

다음과 같이 SchedulerRegistry#addTimeout 메서드를 사용하여 새 타임아웃을 동적으로 생성합니다:

```
addTimeout(name: 문자열, milliseconds: 숫자) {
  const
    callback = () => {
      this.logger.warn(`(${milliseconds}) 이후 실행되는 ${name} 타임
아웃!`);
    };

  const timeout = setTimeout(callback, milliseconds);
  this.schedulerRegistry.addTimeout(name, timeout);
}
```

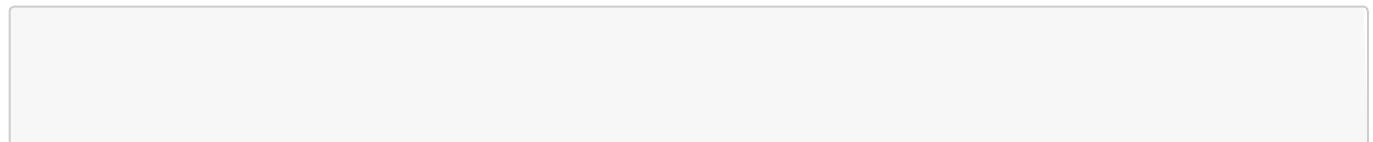
이 코드에서는 표준 자바스크립트 시간 제한을 생성한 다음 이를 SchedulerRegistry#addTimeout 메서드에 전달합니다. 이 메서드에는 타임아웃의 이름과 타임아웃 자체의 두 가지 인수가 필요합니다.

다음과 같이 SchedulerRegistry#deleteTimeout 메서드를 사용하여 지정된 타임아웃을 삭제합니다:

```
deleteTimeout(name: string) {
  this.schedulerRegistry.deleteTimeout(name);
  this.logger.warn(`타임아웃 ${name} 삭제!`);

getTimeouts() {
  const timeouts = this.schedulerRegistry.getTimeouts();
```

다음과 같이 `SchedulerRegistry#getTimeouts` 메서드를 사용하여 모든 시간 초과를 나열합니다:



```
    timeouts.forEach(key => this.logger.log(`Timeout: ${key}`));  
}
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

대기열

큐는 일반적인 애플리케이션 확장 및 성능 문제를 해결하는 데 도움이 되는 강력한 디자인 패턴입니다. 다음은 큐를 통해 해결할 수 있는 몇 가지 문제 예시입니다:

- 처리 피크를 완화하세요. 예를 들어 사용자가 리소스 집약적인 작업을 임의의 시간에 시작할 수 있는 경우 이러한 작업을 동기적으로 수행하는 대신 대기열에 추가할 수 있습니다. 그런 다음 작업자 프로세스가 제어된 방식으로 대기열에서 작업을 가져오도록 할 수 있습니다. 애플리케이션이 확장됨에 따라 새로운 큐 소비자를 쉽게 추가하여 백엔드 작업 처리를 확장할 수 있습니다.
- 모놀리식 작업을 분할하여 Node.js 이벤트 루프를 차단할 수 있습니다. 예를 들어 사용자 요청에 오디오 트랜스코딩과 같은 CPU 집약적인 작업이 필요한 경우 이 작업을 다른 프로세스에 위임하여 사용자 대면 프로세스가 응답성을 유지할 수 있도록 여유를 확보할 수 있습니다.
- 다양한 서비스에서 안정적인 커뮤니케이션 채널을 제공하세요. 예를 들어, 한 프로세스 또는 서비스에서 작업(작업)을 대기열에 추가하고 다른 서비스에서 사용할 수 있습니다. 모든 프로세스 또는 서비스에서 작업 수명 주기의 완료, 오류 또는 기타 상태 변경 시 상태 이벤트를 수신하여 알림을 받을 수 있습니다. 큐 생산자 또는 소비자가 실패해도 상태가 유지되며 노드가 다시 시작될 때 작업 처리가 자동으로 다시 시작될 수 있습니다.

Nest는 인기 있고 잘 지원되는 고성능 Node.js 기반 큐 시스템 구현인 [Bull](#) 위에 추상화/래퍼로 [@nestjs/bull](#) 패키지를 제공합니다. 이 패키지를 사용하면 애플리케이션에 Nest 친화적인 방식으로 Bull 큐를 쉽게 통합할 수 있습니다.

Bull은 Redis를 사용하여 작업 데이터를 유지하므로 시스템에 Redis가 설치되어 있어야 합니다. Redis를 기반으로 하기 때문에, 큐 아키텍처는 완전히 분산되어 플랫폼에 독립적일 수 있습니다. 예를 들어, 일부 큐 생산자와 소비자 및 리스너는 Nest에서 하나(또는 여러) 노드에서 실행하고 다른 생산자, 소비자 및 리스너는 다른 네트워크 노드의 다른 Node.js 플랫폼에서 실행하도록 할 수 있습니다.

이 장에서는 [@nestjs/bull](#) 패키지를 다룹니다. 자세한 배경과 구체적인 구현 세부 사항은 [Bull 설명서](#)를 읽어보시기 바랍니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
npm install --save @nestjs/bull bull
```

설치 프로세스가 완료되면 `BullModule`을 루트 앱모듈로 가져올 수 있습니다.

`@@파일명(앱.모듈)`

'@nestjs/common'에서 `{ Module }`을 가져오고,
'@nestjs/bull'에서 `{ BullModule }`을 가져옵니다;

`모듈({ import: [`

```
BullModule.forRoot({  
  redis: {
```

```

        호스트: 'localhost',
        포트: 6379,
    },
}),
],
})
내보내기 클래스 AppModule {}

```

`forRoot()` 메서드는 애플리케이션에 등록된 모든 큐에서 사용할 `Bull` 패키지 구성 객체를 등록하는 데 사용됩니다 (달리 명시되지 않는 한). 구성 객체는 다음과 같은 프로퍼티로 구성됩니다:

- **리미터:** `RateLimiter` - 대기열의 작업이 처리되는 속도를 제어하는 옵션입니다. 자세한 내용은 [RateLimiter](#)를 참조하세요. 선택 사항입니다.
- **redis:** `RedisOpts` - Redis 연결을 구성하는 옵션입니다. 자세한 내용은 [RedisOpts](#)를 참조하세요. 선택 사항입니다.
- **접두사:** `문자열` - 모든 대기열 키의 접두사입니다. 선택 사항입니다.
- **기본 작업 옵션:** `JobOptions` - 새 작업의 기본 설정을 제어하는 옵션입니다. 자세한 내용은 [JobOptions](#)를 참조하세요. 선택 사항입니다.
- **설정을 변경합니다:** `고급 설정` - 고급 대기열 구성 설정입니다. 이 설정은 일반적으로 변경하지 않는 것이 좋습니다. 자세한 내용은 [고급 설정을](#) 참조하세요. 선택 사항입니다.

모든 옵션은 선택 사항으로, 큐 동작을 세부적으로 제어할 수 있습니다. 이러한 옵션은 `Bull Queue` 생성자에 직접 전달됩니다. 이러한 옵션에 대한 자세한 내용은 [여기](#)를 참조하세요.

대기열을 등록하려면 다음과 같이 `BullModule.registerQueue()` 동적 모듈을 임포트합니다:

```

BullModule.registerQueue({
  name: 'audio',
});

```

정보 힌트 쉼표로 구분된 여러 개의 구성 객체를 전달하여 여러 개의 대기열을 생성합니다.

`registerQueue()` 메서드를 호출합니다.

`registerQueue()` 메서드는 큐를 인스턴스화 및/또는 등록하는 데 사용됩니다. 큐는 동일한 자격 증명을 사용하여 동일한 기본 Redis 데이터베이스에 연결하는 모듈 및 프로세스 간에 공유됩니다. 각 큐는 이름 속성에 의해 고유합니다. 큐 이름은 컨트롤러/프로바이더에 큐를 주입하기 위한 주입 토큰과 소비자 클래스 및 리스너를 큐

와 연결하기 위한 데코레이터에 대한 인수로 사용됩니다.

다음과 같이 특정 대기열에 대해 미리 구성된 옵션 중 일부를 재정의할 수도 있습니다:

```
BullModule.registerQueue({  
  name: 'audio',  
  redis: {  
    port: 6380,  
  },  
});
```

```
},
});
```

작업은 Redis에서 지속되므로, 특정 명명된 큐가 인스턴스화될 때마다(예: 앱이 시작/재시작될 때) 이전 완료되지 않은 세션에서 존재할 수 있는 모든 이전 작업을 처리하려고 시도합니다.

각 대기열에는 하나 또는 여러 개의 프로듀서, 소비자, 리스너가 있을 수 있습니다. 소비자는 특정 순서로 큐에서 작업을 검색합니다: FIFO(기본값), LIFO 또는 우선순위에 따라 검색합니다. 큐 처리 순서 제어에 대해서는 [여기에 서 설명합니다.](#)

명명된 구성

큐가 여러 개의 서로 다른 Redis 인스턴스에 연결되는 경우, 명명된 구성이라는 기술을 사용할 수 있습니다. 이 기능을 사용하면 지정된 키 아래에 여러 구성을 등록한 다음 큐 옵션에서 참조할 수 있습니다.

예를 들어, 애플리케이션에 등록된 몇 개의 큐에서 기본 인스턴스 외에 추가로 사용하는 Redis 인스턴스가 있다고 가정하면 다음과 같이 구성을 등록할 수 있습니다:

```
BullModule.forRoot('alternative-config', {
  redis: {
    포트: 6381,
  },
});
```

위의 예에서 'alternative-config'는 구성 키일 뿐입니다(임의의 문자열일 수 있음). 이제 등록큐(() 옵션

객체에서 이 구성을 가리킬 수 있습니다:

```
BullModule.registerQueue({
  configKey: 'alternative-config',
  name: 'video'
});
```

프로듀서

작업 생산자는 대기열에 작업을 추가합니다. 생산자는 일반적으로 애플리케이션 서비스(네스트 공급자)입니다.

큐에 작업을 추가하려면 먼저 다음과 같이 큐를 서비스에 주입합니다:

```
'@nestjs/common'에서 { Injectable }을 임포트하고,  
'bull'에서 { Queue }를 임포트합니다;  
'@nestjs/bull'에서 { InjectQueue }를 가져옵니다;  
  
@Injectable()  
내보내기 클래스 AudioService {
```

```
생성자(@InjectQueue('audio') private audioQueue: Queue) {}  
}
```

정보 힌트 `@InjectQueue()` 데코레이터는 이름에 제공된 대로 큐를 식별합니다.

`registerQueue()` 메서드 호출(예: '`audio`').

이제 큐의 `add()` 메서드를 호출하여 사용자 정의 작업 객체를 전달하여 작업을 추가합니다. 작업은 직렬화 가능한 JavaScript 객체로 표현됩니다(Redis 데이터베이스에 저장되는 방식이므로). 전달하는 작업의 모양은 임의적이므로 작업 객체의 의미를 나타내는 데 사용하십시오.

```
const job = await this.audioQueue.add({  
  foo: 'bar',  
});
```

명명된 작업

작업에는 고유한 이름이 있을 수 있습니다. 이를 통해 특정 이름의 작업만 처리하는 전문 [소비자를](#) 만들 수 있습니다.

```
const job = await this.audioQueue.add('transcode', {  
  foo: 'bar',  
});
```

경고 경고 명명된 작업을 사용할 때는 대기열에 추가된 각 고유 이름에 대해 프로세서를 만들어야 하며, 그렇지 않으면 대기열에서 해당 작업에 대한 프로세서가 없다는 불만을 표시합니다. 명명된 작업 사용에 대한 자세한 내용은 [여기를 참조하세요](#).

작업 옵션

작업에는 추가 옵션을 연결할 수 있습니다. `Queue.add()` 메서드에서 `작업` 인수 뒤에 옵션 개체를 전달합니다.

작업 옵션 속성은 다음과 같습니다:

- **우선순위:** 숫자 - 선택적 우선순위 값입니다. 범위는 1(가장 높은 우선순위)에서 MAX_INT(가장 낮은 우선순위)까지입니다. 우선순위를 사용하면 성능에 약간의 영향을 미치므로 주의해서 사용하세요. **지연:** 숫자 - 이 작업을 처리할 수 있을 때까지 대기할 시간(밀리초)입니다. 정확한 지연을 위해 서버와 클라이언트의 시

계가 모두 동기화되어 있어야 합니다.

- 시도: 수 - 작업이 완료될 때까지 시도한 총 시도 횟수입니다.
- 반복: 반복 옵션 - 크론 사양에 따라 작업을 반복합니다. 백오프: 숫자 | 백오프 옵션 - 작업 실패
- 시 자동 재시도를 위한 백오프 설정입니다. [백오프옵션을](#) 참조하세요.
- lifo: 부울 - 참이면 작업을 대기열의 왼쪽 끝이 아닌 오른쪽 끝에 추가합니다(기본값은 거짓). timeout:
- 숫자 - 시간 초과 오류로 작업이 실패할 때까지의 시간(밀리초) jobId: 숫자 | 문자열 - 작업 ID 재정의 - 기본적으로 작업 ID는 고유 정수이지만 이 설정을 사용하여 이를 재정의할 수 있습니다. 이 옵션을 사용하는 경우 jobId가 고유한지 확인하는 것은 사용자의 책임입니다. 이미 존재하는 ID로 작업을 추가하려고 하면 추가되지 않습니다.

- `removeOnComplete: boolean | 숫자` - 참이면 작업이 성공적으로 완료되면 작업을 제거합니다.
숫자는 보관할 작업의 양을 지정합니다. 기본 동작은 완료된 세트에 작업을 유지하는 것입니다.
- `removeOnFail: boolean | 숫자` - 참이면 모든 시도 후 실패하면 작업을 제거합니다. 숫자는 유지할 작업의 양을 지정합니다. 기본 동작은 실패한 세트에 작업을 유지하는 것입니다.
- `stackTraceLimit: 숫자` - 스택트레이스에 기록될 스택 추적 줄의 양을 제한합니다.

다음은 작업 옵션으로 작업을 사용자 지정하는 몇 가지 예입니다. 작

업 시작을 지연시키려면 `지연` 구성 속성을 사용합니다.

```
const job = await this.audioQueue.add(  
  {  
    foo: 'bar',  
  },  
  { delay: 3000 }, // 3초 지연  
);
```

대기열의 오른쪽 끝에 작업을 추가하려면(작업을 선입선출(LIFO)로 처리하려면), 다음과 같이 설정합니다.

속성을 `true`로 설정합니다.

```
const job = await this.audioQueue.add(  
  {  
    foo: 'bar',  
  },  
  { lifo: true },  
);
```

작업의 우선순위를 지정하려면 `우선순위` 속성을 사용하세요.

```
const job = await this.audioQueue.add(  
  {  
    foo: 'bar',  
  },  
  { 우선순위: 2 },  
);
```

소비자

컨슈머는 큐에 추가된 작업을 처리하거나 큐의 이벤트를 수신하는 메서드를 정의하는 클래스 또는 둘 다를 정의하는 클래스입니다. 다음과 같이 `@Processor()` 데코레이터를 사용하여 소비자 클래스를 선언합니다:

```
'@nestjs/bull'에서 { Processor }를 가져옵니다;
```

```
@Processor('audio')
내보내기 클래스 AudioConsumer {}
```

정보 힌트 소비자는 [공급자로](#) 등록해야 [@nestjs/bull](#) 패키지가 소비자를 받을 수 있습니다.

여기서 데코레이터의 문자열 인수(예: `'audio'`)는 클래스 메서드와 연결할 대기열의 이름입니다.

소비자 클래스 내에서 핸들러 메서드를 [@Process\(\)](#)로 장식하여 작업 핸들러를 선언합니다.

데코레이터.

```
'@nestjs/bull'에서 { Processor, Process }를 가져오고,
'bull'에서 { Job }을 가져옵니다;
```

```
@Processor('audio')
내보내기 클래스 AudioConsumer {
  @Process()
  async transcode(job: Job<unknown>) {
    let progress = 0;
    for (i = 0; i < 100; i++) {
      await doSomething(job.data);
      progress += 1;
      await job.progress(progress);
    }
    반환 {};
  }
}
```

장식된 메서드(예: `transcode()`)는 워커가 유휴 상태이고 큐에 처리할 작업이 있을 때마다 호출됩니다. 이 핸들러 메서드는 [작업](#) 객체를 유일한 인수로 받습니다. 처리기 메서드가 반환하는 값은 작업 객체에 저장되며 나중에 완료된 이벤트의 리스너 등에서 액세스할 수 있습니다.

[작업](#) 객체에는 해당 상태와 상호작용할 수 있는 여러 메서드가 있습니다. 예를 들어 위 코드는 `progress()` 메서드를 사용하여 작업의 진행 상황을 업데이트합니다. 전체 `Job` 객체 API 참조는 [여기를](#) 참조하세요.

아래와 같이 [@Process\(\)](#) 데코레이터에 해당 [이름](#)을 전달하여 작업 처리기 메서드가 특정 유형의 작업(특정 [이름](#)의 작업)만 처리하도록 지정할 수 있습니다. 지정된 소비자 클래스에는 각 작업 유형([이름](#))에 해당하는 여러 개의 [@Process\(\)](#) 핸들러를 가질 수 있습니다. 명명된 작업을 사용하는 경우 각 이름에 해당하는 핸들러가 있어야 합니다.

```
@Process('트랜스코드')
async transcode(job: Job<unknown>) { ... }
```

경고 경고 동일한 대기열에 대해 여러 소비자를 정의할 때 `@Process({{ '{' }} 동시성: 1 {{ '}' }})`의 `동시성` 옵션이 적용되지 않습니다. 최소 `동시성은` 정의된 소비자 수와 일치합니다. 이는 `@Process()` 핸들러가 다른 이름을 사용하여 명명된 작업을 처리하는 경우에도 적용됩니다.

요청 범위가 지정된 소비자

소비자가 요청 범위로 플래그가 지정되면(여기에서 주입 범위에 대해 자세히 알아보세요), 각 작업에 대해 클래스의 새 인스턴스가 독점적으로 생성됩니다. 인스턴스는 작업이 완료된 후 가비지 수집됩니다.

```
@Processor({
  name: 'audio',
  scope: Scope.REQUEST,
})
```

요청 범위가 지정된 소비자 클래스는 동적으로 인스턴스화되고 단일 작업으로 범위가 지정되므로 표준 접근 방식을 사용하여 생성자를 통해 `JOB_REF`를 주입할 수 있습니다.

```
constructor(@Inject(JOB_REF) jobRef: Job) {
  console.log(jobRef);
}
```

정보 힌트 `JOB_REF` 토큰은 `@nestjs/bull` 패키지에서 가져옵니다.

이벤트 리스너

`Bull`은 대기열 및/또는 작업 상태 변경이 발생할 때 유용한 이벤트 세트를 생성합니다. `Nest`는 표준 이벤트의 핵심 집합을 구독할 수 있는 데코레이터 세트를 제공합니다. 이러한 데코레이터는 `@nestjs/bull` 패키지에서 내보낼 수 있습니다.

이벤트 리스너는 `소비자` 클래스 내에서 선언해야 합니다(즉, `@Processor()` 데코레이터로 장식된 클래스 내에서). 이벤트를 수신하려면 아래 표의 데코레이터 중 하나를 사용하여 이벤트에 대한 핸들러를 선언하세요. 예를 들어 작업이 `오디오` 대기열에서 활성 상태가 될 때 발생하는 이벤트를 수신하려면 다음 구문을 사용합니다:

'@nestjs/bull'에서 { Processor, Process, OnQueueActive }를 임포트하고,
'bull'에서 { Job }을 임포트합니다;

```
@Processor('audio')
내보내기 클래스 AudioConsumer {

    OnQueueActive()
    onActive(job: Job) {
        콘솔 로그(
```

데이터로 \${job.name} 유형의 \${job.id} 작업을 처리 중입니다.

```

${job.data}...`  

    );  

}  

...

```

Bull은 분산(다중 노드) 환경에서 작동하므로 이벤트 로컬리티라는 개념을 정의합니다. 이 개념은 이벤트가 단일 프로세스 내에서만 트리거되거나 다른 프로세스의 공유 큐에서 트리거될 수 있음을 인식합니다. 로컬 이벤트는 로컬 프로세스의 대기열에서 작업 또는 상태 변경이 트리거될 때 생성되는 이벤트입니다. 즉, 이벤트 생산자와 소비자가 단일 프로세스에 로컬인 경우 대기열에서 발생하는 모든 이벤트는 로컬 이벤트입니다.

대기열이 여러 프로세스에서 공유되는 경우 글로벌 이벤트가 발생할 가능성이 있습니다. 한 프로세스의 수신기가 다른 프로세스에 의해 트리거된 이벤트 알림을 수신하려면 글로벌 이벤트에 등록해야 합니다.

이벤트 핸들러는 해당 이벤트가 발생할 때마다 호출됩니다. 핸들러는 아래 표에 표시된 서명으로 호출되며, 이벤트와 관련된 정보에 액세스할 수 있습니다. 아래에서 로컬 이벤트 핸들러 서명과 글로벌 이벤트 핸들러 서명 간의 주요 차이점에 대해 설명합니다.

로컬 이벤트 리스너	글로벌 이벤트 리스너	핸들러 메서드 서명/시기 하고
핸들러(오류: 오류) - 오류		
온큐에러()	온글로벌 큐 에러()	오류가 발생했습니다. 오류에 는 트리거 오류가 포함되어 있습니다.
핸들러(jobId: 숫자 문자열) - 작 업자가 유휴 상태인 즉시 작업이 처리 되기를 기다리는 중입니다. jobId에는 이 상태에 진입한 작업의 ID가 포함되 어 있습니다.		
@OnQueueWaiting()	온글로벌 큐 대기()	
온큐큐액티브()	온글로벌큐큐액티브()	
핸들러(작업: Job) - 작업이 시작되었습 니다.		
@OnQueueStalled()	OnGlobalQueueStalled()	
핸들러(job: 작업) - 작업 작업이 중		

단된 것으로 표시되었습니다. 이벤트 루프를 충돌하거나

일시 중지하는 작업자를 디버깅할 때 유용합니다.

온큐어프로그레스()

온글로벌 큐 진행률()

핸들러(job: 작업, 진행률: 숫자)

- 작업의 진행률이 진행률 값으로

업데이트되었습니다.

OnQueueCompleted() @OnGlobalQueueCompleted()

핸들러(job: Job, result: any)

결과와 함께 작업이 성공적으로 완료되

었습니다.

@OnQueueFailed()

온글로벌 큐 실패()

핸들러(job: 작업, err: 오류)

이유 오류로 작업이 실패했습니다.

@OnQueuePaused()

온글로벌 큐 일시정지()

핸들러() 큐가

일시 중지되었습니다.

대기열 재개()

온글로벌 큐 재개()

핸들러(작업: 작업) 큐에 다음이 있습니다.

가 재개되었습니다.

@OnQueueCleaned()

온글로벌 큐 청소()

핸들러(jobs: Job[], type: 문자열)
대기열에서 오래된 작업이 정리되었습니다. jobs는 정리된 작업의 배열이고 type은 정리된 작업의 유형입니다.

@OnQueueDrained()

OnGlobalQueueDrained()

핸들러() 대기열이 대기 중인 모든 작업을 처리할 때마다 발생합니다(아직 처리되지 않은 지역된 작업이 있을 수 있음에도 불구하고).

핸들러(작업: Job) 작업이 성공적으로 제거되었습니다.

글로벌 이벤트를 수신할 때 메서드 서명은 로컬 버전과 약간 다를 수 있습니다. 특히, 로컬 버전에서 작업 객체를 수신하는 메서드 서명은 글로벌 버전에서 jobId(숫자)를 수신합니다. 이러한 경우 실제 작업 객체에 대한 참조를 얻으려면 Queue#getJob 메서드를 사용하세요. 이 호출은 대기 상태여야 하므로 핸들러를 비동기 상태로 선언해야 합니다. 예를 들어

온글로벌큐큐완료()

```
async onGlobalCompleted(jobId: number, result: any) {
  const job = await this.immediateQueue.getJob(jobId);
  console.log('(Global) on completed: job ', job.id, ' -> result: ', result);
}
```

정보 힌트 큐 객체에 접근하려면(getJob()) 호출을 위해) 당연히 큐 객체를 주입해야 합니다. 또한 큐를 주입하는 모듈에 큐가 등록되어 있어야 합니다.

특정 이벤트 리스너 데코레이터 외에도 일반 @OnQueueEvent() 데코레이터를 BullQueueEvents 또는

`BullQueueGlobalEvents` 열거형과 함께 사용할 수도 있습니다. 이벤트에 대한 자세한 내용은 [여기](#)를 참조하세요.

대기열 관리

대기열에는 일시 중지 및 재개, 다양한 상태의 작업 수 검색 등 여러 가지 관리 기능을 수행할 수 있는 API가 있습니다. 전체 큐 API는 [여기](#)에서 확인할 수 있습니다. 아래 일시 중지/재개 예시와 같이 큐 개체에서 직접 이러한 메서드를 호출할 수 있습니다.

`pause()` 메서드 호출로 큐를 일시 중지합니다. 일시 중지된 큐는 재개될 때까지 새 작업을 처리하지 않지만 처리 중인 현재 작업은 완료될 때까지 계속됩니다.

오디오 큐브. 일시정지()를 기다립니다;

일시 중지된 대기열을 다시 시작하려면 다음과 같이 `resume()` 메서드를 사용합니다:

오디오 큐를 기다립니다;

별도의 프로세스

작업 처리기는 별도의 (포크된) 프로세스([소스](#))에서 실행할 수도 있습니다. 여기에는 몇 가지 장점이 있습니다:

- 프로세스는 샌드박스가 적용되므로 충돌이 발생해도 작업자에게 영향을 미치지 않습니다.
- 지 않습니다. 대기열에 영향을 주지 않고 차단 코드를 실행할 수 있습니다(작업이 중단되지 않음).
- 이 중단되지 않음). 멀티코어 CPU를 훨씬 더 잘 활용합니다.

redis에 대한 연결이 줄어듭니다.

`@@파일명(앱.모듈)`

```
'@nestjs/common'에서 { Module }을 가져오고,  
'@nestjs/bull'에서 { BullModule }을 가져오고,  
'path'에서 { join }을 가져옵니다;
```

```
모듈({ import: [  
    BullModule.registerQueue({  
        name: 'audio',  
        프로세서입니다: [join(_dirname, 'processor.js')],  
    }),  
    ],  
})
```

내보내기 클래스 `AppModule {}`

함수가 포크된 프로세스에서 실행되고 있기 때문에 의존성 주입(및 IoC 컨테이너)을 사용할 수 없다는 점에 유의하세요. 즉, 프로세서 함수는 필요한 외부 종속성의 모든 인스턴스를 포함(또는 생성)해야 합니다.

`@@파일명(프로세서)`

```
'bull'에서 { Job, DoneCallback }을 가져옵니다;  
  
export default function (job: Job, cb: DoneCallback) {  
    console.log(`[${process.pid}] ${JSON.stringify(job.data)}`);  
    cb(null, 'It works');  
}
```

비동기 구성

정적이 아닌 비동기적으로 **불** 옵션을 전달하고 싶을 수도 있습니다. 이 경우 비동기 구성을 처리하는 여러 가지 방법을 제공하는 `forRootAsync()` 메서드를 사용하세요. 마찬가지로 큐 옵션을 비동기적으로 전달하려면 `registerQueueAsync()` 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
BullModule.forRootAsync({
  useFactory: () => ({
    redis: {
      호스트: 'localhost', 포
      트: 6379,
    },
  }),
});
```

저희 팩토리는 다른 [비동기 공급자처럼](#) 동작합니다(예: [비동기일](#) 수 있고, [주입을](#) 통해 종속성을 주입할 수 있습니다).

```
BullModule.forRootAsync({
  import: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    redis: {
      host: configService.get('QUEUE_HOST'),
      port: configService.get('QUEUE_PORT'),
    },
  }),
  주입합니다: [구성 서비스],
});
```

또는 `useClass` 구문을 사용할 수도 있습니다:

```
BullModule.forRootAsync({
  useClass: BullConfigService,
});
```

위의 구성은 `BullModule` 내부에 `BullConfigService`를 인스턴스화하고 이를 사용하여 `createSharedConfiguration()`을 호출하여 옵션 객체를 제공합니다. 이는 아래 그림과 같이 `BullConfigService`가 `SharedBullConfigurationFactory` 인터페이스를 구현해야 한다는 것을 의미합니다:

```
@Injectable()
BullConfigService 클래스는 SharedBullConfigurationFactory를 구현합니다 {
    createSharedConfiguration(): BullModuleOptions {
        반환 {
            redis: {
```

```
    호스트: 'localhost',
    포트: 6379,
  },
};

}
```

BullModule 내부에 BullConfigService를 생성하지 않고 다른 모듈에서 가져온 프로바이더를 사용하려면 useExisting 구문을 사용할 수 있습니다.

```
BullModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

이 구조는 useClass와 동일하게 작동하지만 한 가지 중요한 차이점이 있습니다. BullModule은 가져온 모듈을 조회하여 새 구성 서비스를 인스턴스화하는 대신 기존 구성 서비스를 재사용합니다.

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

로거

Nest에는 애플리케이션 부트스트래핑 및 잡힌 예외 표시(예: 시스템 로깅)와 같은 여러 상황에서 사용되는 텍스트 기반 로거가 내장되어 있습니다. 이 기능은 [@nestjs/common](#) 패키지의 `Logger` 클래스를 통해 제공됩니다. 다음 중 하나를 포함하여 로깅 시스템의 동작을 완전히 제어할 수 있습니다:

- 로깅을 완전히 비활성화합니다.
- 로그 세부 수준 지정(예: 오류, 경고, 디버그 정보 표시 등)
- 기본 로거의 타임스탬프 재정의(예: 날짜 형식으로 ISO8601 표준 사용)
- 기본 로거를 완전히 재정의합니다.
- 기본 로거를 확장하여 사용자 정의하기
- 종속성 주입을 사용하여 애플리케이션 작성 및 테스트를 간소화하세요.

기본 제공 로거를 사용하거나 사용자 정의 구현을 만들어 애플리케이션 수준 이벤트와 메시지를 직접 기록할 수도 있습니다.

고급 로깅 기능을 사용하려면 [Winston](#)과 같은 Node.js 로깅 패키지를 사용하여 완전히 사용자 정의된 프로덕션 등급 로깅 시스템을 구현할 수 있습니다.

기본 사용자 지정

로깅을 비활성화하려면 `NestFactory.create()` 메서드의 두 번째 인수로 전달되는 (선택 사항) Nest 애플리케이션 옵션 객체에서 `logger` 속성을 `false`로 설정합니다.

```
const app = await NestFactory.create(AppModule, {
  logger: false,
});
await app.listen(3000);
```

특정 로깅 수준을 활성화하려면 다음과 같이 표시할 로그 수준을 지정하는 문자열 배열로 `logger` 속성을 설정합니다:

```
const app = await NestFactory.create(AppModule, {
  logger: ['error', 'warn'],
});
await app.listen(3000);
```

배열의 값은 '로그', '오류', '경고', '디버그', '상세' 중 어떤 조합이든 사용할 수 있습니다.

정보 힌트 기본 로거의 메시지에서 색상을 사용하지 않으려면 `NO_COLOR` 환경 변수를 비어 있지 않은 문자열로 설정하세요.

사용자 지정 구현

로거 프로퍼티의 값을 `LoggerService` 인터페이스를 구현하는 객체로 설정하여 Nest에서 시스템 로깅에 사용할 사용자 정의 로거 구현을 제공할 수 있습니다. 예를 들어, 다음과 같이 Nest에 내장된 전역 JavaScript 콘솔 객체(`LoggerService` 인터페이스를 구현)를 사용하도록 지시할 수 있습니다:

```
const app = await NestFactory.create(AppModule, {
  logger: console,
});
await app.listen(3000);
```

나만의 커스텀 로거를 구현하는 방법은 간단합니다. 각 메서드를 구현하기만 하면 됩니다.

`LoggerService` 인터페이스와 같습니다.

'@nestjs/common'에서 `{ LoggerService }` 임포트; 내보내기

클래스 `MyLogger`는 `LoggerService`를 구현합니다.

```
/**
 * '로그' 수준의 로그를 작성합니다.
 */
log(message: any, ...optionalParams: any[]) {}

/**
 * '오류' 수준 로그를 작성합니다.
 */
error(message: any, ...optionalParams: any[]) {}

/**
 * '경고' 수준 로그를 작성합니다.
 */
warn(message: any, ...optionalParams: any[]) {}

/**
 * '디버그' 수준 로그를 작성합니다.
 */
debug?(message: any, ...optionalParams: any[]) {}

/**
 * '상세' 수준 로그를 작성합니다.
 */
verbose?(message: any, ...optionalParams: any[])
}
```

그런 다음 Nest 애플리케이션 옵션 객체의 `logger` 속성을 통해 `MyLogger` 인스턴스를 제공할 수 있습니다.

```
const app = await NestFactory.create(AppModule, {  
  logger: new MyLogger(),
```

```
});  
await app.listen(3000);
```

이 기법은 간단하지만 `MyLogger` 클래스에 대한 의존성 주입을 활용하지 않습니다. 이는 특히 테스트 시 몇 가지 문제를 야기할 수 있으며 `MyLogger`의 재사용성을 제한할 수 있습니다. 더 나은 해결 방법은 아래의 [의존성 주입](#) 섹션을 참조하세요.

내장 로거 확장

로거를 처음부터 작성하는 대신 기본 제공 로거를 확장하여 요구 사항을 충족할 수 있습니다.

`ConsoleLogger` 클래스 및 기본 구현의 선택된 동작을 재정의합니다.

```
'@nestjs/common'에서 { ConsoleLogger } 임포트; 내보내기
```

```
클래스 MyLogger extends ConsoleLogger {  
    error(message: any, stack?: string, context?: string) {  
        // 여기에 맞춤형 로직을 추가하세요;  
    }  
}
```

아래의 [애플리케이션 로깅을 위한 로거 사용](#) 섹션에 설명된 대로 기능 모듈에서 이러한 확장 로거를 사용할 수 있습니다.

위의 [사용자 정의 구현](#) 섹션에 표시된 것처럼 애플리케이션 옵션 객체의 `logger` 속성을 통해 인스턴스를 전달하거나 아래의 [종속성 주입](#) 섹션에 표시된 기술을 사용하여 Nest가 시스템 로깅에 확장 로거를 사용하도록 지시할 수 있습니다. 이 경우 위의 샘플 코드에 표시된 것처럼 `super`를 호출하여 특정 로그 메서드 호출을 부모(내장) 클래스에 위임하여 Nest가 예상되는 내장 기능을 사용할 수 있도록 해야 합니다.

종속성 주입

고급 로깅 기능을 사용하려면 종속성 주입을 활용하고 싶을 것입니다. 예를 들어 로거에 `ConfigService`를 주입하여 사용자 정의한 다음 다른 컨트롤러 및/또는 프로바이더에 사용자 정의 로거를 주입할 수 있습니다. 사용자 정의 로거에 종속성 주입을 사용하려면 `LoggerService`를 구현하는 클래스를 생성하고 해당 클래스를 일부 모듈에 프로바이더로 등록하세요. 예를 들어 다음과 같이 할 수 있습니다.

. 이전 섹션에 표시된 대로 기본 제공 `ConsoleLogger`를 확장하거나 완전히 재정의하는 `MyLogger` 클래스

를 정의합니다. `LoggerService` 인터페이스를 구현해야 합니다.

. 아래와 같이 `LoggerModule`을 생성하고 해당 모듈에서 `MyLogger`을 제공합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
```

```
'./my-logger.service'에서 { MyLogger }를 가져옵니다;
```

```
모듈({
```

```
  공급자: [MyLogger],
```

```
내보내기: [MyLogger],  
})  
내보내기 클래스 LoggerModule {}
```

이 구성을 사용하면 이제 다른 모듈에서 사용할 수 있도록 사용자 정의 로거를 제공하게 됩니다. `MyLogger` 클래스는 모듈의 일부이므로 의존성 주입을 사용할 수 있습니다(예: `컨피그서비스` 주입). Nest에서 시스템 로깅(예: 부트 스트랩 및 오류 처리)에 사용할 수 있도록 이 사용자 정의 로거를 제공하는 데 필요한 기술이 한 가지 더 있습니다

애플리케이션 인스턴스화(`NestFactory.create()`)는 모듈의 컨텍스트 외부에서 발생하기 때문에 초기화의 일반적인 의존성 주입 단계에 참여하지 않습니다. 따라서 적어도 하나의 애플리케이션 모듈이 `LoggerModule`을 임포트하여 Nest가 `MyLogger` 클래스의 싱글톤 인스턴스를 인스턴스화하도록 트리거하도록 해야 합니다.

그런 다음 Nest에 다음과 같은 구성으로 동일한 싱글톤 인스턴스를 사용하도록 지시할 수 있습니다:

```
const app = await NestFactory.create(AppModule, {  
  bufferLogs: true,  
});  
app.useLogger(app.get(MyLogger));  
await app.listen(3000);
```

정보 참고 위의 예제에서는 사용자 정의 로거(이 경우 `MyLogger`)가 첨부되고 애플리케이션 초기화 프로세스가 완료되거나 실패할 때까지 모든 로그가 버퍼링되도록 `bufferLogs`를 `true`로 설정했습니다. 초기화 프로세스가 실패하면 Nest는 원래 `ConsoleLogger`로 풀백하여 보고된 오류 메시지를 인쇄합니다.

또한, 자동 플러시 로그를 `거짓`(기본값 참)으로 설정하여 로그를 수동으로 플러시할 수 있습니다 (`Logger#flush()` 메서드 사용).

여기서는 `NestApplication` 인스턴스의 `get()` 메서드를 사용하여 `MyLogger` 객체의 싱글톤 인스턴스를 검색합니다. 이 기술은 본질적으로 Nest에서 사용할 로거 인스턴스를 "주입"하는 방법입니다. `app.get()` 호출은 `MyLogger`의 싱글톤 인스턴스를 검색하며, 위에서 설명한 대로 해당 인스턴스가 다른 모듈에 먼저 주입되는지에 따라 달라집니다.

기능 클래스에 이 `MyLogger` 공급자를 삽입하여 Nest 시스템 로깅과 애플리케이션 로깅 모두에서 일관된 로깅 동작을 보장할 수도 있습니다. 자세한 내용은 아래에서 [애플리케이션 로깅에 로거 사용 및 사용자 정의 로거 삽입하기](#)를 참조하세요.

애플리케이션 로깅에 로거 사용

위의 몇 가지 기술을 결합하여 Nest 시스템 로깅과 자체 애플리케이션 이벤트/메시지 로깅 모두에서 일관된 동작과 형식을 제공할 수 있습니다.

좋은 방법은 각 서비스에서 `@nestjs/common`의 `Logger` 클래스를 인스턴스화하는 것입니다. 다음과 같이 `Logger` 생성자에서 서비스 이름을 `컨텍스트` 인수로 제공할 수 있습니다:

```
'@nestjs/common'에서 { 로거, 인젝터블 }을 임포트합니다;
```

주입 가능() 클래스

```
MyService {
    비공개 읽기 전용 로거 = 새로운 로거(MyService.name);

    doSomething() {
        this.logger.log('작업 중...');
    }
}
```

기본 로거 구현에서 컨텍스트는 아래 예제의 `NestFactory`와 같이 대괄호 안에 인쇄됩니다:

Nest] 19096	12/08/2019, 7:12:59 AM	[NestFactory]Nest 애플리케이션 시
		작 중...

`app.useLogger()`를 통해 사용자 정의 로거를 제공하면 실제로 Nest 내부에서 사용하게 됩니다. 즉, 코드가 구현에 구애받지 않고 유지되며, `app.useLogger()`를 호출하여 기본 로거를 사용자 정의 로거로 쉽게 대체할 수 있습니다.

이렇게 하면 이전 섹션의 단계를 따라 `app.useLogger(app.get(MyLogger))`를 호출하면 `MyService`에서 `MyLogger.log()`를 다음과 같이 호출하면 `MyLogger` 인스턴스에서 메서드 `log`를 호출하게 됩니다.

이 방법이 대부분의 경우에 적합합니다. 그러나 사용자 지정 메서드 추가 및 호출과 같은 추가 사용자 지정이 필요한 경우 다음 섹션으로 이동하세요.

사용자 지정 로거 삽입

시작하려면 다음과 같은 코드를 사용하여 기본 제공 로거를 확장합니다. 각 기능 모듈에 고유한 `MyLogger` 인스턴스를 갖도록 하기 위해 일시적인 범위를 지정하는 `ConsoleLogger` 클래스에 대한 구성 메타데이터로 범위 옵션을 제공합니다. 이 예제에서는 개별 `ConsoleLogger` 메서드(`로그()`, `경고()` 등)를 확장하지 않지만 원하는 경우 확장할 수 있습니다.

```
'@nestjs/common'에서 { Injectable, Scope, ConsoleLogger } import;

@Injectable({ scope: Scope.TRANSIENT })
export class MyLogger extends ConsoleLogger {
    customLog() {
        this.log('고양이에게 밥을 주세요!');
    }
}
```

다음으로, 다음과 같은 구조의 [LoggerModule](#)을 생성합니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./my-logger.service'에서 { MyLogger }를 가져옵니다;
```

```
모듈({
  제공자: [MyLogger], 내보내기
  : [MyLogger],
})
내보내기 클래스 LoggerModule {}
```

다음으로 `LoggerModule`을 기능 모듈로 가져옵니다. 기본 로거를 확장했기 때문에 `setContext` 메서드를 편리하게 사용할 수 있습니다. 이제 다음과 같이 컨텍스트 인식 커스텀 로거를 사용할 수 있습니다:

```
'@nestjs/common'에서 { Injectable }을 임포트하고,
'./my-logger.service'에서 { MyLogger }를 임포트합니다;

@Injectable()
내보내기 클래스 CatsService {
  비공개 읽기 전용 고양이: Cat[] = [];

  constructor(private myLogger: MyLogger) {
    // 일시적인 범위로 인해 CatsService에는 고유한 MyLogger 인스턴스가 있습니다,
    // 따라서 여기서 컨텍스트를 설정해도 다른 서비스의 다른 인스턴스에는 영향을 미치지 않습니다.
    this.myLogger.setContext('CatsService');
  }

  findAll(): Cat[] {
    // 모든 기본 메소드를 호출할 수 있습니다
    this.myLogger.warn('고양이를 반환하려고 합니다
    !');

    // 그리고 사용자 정의 메서드
    this.myLogger.customLog();
    반환 this.cats;
  }
}
```

마지막으로, 아래와 같이 `메인.ts` 파일에서 사용자 정의 로거의 인스턴스를 사용하도록 Nest에 지시합니다. 물론 이 예제에서는 `log()`, `warn()` 등과 같은 로거 메서드를 확장하여 로거 동작을 실제로 사용자 정의하지 않았으므로 이 단계는 실제로 필요하지 않습니다. 하지만 해당 메서드에 사용자 정의 로직을 추가하고 Nest가 동일한 구현을 사용하도록 하려면 이 단계가 필요합니다.

```
const app = await NestFactory.create(AppModule, {
  bufferLogs: true,
});
app.useLogger(new MyLogger());
await app.listen(3000);
```

정보 힌트 또는 `bufferLogs`를 `true`로 설정하는 대신 `logger: false` 명령어를 사용하여 일시적으로 로거를 비활성화할 수 있습니다. `NestFactory.create`에 `logger: false`를 제공하면 `useLogger`를 호출할 때까지 아무 것도 기록되지 않으므로 중요한 초기화 오류를 놓칠 수 있다는 점에 유의하세요. 초기 메시지 중 일부가 기본 로거로 기록되어도 괜찮다면 `logger: false` 옵션을 생략할 수 있습니다.

외부 로거 사용

프로덕션 애플리케이션에는 고급 필터링, 서식 지정, 중앙 집중식 로깅 등 특정 로깅 요구 사항이 있는 경우가 많습니다. Nest의 기본 제공 로거는 Nest 시스템 동작을 모니터링하는데 사용되며 개발 중 기능 모듈에서 기본 형식의 텍스트 로깅에도 유용할 수 있지만, 프로덕션 애플리케이션은 [Winston](#)과 같은 전용 로깅 모듈을 활용하는 경우가 많습니다. 다른 표준 Node.js 애플리케이션과 마찬가지로 Nest에서도 이러한 모듈을 최대한 활용할 수 있습니다.

쿠키

HTTP 쿠키는 사용자의 브라우저에 저장되는 작은 데이터 조각입니다. 쿠키는 웹사이트가 상태 정보를 기억할 수 있는 신뢰할 수 있는 메커니즘으로 설계되었습니다. 사용자가 웹사이트를 다시 방문하면 요청과 함께 쿠키가 자동으로 전송됩니다.

Express와 함께 사용(기본값)

먼저 [필요한 패키지](#)(TypeScript 사용자의 경우 해당 유형)를 설치합니다:

```
$ npm i 쿠키 파서  
$ npm i -D @타입스/쿠키-파서
```

설치가 완료되면 [쿠키 파서](#) 미들웨어를 글로벌 미들웨어로 적용합니다(예: [main.ts](#) 파일).

```
'쿠키 파서'에서 쿠키 파서로 *를 가져옵니다;  
// 초기화 파일 어딘가에 앱.사용(쿠키파서());
```

[쿠키파서](#) 미들웨어에 몇 가지 옵션을 전달할 수 있습니다:

- [비밀](#) 쿠키를 서명하는 데 사용되는 문자열 또는 배열입니다. 이 옵션은 선택 사항이며 지정하지 않으면 서명된 쿠키를 구문 분석하지 않습니다. 문자열이 제공되면 이 문자열이 비밀로 사용됩니다. 배열을 제공하면 각 비밀을 순서대로 사용하여 쿠키의 서명을 해제하려고 시도합니다.
- 옵션은 두 번째 옵션으로 [cookie.parse](#)에 전달되는 객체입니다. 자세한 내용은 [쿠키](#)를 참조하세요.

미들웨어는 요청에 대한 [쿠키](#) 헤더를 구문 분석하여 쿠키 데이터를 [req.cookies](#) 속성으로, 비밀이 제공된 경우 [req.signedCookies](#) 속성으로 노출합니다. 이러한 속성은 쿠키 이름과 쿠키 값의 이름 값 쌍입니다.

비밀이 제공되면 이 모듈은 서명된 쿠키 값의 서명을 해제하고 유효성을 검사한 후 해당 이름 값 쌍을 [req.cookies](#)에서 [req.signedCookies](#)로 이동합니다. 서명된 쿠키는 값 앞에 [s:](#) 접두사가 붙은 쿠키입니다. 서명 유효성 검사에 실패한 서명된 쿠키는 변조된 값 대신 [false](#) 값을 갖습니다.

이제 다음과 같이 라우트 핸들러 내에서 쿠키를 읽을 수 있습니다:

```
@Get()  
findAll(@Req() request: 요청) {  
    console.log(request.cookies); // 또는 "request.cookies['cookieKey']"  
    // 또는 console.log(request.signedCookies);  
}
```

정보 힌트 `@Req()` 데코레이터는 `@nestjs/common`에서 가져온 것이고, 요청은 익스프레스 패키지.

발신 응답에 쿠키를 첨부하려면 `응답#쿠키()` 메서드를 사용합니다:

```
@Get()  
findAll(@Res({ 패스스루: true }) response: Response) { response.cookie('key',  
    'value')  
}
```

경고 경고 응답 처리 로직을 프레임워크에 맡기려면 위와 같이 `패스스루` 옵션을 `true`로 설정해야 합니다.

자세한 내용은 [여기를 참조하세요](#).

정보 힌트 `@Res()` 데코레이터는 `@nestjs/common`에서 가져오고, 응답은

익스프레스 패키지.

Fastify와 함께 사용

먼저 필요한 패키지를 설치합니다:

```
$ npm i @fastify/cookie
```

설치가 완료되면 `@fastify/cookie` 플러그인을 등록합니다:

```
'@fastify/cookie'에서 fastifyCookie를 가져옵니다;  
  
// 초기화 파일 어딘가에  
const app = await NestFactory.create<NestFastifyApplication>( AppModule,  
    새로운 FastifyAdapter(), );  
await app.register(fastifyCookie, { secret: 'my-secret', // 쿠키 서명용 });
```

이제 다음과 같이 라우트 핸들러 내에서 쿠키를 읽을 수 있습니다:

```
@Get()  
findAll(@Req() request: FastifyRequest) {  
    console.log(request.cookies); // 또는  
    "request.cookies['cookieKey']"  
}
```

정보 힌트 `@Req()` 데코레이터는 `@nestjs/common`에서 가져온 반면, `FastifyRequest`는 를 `Fastify` 패키지에 추가합니다.

발신 응답에 쿠키를 첨부하려면 `FastifyReply#setCookie()` 메서드를 사용합니다:

```
@Get()
findAll(@Res({ 패스스루: true }) 응답: FastifyReply) { response.setCookie('key',
  'value')
}
```

`FastifyReply#setCookie()` 메서드에 대해 자세히 알아보려면 이 [페이지를](#) 확인하세요.

경고 경고 응답 처리 로직을 프레임워크에 맡기려면 위와 같이 `패스스루` 옵션을 `true`로 설정해야 합니다.

자세한 내용은 [여기를](#) 참조하세요.

정보 힌트 `@Res()` 데코레이터는 `@nestjs/common`에서, `FastifyReply`는 `fastify` 패키지에서 가져옵니다.

사용자 지정 데코레이터 만들기(크로스 플랫폼)

들어오는 쿠키에 액세스하는 편리하고 선언적인 방법을 제공하기 위해 [사용자 지정 데코레이터를](#) 만들 수 있습니다.

```
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const Cookies = createParamDecorator(
  (데이터: 문자열, ctx: 실행 컨텍스트) => {
    const request = ctx.switchToHttp().getRequest();
    반환 데이터 ? 요청.쿠키?.[데이터] : 요청.쿠키;
  },
);
```

`쿠키()` 데코레이터는 모든 쿠키 또는 `req.cookies` 객체에서 명명된 쿠키를 추출하여 데코레이션된 매개변수를 해당 값으로 채웁니다.

이제 다음과 같이 경로 핸들러 시그니처에 데코레이터를 사용할 수 있습니다:

```
@Get()
findAll(@Cookies('name') name: string) {}
```

이벤트

이벤트 이미터 패키지(@nestjs/event-emitter)는 간단한 옵저버 구현을 제공하여 애플리케이션에서 발생하는 다양한 이벤트를 구독하고 수신할 수 있도록 합니다. 이벤트는 하나의 이벤트에 서로 의존하지 않는 여러 리스너를 가질 수 있으므로 애플리케이션의 다양한 측면을 분리하는 좋은 방법입니다.

EventEmitterModule은 내부적으로 eventemitter2 패키지를 사용합니다

다. 시작하기

먼저 필요한 패키지를 설치합니다:

```
npm i --save @nestjs/event-emitter
```

설치가 완료되면 이벤트이미터모듈을 루트 앱모듈로 가져와서

forRoot() 정적 메서드를 호출합니다:

@@파일명(앱.모듈)

'@nestjs/common'에서 { Module }을 가져옵니다;

'@nestjs/event-emitter'에서 { EventEmitterModule }을 임포트합니다;

```
모듈({ import: [
    EventEmitterModule.forRoot()
  ],
})
```

내보내기 클래스 AppModule {}

.forRoot() 호출은 이벤트 이미터를 초기화하고 앱 내에 존재하는 모든 선언적 이벤트 리스너를 등록합니다. 등록은 onApplicationBootstrap 수명 주기 후크가 발생할 때 발생하며, 모든 모듈이 모든 예약된 작업을 로드하고 선언했는지 확인합니다.

기본 이벤트이미터 인스턴스를 구성하려면 구성 객체를 .forRoot() 메서드에 전달합니다.

메서드에 대해 다음과 같이 설명합니다:

```
EventEmitterModule.forRoot({  
    // 와일드카드를 사용하려면 이것을 'true'로 설정합니다  
    . wildcard: false,  
    // 네임스페이스를 구분하는 데 사용되는 구분 기호 구분 기호:  
    ' ',  
    // 새로운 리스너 이벤트를 발생시키려면 이것을 `true`로 설정합니다,  
    // 제거 리스너 이벤트를 발생시키려면 이 값을 `true`로 설정합니다,  
    // 이벤트에 할당할 수 있는 최대 리스너 수입니다.
```

```
최대 청취자: 10,  
// 최대 리스너 수보다 많은 리스너가 할당된 경우 메모리 누수 메시지에 이벤트 이름 표시  
verboseMemoryLeak: false,  
// 오류 이벤트가 발생하고 리스너가 없는 경우 불포함 예외 발생을 비활성화합니다.  
무시 오류: 거짓,  
});
```

이벤트 발송

이벤트를 디스패치(즉, 발동)하려면 먼저 표준 생성자 주입을 사용하여 `EventEmitter2`를 주입합니다:

```
constructor(private eventEmitter: EventEmitter2) {}
```

정보 힌트 `@nestjs/event-emitter` 패키지에서 `EventEmitter2`를 가져옵니다.

그런 다음 다음과 같이 수업에서 사용하세요:

```
this.eventEmitter.emit(  
  'order.created',  
  새로운 주문 생성 이벤트({  
    orderId: 1,  
    페이로드: {},  
  }),  
);
```

이벤트 듣기

이벤트 리스너를 선언하려면 다음과 같이 실행할 코드가 포함된 메서드 정의 앞에 `@OnEvent()` 데코레이터를 사용하여 메서드를 장식합니다:

```
@OnEvent('order.created')  
handleOrderCreatedEvent(payload: OrderCreatedEvent) {  
  // "주문 생성 이벤트" 이벤트 처리 및 처리  
}
```

경고 경고 이벤트 구독자는 요청 범위를 지정할 수 없습니다.

첫 번째 인수는 단순 이벤트 이미터의 경우 문자열 또는 심볼, 와일드카드 이미터의 경우 문자열 | 심볼 |

Array<string | 심볼>일 수 있습니다. 두 번째 인수(선택 사항)는 리스너 옵션 객체입니다([자세히 보기](#)).

```
@OnEvent('order.created', { async: true })
handleOrderCreatedEvent(payload: OrderCreatedEvent) {
    // "주문 생성 이벤트" 이벤트 처리 및 처리
}
```

네임스페이스/와일드카드를 사용하려면 [이벤트이미터모듈#forRoot\(\)](#) 메서드에 [와일드카드](#) 옵션을 전달하세요. 네임스페이스/와일드카드가 활성화되면 이벤트는 구분 기호로 구분된 문자열(`foo.bar`)이거나 배열(`['foo', 'bar']`)이 될 수 있습니다. 구분 기호는 구성 속성(구분 [기호](#))으로도 구성할 수 있습니다. 네임스페이스 기능을 활성화하면 와일드카드를 사용하여 이벤트를 구독할 수 있습니다:

```
@OnEvent('order.*')
handleOrderEvents(payload: OrderCreatedEvent | OrderRemovedEvent |
OrderUpdatedEvent) {
    // 이벤트 처리 및 처리
}
```

이러한 와일드카드는 하나의 블록에만 적용됩니다. 예를 들어 `order.*` 인수는 `order.created` 및 `order.shipped` 이벤트와 일치하지만 `order.delayed.out_of_stock` 이벤트와는 일치하지 않습니다. 이러한 이벤트를 수신하려면 [EventEmitter2 문서](#)에 설명된 [다중 레벨 와일드카드 패턴](#)(예: `**`)을 사용하세요.

예를 들어 이 패턴을 사용하면 모든 이벤트를 수신하는 이벤트 리스너를 만들 수 있습니다.

```
@OnEvent('**')
handleEverything(payload: any) {
    // 이벤트 처리 및 처리
}
```

정보 힌트 [이벤트이미터2](#) 클래스는 다음과 같이 이벤트와 상호작용하는 데 유용한 몇 가지 메서드를 제공합니다.

`waitFor` 및 `onAny`. 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

압축

압축은 응답 본문의 크기를 크게 줄여 웹 앱의 속도를 높일 수 있습니다.

운영 중인 트래픽이 많은 웹사이트의 경우 일반적으로 역방향 프록시(예: Nginx)를 통해 애플리케이션 서버에서 압축을 오프로드하는 것이 좋습니다. 이 경우 압축 미들웨어를 사용해서는 안 됩니다.

Express와 함께 사용(기본값)

압축 미들웨어 패키지를 사용하여 gzip 압축을 활성화합니다. 먼저 필요한 패키

```
$ npm i --save 압축
```

지를 설치합니다:

설치가 완료되면 압축 미들웨어를 글로벌 미들웨어로 적용합니다.

```
'압축'에서 *를 압축으로 가져옵니다;  
// 초기화 파일 어딘가에 앱.사용(압축());
```

Fastify와 함께 사용

FastifyAdapter를 사용하는 경우 [fastify-compress](#)를 사용하는 것이 좋습니다:

```
$ npm i --save @fastify/compress
```

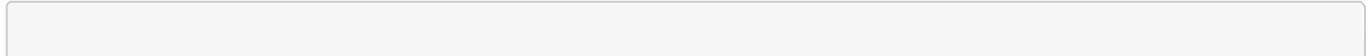
설치가 완료되면 [@fastify/compress](#) 미들웨어를 글로벌 미들웨어로 적용합니다.

```
'@fastify/compress'에서 압축을 가져옵니다;  
// 초기화 파일 어딘가에 await  
app.register(compression);
```

브라우저가 인코딩을 지원한다고 표시하면 기본적으로 [@fastify/compress](#)는 (노드 >= 11.7.0에서) Brotli 압축

을 사용합니다. Brotli는 압축률 측면에서 매우 효율적이지만 속도도 상당히 느립니다. 따라서 응답을 압축할 때 deflate와 gzip만 사용하도록 fastify-compress에 지시하면 응답 크기가 커지지만 훨씬 더 빠르게 전달됩니다.

인코딩을 지정하려면 `app.register`에 두 번째 인수를 제공합니다:



```
await app.register(compression, { encodings: ['gzip', 'deflate'] });
```

위의 내용은 클라이언트가 두 인코딩을 모두 지원하는 경우 gzip과 디플리플레이트 인코딩만 사용하도록 fastify-compress에 지시하고 gzip을 선호합니다.

파일 업로드

파일 업로드를 처리하기 위해 Nest는 Express용 **멀티** 미들웨어 패키지를 기반으로 하는 내장 모듈을 제공합니다. **멀티**는 주로 HTTP **POST** 요청을 통해 파일을 업로드하는 데 사용되는 **멀티파트/폼 데이터** 형식으로 게시된 데이터를 처리합니다. 이 모듈은 완전히 구성할 수 있으며 애플리케이션 요구 사항에 맞게 동작을 조정할 수 있습니다.

경고 멀티터는 지원되는 다중 파트 형식(**다중 파트/양식 데이터**)이 아닌 데이터를 처리할 수 없습니다. 또한 이 패키지는 **FastifyAdapter**와 호환되지 않습니다.

더 나은 타이핑 안전을 위해 다중 타이핑 패키지를 설치해 보겠습니다:

```
$ npm i -D @types/multer
```

이 패키지가 설치되었으므로 이제 **Express.Multer.File** 유형을 사용할 수 있습니다(이 유형을 다음과 같이 가져올 수 있습니다: `import {{ '{' }}, Express {{ '}' }} from 'express'`).

기본 예제

단일 파일을 업로드하려면 **FileInterceptor()** 인터셉터를 라우트 핸들러에 연결하고 다음을 추출하면 됩니다. 데코레이터를 사용하여 요청에서 파일을 업로드합니다.

```
@@파일명()
@Post('upload')
사용 인터셉터(FileInterceptor('file')) 업로드 파일(@UploadedFile() 파일: Express.Multer.File) {
    콘솔 로그(파일);
}

스위치 @포스트('업로드')

사용 인터셉터(FileInterceptor('file')) 바인드(업로드된 파일())
uploadFile(file) {
    console.log(file);
}
```

정보 힌트 **FileInterceptor()** 데코레이터는 [@nestjs/platform-express](#)에서 내보냅니다.

패키지로 내보냅니다. **업로드된 파일()** 데코레이터는 [@nestjs/common](#)에서 내보냅니다.

`FileInterceptor()` 데코레이터는 두 개의 인수를 받습니다:

- ◆ `fieldName`: 파일을 담고 있는 HTML 양식의 필드 이름을 제공하는 문자열 옵션: [다중 옵션 유형의 선택](#)
- ◆ 적 객체입니다. 이 객체는 다중 생성자에서 사용하는 것과 동일한 객체입니다(자세한 내용은 [여기](#)를 참조하세요).

경고 `FileInterceptor()`는 Google FireBase 등의 타사 클라우드 제공업체와 호환되지 않을 수 있습니다.

파일 유효성 검사

종종 파일 크기나 파일 MIME 유형과 같은 수신 파일 메타데이터의 유효성을 검사하는 것이 유용할 수 있습니다. 이를 위해 자신만의 `파이프`를 만들고 `UploadedFile` 데코레이터로 주석이 달린 매개변수에 바인딩할 수 있습니다. 아래 예시는 기본 파일 크기 유효성 검사기 파이프를 구현하는 방법을 보여줍니다:

```
'@nestjs/common'에서 { PipeTransform, Injectable, ArgumentMetadata }를 가져옵니다;
```

```
@Injectable()
export class FileSizeValidationPipe implements PipeTransform {
    transform(value: any, metadata: ArgumentMetadata) {
        // "value"는 파일의 속성과 메타데이터를 포함하는 객체 const oneKb = 1000입니다;
        반환 값.크기 < 1Kb;
    }
}
```

Nest는 일반적인 사용 사례를 처리하고 새로운 사용 사례의 추가를 용이하게/표준화하기 위한 내장 파이프를 제공합니다. 이 파이프는 `ParseFilePipe`라고 불리며 다음과 같이 사용할 수 있습니다:

```
@Post('file')
uploadFileAndPassValidation(
    Body() 본문: SampleDto,
    @UploadedFile(
        새로운 ParseFilePipe({ 유효성
            검사기: [
                // ... 여기에 파일 유효성 검사기 인스턴스 집합
            ]
        })
    )
    파일에 저장합니다: Express.Multer.File,
) {
    반환 { body,
        파일: file.buffer.toString(),
    };
}
```

보시다시피, `ParseFilePipe`에서 실행할 파일 유효성 검사기 배열을 지정해야 합니다. 유효성 검사기의 인터페이스에 대해서는 나중에 설명하겠지만, 이 파이프에는 두 가지 추가 옵션도 있다는 점을 언급할 필요가 있습니다:

`errorHttpStatusCode` 유효성 검사기가 실패할 경우 `throw`할 HTTP 상태 코드입니다. 기본값은 `400`(잘못된 요청)

예외 팩토리

오류 메시지를 수신하고 오류를 반환하는 팩토리입니다.

이제 `FileValidator` 인터페이스로 돌아갑니다. 유효성 검사기를 이 파이프와 통합하려면 기본 제공 구현을 사용하거나 사용자 지정 `FileValidator`를 제공해야 합니다. 아래 예시를 참조하세요:

```
내보내기 추상 클래스 FileValidator<TValidationOptions = Record<string, any>>
{
  생성자(보호된 읽기 전용 유효성 검사 옵션: TValidationOptions) {}

  /**
   * 생성자에서 전달된 옵션에 따라 이 파일을 유효한 파일로 간주할지 여부를 나타냅니다.
   * 매개변수 요청 객체의 파일을 파일로 저장합니다.
  */

  추상 isValid(file?: any): boolean | Promise<boolean>;

  /**
   * 유효성 검사에 실패할 경우 오류 메시지를 작성합니다.
   * 매개변수 요청 객체의 파일을 파일로 저장합니다.
  */

  추상 빌드 에러 메시지(파일: any): 문자열;
}
```

정보 힌트 `FileValidator` 인터페이스는 `isValid` 함수를 통해 비동기 유효성 검사를 지원합니다

. 유형 보안을 활용하려면 Express(기본값)를 드라이버로 사용하는 경우 파일 매개 변수를

`Express.Multer.File`로 입력할 수도 있습니다.

`FileValidator`는 파일 객체에 액세스하고 클라이언트가 제공한 옵션에 따라 유효성을 검사하는 일반 클래스입니다. Nest에는 프로젝트에서 사용할 수 있는 두 가지 `FileValidator` 구현이 내장되어 있습니다:

- `MaxFileSizeValidator` - 주어진 파일의 크기가 제공된 값보다 작은지 확인합니다(바이트)

`FileTypeValidator` - 지정된 파일의 파일 유형이 지정된 값과 일치하는지 확인합니다. 경고 경고 파일 유형을 확인하기 위해 `FileTypeValidator` 클래스는 멀티가 감지한 유형을 사용합니다. 기본적으로 멀티는 사용자 디바이스의 파일 확장자로부터 파일 유형을 도출합니다. 하지만 실제 파일 내용은 확인하지 않습니다. 파일의 이름을 임의의 확장자로 변경할 수 있으므로 앱에 더 안전한 솔루션이 필요 한 경우 파일의 `magic number` 확인과 같은 사용자 정의 구현을 사용하는 것이 좋습니다.

앞서 언급한 `FileParsePipe`와 함께 어떻게 사용할 수 있는지 이해하기 위해 지난번 제시된 예제의 변경된 스

니펫을 사용하겠습니다:

```
업로드된 파일(  
    새로운 ParseFilePipe({ 유효성 검  
        사기: [  
            새로운 MaxFileSizeValidator({ maxSize: 1000 }),  
            새로운 FileTypeValidator({ fileType: 'image/jpeg' })],
```

```
    ],
  }),
)

파일에 저장합니다: Express.Multer.File,
```

정보 힌트 유효성 검사기 수가 크게 증가하거나 옵션이 파일을 복잡하게 만드는 경우, 이 배열을 별도의 파일에 정의한 다음 `fileValidators` 같은 명명된 상수로 가져올 수 있습니다.

마지막으로, 유효성 검사기를 작성하고 구성할 수 있는 특별한 `ParseFilePipeBuilder` 클래스를 사용할 수 있습니다. 아래 그림과 같이 이 클래스를 사용하면 각 유효성 검사기의 수동 인스턴스화를 피하고 옵션을 직접 전달할 수 있습니다:

```
업로드된 파일(
  새로운 ParseFilePipeBuilder()
    .addFileTypeValidator({
      fileType: 'jpeg',
    })
    .addMaxSizeValidator({
      maxSize: 1000
    })
    .build({
      errorHttpStatusCode: HttpStatus.UNPROCESSABLE_ENTITY
    }),
)

파일에 저장합니다: Express.Multer.File,
```

파일 배열

파일 배열(단일 필드명으로 식별됨)을 업로드하려면 `FilesInterceptor()` 데코레이터를 사용합니다(데코레이터 이름에 복수의 파일이 있는 것을 참고하세요). 이 데코레이터는 세 개의 인수를 받습니다:

- `fieldName`: 위에 설명된 대로
- `최대` 개수: 허용할 최대 파일 수를 정의하는 선택적 숫자입니다.
- `옵션`: 위에서 설명한 대로 선택적 `다중` 옵션 객체입니다.

`FilesInterceptor()`를 사용하는 경우, `@UploadedFiles()`를 사용하여 요청에서 파일을 추출합니다. 데코레이터.

```
@@파일명()
@Post('upload')
사용 인터셉터(FilesInterceptor('files')) 업로드 파일(@UploadedFiles() 파일:
Array<Express.Multer.File>) {
    콘솔 로그(파일);
}

스위치 @포스트('업로드')

사용 인터셉터(FilesInterceptor('files')) 바인드(업로드된 파일
())
```

```
uploadFile(files) {
  console.log(files);
}
```

정보 힌트 `FilesInterceptor()` 데코레이터는 `@nestjs/platform-express` 패키지에서 내보냅니다. 업로드된 파일() 데코레이터는 `@nestjs/common`에서 내보냅니다.

여러 파일

여러 파일(모두 필드 이름 키가 다른)을 업로드하려면 `FileFieldsInterceptor()` 데코레이터. 이 데코레이터는 두 개의 인자를 받습니다:

- `uploadedFields`: 객체 배열로, 각 객체는 위에서 설명한 대로 필드 이름을 지정하는 문자열 값과 함께 필수 이름 속성을 지정하고 위에서 설명한 대로 선택적 `maxCount` 속성을 지정합니다.
- `옵션`: 위에서 설명한 대로 선택적 `다중 옵션` 객체입니다.

`FileFieldsInterceptor()`를 사용하는 경우, `@UploadedFiles()`를 사용하여 요청에서 파일을 추출합니다. 데코레이터.

```
@@파일명()
@Post('upload')
@UseInterceptors(FileFieldsInterceptor([
  { name: 'avatar', maxCount: 1 },
  { name: 'background', maxCount: 1 },
]))
업로드 파일(@UploadedFiles() 파일: { avatar?: Express.Multer.File[], background?: Express.Multer.File[] }) {
  콘솔 로그(파일);
}

스위치 @Post('업로드')

@Bind(업로드된파일())
@UseInterceptors(FileFieldsInterceptor([
  { name: 'avatar', maxCount: 1 },
  { name: 'background', maxCount: 1 },
]))
uploadFile(files) {
  console.log(files);
}
```

모든 파일

임의의 필드 이름 키가 있는 모든 필드를 업로드하려면 `AnyFilesInterceptor()` 데코레이터를 사용합니다. 이 데코레이터는 위에서 설명한 대로 선택적 옵션 객체를 받을 수 있습니다.

`AnyFilesInterceptor()`를 사용하는 경우, `@UploadedFiles()`를 사용하여 요청에서 파일을 추출합니다. 데코레이터.

```

@@파일명()
@Post('upload')
사용 인터셉터(AnyFilesInterceptor()) 업로드 파일(@UploadedFiles() 파일:
Array<Express.Multer.File>) {
    콘솔 로그(파일);
}

스위치 @Post('업로드')

@Bind(업로드된파일())
UseInterceptors(AnyFilesInterceptor())
uploadFile(files) {
    콘솔 로그(파일);
}

```

파일 없음

멀티파트/양식 데이터는 허용하지만 파일 업로드를 허용하지 않으려면 `NoFilesInterceptor`를 사용하세요. 이렇게 하면 요청 본문에 멀티파트 데이터를 속성으로 설정합니다. 요청과 함께 전송된 모든 파일은 `BadRequestException`을 던집니다.

```

Post('upload')
@UseInterceptors(NoFilesInterceptor())
handleMultiPartData(@Body() body) {
    콘솔 로그(본문)
}

```

기본 옵션

위에서 설명한 대로 파일 인터셉터에서 멀티 옵션을 지정할 수 있습니다. 기본 옵션을 설정하려면, `멀티모듈`을 임포트할 때 정적 `register()` 메서드를 호출하여 지원되는 옵션을 전달하면 됩니다. 여기에 나열된 모든 옵션을 사용할 수 있습니다.

```

Multimodule.register({
  dest: './upload',
});

```

정보 힌트 `멀티모듈` 클래스는 `@nestjs/플랫폼-익스프레스` 패키지에서 내보냅니다.

비동기 구성

멀티모듈 옵션을 정직이 아닌 비동기적으로 설정해야 하는 경우 `registerAsync()` 메서드를 사용하세요. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성의 처리하는 몇 가지 기술을 제공합니다.

한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
Multimodule.registerAsync({
  useFactory: () => ({
    목적지: './업로드',
  }),
});
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 [비동기화될](#) 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.
주입합니다.

```
MulterModule.registerAsync({
  import: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    dest: configService.get<string>('MULTER_DEST'),
  }),
  주입합니다: [구성 서비스],
});
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 [멀티모듈을](#) 구성할 수도 있습니다:

```
MulterModule.registerAsync({
  useClass: MulterConfigService,
});
```

위의 구조는 [MulterModule](#) 내부에 [MulterConfigService](#)를 인스턴스화하여 이를 사용하여 필요한 옵션 객체를 생성합니다. 이 예제에서 [MulterConfigService](#)는 아래와 같이 [MulterOptionsFactory](#) 인터페이스를 구현해야 한다는 점에 유의하세요. 제공된 클래스의 인스턴스화된 객체에서 [MulterModule](#)의 [createMulterOptions\(\)](#) 메서드를 호출합니다.

```
@Injectable()
MulterConfigService 클래스는 MulterOptionsFactory를 구현합니다 {
  createMulterOptions(): MulterModuleOptions {
    반환 {
      목적지: './업로드',
    };
  }
}
```

내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려는 경우
다중 모듈을 사용하려면 [useExisting](#) 구문을 사용합니다.

```
MulterModule.registerAsync({  
  import: [ConfigModule],
```

```
사용Existing: ConfigService,  
});
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

스트리밍 파일

정보 참고 이 장에서는 HTTP 애플리케이션에서 파일을 스트리밍하는 방법을 설명합니다. 예제 아래에 제시된 내용은 GraphQL 또는 마이크로서비스 애플리케이션에는 적용되지 않습니다.

REST API에서 클라이언트로 파일을 다시 보내고 싶을 때가 있을 수 있습니다. Nest에서 이 작업을 수행하려면 일반적으로 다음을 수행합니다:

```
컨트롤러('파일')
내보내기 클래스 FileController {
  @Get()
  getFile(@Res() res: Response) {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    file.pipe(res);
  }
}
```

하지만 이렇게 하면 컨트롤러 이후 인터셉터 로직에 액세스할 수 없게 됩니다. 이를 처리하기 위해 StreamableFile 인스턴스를 반환하면 프레임워크가 내부에서 응답을 파이핑하는 작업을 처리합니다.

스트리밍 가능한 파일 클래스

StreamableFile은 반환할 스트림을 보유하는 클래스입니다. 새로운

StreamableFile 생성자에 버퍼 또는 스트림을 전달할 수 있습니다.

정보 힌트 StreamableFile 클래스는 [@nestjs/common](#)에서 가져올 수 있습니다.

크로스 플랫폼 지원

Fastify는 기본적으로 stream.pipe(res)를 호출할 필요 없이 파일 전송을 지원할 수 있으므로

StreamableFile 클래스를 전혀 사용할 필요가 없습니다. 그러나 Nest는 두 플랫폼 유형 모두에서

StreamableFile 사용을 지원하므로 Express와 Fastify를 전환하는 경우 두 엔진 간의 호환성에 대해 걱정할 필요가 없습니다.

예

아래에서 패키지.json을 JSON 대신 파일로 반환하는 간단한 예제를 찾을 수 있지만 이 아이디어는 이미지, 문서 및 기타 모든 파일 유형으로 자연스럽게 확장됩니다.

```
'@nestjs/common'에서 { Controller, Get, StreamableFile }을 임포트하고,  
'fs'에서 { createReadStream }을 임포트합니다;  
'경로'에서 { join }을 가져옵니다;
```

컨트롤러('파일')

```
내보내기 클래스 FileController {
```

```
    @Get()
```

```
getFile(): StreamableFile {
  const file = createReadStream(join(process.cwd(), 'package.json'));
  return new StreamableFile(file);
}
```

기본 콘텐츠 유형은 애플리케이션/옥텟 스트림이며, 응답을 사용자 정의해야 하는 경우 다음과 같이 `res.set` 메서드 또는 `@Header()` 데코레이터를 사용할 수 있습니다:

```
'@nestjs/common'에서 { Controller, Get, StreamableFile, Res }를 임포트하고,
'fs'에서 { createReadStream }을 임포트합니다;
'경로'에서 { join }을 가져옵니다;
'express'에서 { Response } 유형을 가져옵니다;
```

컨트롤러('파일')

```
내보내기 클래스 FileController {
  @Get()
  getFile(@Res({ 패스스루: true }) res: Response): StreamableFile { const
    file = createReadStream(join(process.cwd(), 'package.json'));
    res.set({
      '콘텐츠 유형': '애플리케이션/json',
      '콘텐츠-처분': '첨부파일; 파일명="package.json"',
    });
    새로운 StreamableFile(file)을 반환합니다;
  }

  // 또는 짹수:
  @Get()
  헤더('콘텐츠 유형', '애플리케이션/json')
  @Header('Content-Disposition', 'attachment; filename="package.json"')
  getStaticFile(): StreamableFile {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    return new StreamableFile(file);
  }
}
```

HTTP 모듈

Axios는 널리 사용되는 풍부한 기능을 갖춘 HTTP 클라이언트 패키지입니다. Nest는 Axios를 래핑하여 내장된 [HttpModule](#)을 통해 노출합니다. HttpModule은 HTTP 요청을 수행하기 위한 Axios 기반 메서드를 노출하는 [HttpService](#) 클래스를 내보냅니다. 또한 라이브러리는 결과 HTTP 응답을 [Observable](#)로 변환합니다.

정보 힌트 [got](#) 또는 [undici](#)를 포함한 모든 범용 Node.js HTTP 클라이언트 라이브러리를 직접 사용할 수도 있습니다.

설치

사용을 시작하려면 먼저 필수 종속성을 설치합니다.

```
npm i --save @nestjs/axios axios
```

시작하기

설치 프로세스가 완료되면 [HttpService](#)를 사용하려면 먼저 [HttpModule](#)을 가져옵니다.

```
모듈 ({  
  수입: [HttpModule], 공급자:  
  [CatsService],  
})  
내보내기 클래스 CatsModule {}
```

다음으로 일반 생성자 주입을 사용하여 [HttpService](#)를 주입합니다.

정보 힌트 `HttpModule`과 `HttpService`는 `@nestjs/axios` 패키지에서 가져옵니다.

```
@@파일명()
@Injectable()
내보내기 클래스 CatsService {
    생성자(비공개 읽기 전용 httpService: HttpService) {}

    findAll(): Observable<AxiosResponse<Cat[]>> {
        이.httpService.get('http://localhost:3000/cats')을 반환합니다;
    }
}

@@스위치 @Injectable()
@Dependencies(HttpService)
내보내기 클래스 CatsService {
    constructor(httpService) {
        this.httpService = httpService;
    }
}
```

```
findAll() {
  이.httpService.get('http://localhost:3000/cats')을 반환합니다;
}
}
```

정보 힌트 AxiosResponse는 `axios` 패키지에서 내보낸 인터페이스입니다(`$ npm i axios`).

모든 `HttpService` 메서드는 관찰 가능한 객체로 래핑된 `AxiosResponse`을 반환합니다.

구성

Axios는 다양한 옵션으로 구성하여 `HttpService`의 동작을 사용자 정의할 수 있습니다. Read 자세한 내용은 [여기를](#) 참조하세요. 기본 Axios 인스턴스를 구성하려면 인스턴스를 가져올 때 선택적 옵션 객체를 `HttpModule`의 `register()` 메서드에 전달합니다. 이 옵션 객체는 기본 Axios 생성자에게 직접 전달됩니다.

```
모듈({ import: [
  HttpModule.register({
    timeout: 5000,
    최대 리디렉션: 5,
  }),
],
공급자: [CatsService],
})
내보내기 클래스 CatsModule {}
```

비동기 구성

모듈 옵션을 정적이 아닌 비동기적으로 전달해야 하는 경우, `registerAsync()`

메서드를 사용합니다. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성을 처리하는 몇 가지 기술을 제공합니다. 한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
HttpModule.registerAsync({  
    useFactory: () => ({  
        시간 초과: 5000,  
        최대 리디렉션: 5,  
    }),  
});
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 [비동기화될](#) 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.

[주입합니다.](#)

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    timeout: configService.get('HTTP_TIMEOUT'),
    maxRedirects: configService.get('HTTP_MAX_REDIRECTS'),
  }),
  주입합니다: [구성 서비스],
});
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 HttpModule을 구성할 수도 있습니다.

```
HttpModule.registerAsync({
  useClass: HttpConfigService,
});
```

위의 구조는 `HttpModule` 내부에 `HttpConfigService`를 인스턴스화하여 이를 사용하여 옵션 객체를 생성합니다. 이 예제에서 `HttpConfigService`는 아래와 같이 `HttpModuleOptionsFactory` 인터페이스를 구현해야 한다는 점에 유의하세요. `HttpModule`은 제공된 클래스의 인스턴스화된 객체에서 `createHttpOptions()` 메서드를 호출합니다.

```
@Injectable()
HttpConfigService 클래스는 HttpModuleOptionsFactory를 구현합니다 {
  createHttpOptions(): HttpModuleOptions {
    반환 { 시간 초과:
      5000,
      최대 리디렉션: 5,
    };
  }
}
```

`HttpModule` 내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려면 `useExisting` 구문을 사용하세요.

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useExisting: HttpConfigService,
});
```

Axios 직접 사용

`HttpModule.register`의 옵션이 충분하지 않다고 생각되거나 `@nestjs/axios`에서 생성한 기본 Axios 인스턴스에만 액세스하려는 경우, 다음과 같이 `HttpService#axiosRef`를 통해 액세스할 수 있습니다:

```

@Injectable()
내보내기 클래스 CatsService {
    생성자(비공개 읽기 전용 httpService: HttpService) {}

    findAll(): Promise<AxiosResponse<Cat[]>> {
        이.httpService.axiosRef.get('http://localhost:3000/cats')을 반환합니다;
        //^ AxiosInstance 인터페이스
    }
}

```

전체 예제

HttpService 메서드의 반환값이 Observable이기 때문에, rxjs를 사용할 수 있습니다.

첫 번째 값 또는 마지막 값에서 요청의 데이터를 프로미스 형태로 검색합니다.

```

'rxjs'에서 { catchError, firstValueFrom } import;

@Injectable()
내보내기 클래스 CatsService {
    private readonly logger = new Logger(CatsService.name);
    constructor(private readonly httpService: HttpService) {}

    비동기 findAll(): Promise<Cat[]> {
        const { data } = await firstValueFrom(
            this.httpService.get<Cat[]>('http://localhost:3000/cats').pipe(
                catchError((error: AxiosError) => {
                    this.logger.error(error.response.data);
                    throw '오류가 발생했습니다!';
                }),
            ),
        );
        데이터를 반환합니다;
    }
}

```

정보 힌트 첫 번째 값과 마지막 값의 차이점은 RxJS의 첫 번째 값과 마지막 값에 대한 설명서를 참조하세요.

세션

HTTP 세션은 여러 요청에 걸쳐 사용자에 대한 정보를 저장할 수 있는 방법을 제공하며, 특히 [MVC](#) 애플리케이션에 유용합니다.

Express와 함께 사용(기본값)

먼저 [필요한 패키지](#)(TypeScript 사용자의 경우 해당 유형)를 설치합니다:

```
$ npm i express-session
$ npm i -D @types/express-session
```

설치가 완료되면 [익스프레스 세션](#) 미들웨어를 전역 미들웨어로 적용합니다(예: `main.ts` 파일에).

```
'express-session'에서 *를 세션으로 가져옵니다;
// 초기화 파일 어딘가에 앱.사용(
  세션({
    secret: 'my-secret',
    resave: false,
    saveUninitialized: false,
  }),
);
```

경고 기본 서버 측 세션 저장소는 의도적으로 프로덕션 환경을 위해 설계되지 않았습니다. 대부분의 조건에서 메모리가 누수되고 단일 프로세스를 초과하여 확장되지 않으며 디버깅 및 개발용입니다. [공식 리포지토리](#)에서 자세히 알아보세요.

비밀은 세션 ID 쿠키에 서명하는 데 사용됩니다. 이는 단일 비밀에 대한 문자열이거나 여러 비밀의 배열일 수 있습니다. 시크릿 배열이 제공되면 첫 번째 요소만 세션 ID 쿠키에 서명하는 데 사용되며, 요청에서 서명을 확인할 때 모든 요소가 고려됩니다. 시크릿 자체는 사람이 쉽게 파싱할 수 없어야 하며 임의의 문자 집합을 사용하는 것이 가장 좋습니다.

다시 저장 옵션을 활성화하면 요청 중에 세션을 수정하지 않은 경우에도 세션이 세션 저장소에 강제로 저장됩니다. 기본값은 `true`이지만 기본값은 향후 변경될 예정이므로 기본값을 사용하는 것은 더 이상 권장되지 않습니다.

마찬가지로 저장 초기화 옵션을 활성화하면 "초기화되지 않은" 세션이 스토어에 강제로 저장됩니다. 세션은 새 세션이지만 수정되지 않은 경우 초기화되지 않습니다. false를 선택하면 로그인 세션을 구현하거나 서버 스토리지 사용량을 줄이거나 쿠키를 설정하기 전에 허가가 필요한 법률을 준수하는 데 유용합니다. false를 선택하면 클라이언트가 세션([소스](#)) 없이 여러 개의 병렬 요청을 하는 경쟁 조건에도 도움이 됩니다.

세션 미들웨어에 몇 가지 다른 옵션을 전달할 수 있으며, 이에 대한 자세한 내용은 [API 설명서를](#) 참조하세요.

정보 힌트 보안: 참이 권장 옵션입니다. 단, 보안 쿠키를 사용하려면 HTTPS가 활성화된 웹사이트가 필요합니다. 보안이 설정되어 있고 HTTP를 통해 사이트에 액세스하면 쿠키가 설정되지 않습니다. 프록시 뒤에 node.js가 있고 보안: true를 사용하는 경우 익스프레스에서 "신뢰 프록시"를 설정해야 합니다.

이제 다음과 같이 라우트 핸들러 내에서 세션 값을 설정하고 읽을 수 있습니다:

```
@Get()  
findAll(@Req() request: 요청) {  
    요청.세션.방문 = 요청.세션.방문 ? 요청.세션.방문  
    + 1 : 1;  
}
```

정보 힌트 @Req() 데코레이터는 @nestjs/common에서 가져온 것이고, 요청은 익스프레스 패키지.

또는 다음과 같이 @Session() 데코레이터를 사용하여 요청에서 세션 객체를 추출할 수 있습니다:

```
@Get()  
findAll(@Session() 세션: Record<string, any>) { session.visits =  
    세션.방문 ? 세션.방문 + 1 : 1;  
}
```

정보 힌트 @Session() 데코레이터는 @nestjs/common 패키지에서 가져옵니다.

Fastify와 함께 사용

먼저 필요한 패키지를 설치합니다:

```
$ npm i @fastify/secure-session
```

설치가 완료되면 fastify-secure-session 플러그인을 등록합니다:

'@fastify/secure-session'에서 `secureSession`을 가져옵니다;

```
// 초기화 파일 어딘가에
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  새로운 FastifyAdapter(),
);
await app.register(secureSession, {
  비밀: 'averylogphrasebiggerthanthirtytwochars', 소금:
  'mq9hDxBVDbspDR6n',
});
```

정보 힌트 키를 미리 생성하거나([지침 참조](#)) 키 회전을 사용할 수도 있습니다.

공식 리포지토리에서 사용 가능한 옵션에 대해 자세히 알아보세요.

이제 다음과 같이 라우트 핸들러 내에서 세션 값을 설정하고 읽을 수 있습니다:

```
@Get()  
findAll(@Req() 요청: FastifyRequest) { const  
  visits = request.session.get('visits');  
  요청.세션.set('방문', 방문 ? 방문 + 1 : 1);  
}
```

또는 다음과 같이 `@Session()` 데코레이터를 사용하여 요청에서 세션 객체를 추출할 수 있습니다:

```
@Get()  
findAll(@Session() session: secureSession.Session) {  
  const visits = session.get('visits');  
  session.set('visits', visits ? visits + 1 : 1);  
}
```

정보 힌트 `@Session()` 데코레이터는 [@nestjs/common](#)에서, `secureSession.Session`은 [@fastify/sure-session](#) 패키지에서 가져옵니다(가져오기 문: '`@fastify/sure-session`'에서 *를 `secureSession`으로 가져오기).

모델-보기-컨트롤러

Nest는 기본적으로 [Express](#) 라이브러리를 내부적으로 사용합니다. 따라서 Express에서 MVC(모델-뷰-컨트롤러) 패턴을 사용하는 모든 기술은 Nest에도 적용됩니다.

먼저 [CLI](#) 도구를 사용하여 간단한 Nest 애플리케이션을 스캐폴딩해 보겠습니다:

```
$ npm i -g @nestjs/cli  
둥지 새 프로젝트
```

MVC 앱을 만들려면 HTML 뷰를 렌더링할 [템플릿 엔진](#)도 필요합니다:

```
$ npm install --save hbs
```

여기서는 [hbs\(핸들바\)](#) 엔진을 사용했지만 요구 사항에 맞는 엔진을 사용할 수 있습니다. 설치 프로세스가 완료되면 다음 코드를 사용하여 익스프레스 인스턴스를 구성해야 합니다:

@@파일명(메인)

'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/platform-express'에서 { NestExpressApplication } 임포트; '경로'
에서 { join } 임포트;
'./app.module'에서 { AppModule }을 가져옵니다;

비동기 함수 부트스트랩()

```
const app = await NestFactory.create<NestExpressApplication>(  
  AppModule,  
);  
  
app.useStaticAssets(join(__dirname, '.', 'public'));  
app.setBaseViewsDir(join(__dirname, '.', 'views'));  
app.setViewEngine('hbs');  
  
await app.listen(3000);  
}
```

부트스트랩(); @@스

위치

'@nestjs/core'에서 { NestFactory }를 가져오고,
'path'에서 { join }를 가져옵니다;
'./app.module'에서 { AppModule }을 가져옵니다;

비동기 함수 부트스트랩()

```
const app = await NestFactory.create(  
  AppModule,  
);  
  
app.useStaticAssets(join(__dirname, '.', 'public'));  
app.setBaseViewsDir(join(__dirname, '.', 'views'));
```

```
app.setViewEngine('hbs');

await app.listen(3000);
}

부트스트랩();
```

공용 디렉토리는 정적 애셋을 저장하는 데 사용되고, 뷰에는 템플릿이 포함되며, HTML 출력을 렌더링하는 데는 hbs 템플릿 엔진을 사용해야 한다고 [Express에](#) 설명했습니다.

템플릿 렌더링

이제 뷰 디렉터리와 그 안에 `index.hbs` 템플릿을 만들어 보겠습니다. 템플릿에서 메시지를 전달합니다:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>앱</title>
  </head>
  <body>
    {{ "{{ 메시지 }}" }}
  </body>
</html>
```

그런 다음 `app.controller` 파일을 열고 `root()` 메서드를 다음 코드로 바꿉니다:

```
@@파일명(앱.컨트롤러)

'@nestjs/common'에서 { Get, Controller, Render }를 가져옵니다;

컨트롤러()

내보내기 클래스 AppController {
  @Get()
  @Render('index')
  root() {
    반환 { 메시지: '안녕하세요!' };
  }
}
```

이 코드에서는 `@Render()` 데코레이터에서 사용할 템플릿을 지정하고 있으며, 라우트 핸들러 메서드의 반환값은 렌더링을 위해 템플릿으로 전달됩니다. 반환 값은 템플릿에서 생성한 `메시지` 자리 표시자와 일치하는 속성 `메`

시지가 있는 객체입니다.

애플리케이션이 실행되는 동안 브라우저를 열고 <http://localhost:3000> 으로 이동합니다. 안녕하세요! 메시지가 표시될 것입니다.

동적 템플릿 렌더링

애플리케이션 로직이 렌더링할 템플릿을 동적으로 결정해야 하는 경우, `@Res()`를 사용해야 합니다.

데코레이터를 사용하고 `@Render()` 데코레이터가 아닌 경로 핸들러에 뷰 이름을 지정합니다:

정보 힌트 Nest가 `@Res()` 데코레이터를 감지하면 라이브러리별 응답 객체를 삽입합니다. 이 객체를 사용하여 템플릿을 동적으로 렌더링할 수 있습니다. 여기에서 응답 객체 API에 대해 자세히 알아보세요.

@@파일명(앱.컨트롤러)

```
'@nestjs/common'에서 { Get, Controller, Res, Render }를 가져오고,  
'express'에서 { Response }를 가져옵니다;  
'./app.service'에서 { AppService }를 가져옵니다;
```

컨트롤러()

```
내보내기 클래스 AppController {  
    constructor(private appService: AppService) {}  
  
    @Get()  
    root(@Res() res: Response) {  
        return res.render(  
            이.앱서비스.getAppName(),  
            { 메시지: 'Hello world!' },  
        );  
    }  
}
```

예

작동 예제는 [여기에서](#) 확인할 수 있습

니다. Fastify

이 [장에서](#) 언급했듯이 호환 가능한 모든 HTTP 공급자를 Nest와 함께 사용할 수 있습니다. One 이러한 라이브러리가 바로 [Fastify입니다](#). Fastify를 사용하여 MVC 애플리케이션을 생성하려면 다음 패키지를 설치해야 합니다:

```
$ npm i --save @fastify/static @fastify/view 핸들바 저장
```

다음 단계는 플랫폼에 따라 약간의 차이가 있지만 Express에서 사용되는 프로세스와 거의 동일합니다. 설치 프로

세스가 완료되면 `main.ts` 파일을 열고 내용을 업데이트합니다:

@@파일명(메인)

'@nestjs/core'에서 { NestFactory }를 가져옵니다;

'@nestjs/platform-fastify'에서 { NestFastifyApplication, FastifyAdapter }를 가져옵니다;

'./app.module'에서 { AppModule }을 임포트하고, '경로'에서 { join }을 임포트합니다;

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    새로운 FastifyAdapter(),
  );
  app.useStaticAssets({
    root: join(__dirname, '.', 'public'),
    접두사: '/public/',
  });
  app.setViewEngine({ 엔
    진: {
      핸들바: require('핸들바'),
    },
    템플릿: 조인(__dirname, '.', 'views'),
  });
  await app.listen(3000);
}

부트스트랩(); @@스
```

위치

'@nestjs/core'에서 { NestFactory }를 가져옵니다;
 '@nestjs/platform-fastify'에서 { FastifyAdapter }를 임포트하고,
 './app.module'에서 { AppModule }을 임포트합니다;
 '경로'에서 { join }을 가져옵니다;

비동기 함수 부트스트랩() {

```
const app = await NestFactory.create(AppModule, new FastifyAdapter());
app.useStaticAssets({
  root: join(__dirname, '.', 'public'),
  접두사: '/public/',
});
app.setViewEngine({ 엔
  진: {
    핸들바: require('핸들바'),
  },
  템플릿: 조인(__dirname, '.', 'views'),
});
await app.listen(3000);
}

부트스트랩();
```

Fastify API는 약간 다르지만 메서드 호출의 최종 결과는 동일하게 유지됩니다. Fastify의 한 가지 차이점은 `@Render()` 데코레이터에 전달되는 템플릿 이름에 파일 확장자가 포함되어야 한다는 점입니다.

@@파일명(앱.컨트롤러)

'@nestjs/common'에서 { Get, Controller, Render }를 가져옵니다;

컨트롤러()

```
export class AppController
{ @Get()
  @Render('index.hbs')
```

```
root() {  
    반환 { 메시지: '안녕하세요!' };  
}  
}
```

애플리케이션이 실행되는 동안 브라우저를 열고 <http://localhost:3000> 으로 이동합니다. 안녕하세요! 메시지가 표시될 것입니다.

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

성능(Fastify)

기본적으로 Nest는 [Express](#) 프레임워크를 사용합니다. 앞서 언급했듯이 Nest는 [Fastify](#)와 같은 다른 라이브러리 와의 호환성을 제공합니다. Nest는 미들웨어와 핸들러를 적절한 라이브러리별 구현으로 프록시하는 것을 주요 기능으로 하는 프레임워크 어댑터를 구현함으로써 이러한 프레임워크 독립성을 달성합니다.

정보 힌트 프레임워크 어댑터를 구현하려면 대상 라이브러리가 Express에서와 유사한 요청/응답 파이프 라인 처리 기능을 제공해야 합니다.

Fastify는 Express와 비슷한 방식으로 디자인 문제를 해결하기 때문에 Nest를 위한 좋은 대체 프레임워크를 제공합니다. 그러나 Fastify는 Express보다 훨씬 빠르며 벤치마크 결과도 거의 두 배 더 우수합니다. 그렇다면 Nest 가 기본 HTTP 공급자로 Express를 사용하는 이유는 무엇일까요? 그 이유는 Express가 널리 사용되고 잘 알려져 있으며 Nest 사용자가 즉시 사용할 수 있는 호환 가능한 미들웨어 세트가 방대하기 때문입니다.

그러나 Nest는 프레임워크 독립성을 제공하므로 두 프레임워크 간에 쉽게 마이그레이션할 수 있습니다. 매우 빠른 성능을 중시하는 경우 Fastify가 더 나은 선택이 될 수 있습니다. Fastify를 사용하려면 이 챕터에 표시된 대로 기본 제공되는 [FastifyAdapter](#)를 선택하기만 하면 됩니다.

설치

먼저 필요한 패키지를 설치해야 합니다:

```
npm i --save @nestjs/platform-fastify
```

어댑터

Fastify 플랫폼이 설치되면 [FastifyAdapter](#)를 사용할 수 있습니다.

@@파일명 (메인)

'@nestjs/core'에서 { NestFactory }를 가져옵니다.

FastifyAdapter,
NestFastifyApplication,
}를 '@nestjs/platform-fastify'에서 가져옵니다;
'./app.module'에서 { AppModule }을 가져옵니다;

비동기 함수 부트스트랩() {

```
const app = await NestFactory.create<NestFastifyApplication>(  
  AppModule,  
  새로운 FastifyAdapter()  
)  
await app.listen(3000);  
}
```

부트스트랩();

기본적으로 Fastify는 `localhost 127.0.0.1` 인터페이스에서만 수신 대기합니다([자세히 보기](#)). 다른 호스트에서 연결을 수락하려면 `listen()` 호출에 '`0.0.0.0`'을 지정해야 합니다:

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    새로운 FastifyAdapter(),
  );
  await app.listen(3000, '0.0.0.0');
}
```

플랫폼별 패키지

`FastifyAdapter`을 사용할 때 Nest는 HTTP 공급자로 Fastify를 사용한다는 점에 유의하세요. 즉, Express에 의존하는 각 레시피가 더 이상 작동하지 않을 수 있습니다. 대신 Fastify와 동등한 패키지를 사용해야 합니다.

응답 리디렉션

Fastify는 리디렉션 응답을 Express와 약간 다르게 처리합니다. Fastify로 올바른 리디렉션을 수행하려면 다음과 같이 상태 코드와 URL을 모두 반환하세요:

```
@Get()
index(@Res() res) {
  res.status(302).redirect('/login');
}
```

단축 옵션

`FastifyAdapter` 생성자를 통해 Fastify 생성자에 옵션을 전달할 수 있습니다. 예를 들어

```
새로운 FastifyAdapter({ logger: true });
```

미들웨어

미들웨어 함수는 Fastify의 래퍼 대신 원시 `요청` 및 `res` 객체를 검색합니다. 이것이 [미디](#) 패키지가 작동하는 방식(내부에서 사용되는 방식)이며, 자세한 내용은 이 [페이지를](#) 참조하세요,

@@파일명(logger.middleware)

'@nestjs/common'에서 { Injectable, NestMiddleware }를 임포트하고,

'fastify'에서 { FastifyRequest, FastifyReply }를 임포트합니다;

@Injectable()

```
내보내기 클래스 LoggerMiddleware는 NestMiddleware를 구현합니다 {
  use(req: FastifyRequest['raw'], res: FastifyReply['raw'], next: () =>
void) {
  console.log('요청...'); next();
}
}

@@switch
'@nestjs/common'에서 { Injectable }을 임포트합니다;

@Injectable()
export class LoggerMiddleware {
  use(req, res, next) {
    console.log('요청...'); next();
  }
}
```

경로 구성

Fastify의 경로 구성 기능을 `@RouteConfig()` 데코레이터와 함께 사용할 수 있습니다.

```
@RouteConfig({ 출력: 'hello world' })
@Get()
index(@Req() req) {
  req.routeConfig.output을 반환합니다;
}
```

정보 힌트 `@RouteConfig()`는 `@nestjs/platform-fastify`에서 가져온 것입니다.

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

서버 전송 이벤트

서버 전송 이벤트(SSE)는 클라이언트가 HTTP 연결을 통해 서버로부터 자동 업데이트를 수신할 수 있도록 하는 서버 푸시 기술입니다. 각 알림은 한 쌍의 개행으로 끝나는 텍스트 블록으로 전송됩니다([여기에서](#) 자세히 알아보기).

사용법

라우트(컨트롤러 클래스 내에 등록된 라우트)에서 서버 전송 이벤트를 활성화하려면 메서드 핸들러에 `@Sse()` 데코레이터를 주석으로 추가합니다.

```
@Sse('sse')
sse(): Observable<MessageEvent> {
    반환 간격(1000).pipe(map(_ => ({ 데이터: { hello: 'world' } })));
}
```

정보 힌트 `@Sse()` 데코레이터와 메시지 이벤트 인터페이스는
에서 가져오고, 관찰 가능, 간격 및 지도는 rxjs 패키지에서 가져옵니다.

경고 서버에서 보낸 이벤트 경로는 관찰 가능한 스트림을 반환해야 합니다.

위의 예제에서는 실시간 업데이트를 전파할 수 있는 `sse`라는 경로를 정의했습니다. 이러한 이벤트는 [EventSource API](#)를 사용하여 수신할 수 있습니다.

`sse` 메서드는 여러 `MessageEvent`를 방출하는 `Observable`을 반환합니다(이 예제에서는 매초마다 새로운 `MessageEvent`를 방출합니다). `MessageEvent` 객체는 사양과 일치하도록 다음 인터페이스를 준수해야 합니다:

```
내보내기 인터페이스 MessageEvent { 데이
터: 문자열 | 객체;
id?: 문자열; 유형?:
문자열; 재시도?: 숫자
};
```

이제 클라이언트 측 애플리케이션에서 `이벤트 소스` 클래스의 인스턴스를 생성할 수 있으며, 생성자 인수로 (위에서 `@Sse()` 데코레이터에 전달한 엔드포인트와 일치하는) `/sse` 경로를 전달할 수 있습니다.

`EventSource` 인스턴스는 텍스트/이벤트 스트림 형식으로 이벤트를 전송하는 HTTP 서버에 대한 영구 연결을 엽니다. 연결은 `EventSource.close()`를 호출하여 닫을 때까지 열린 상태로 유지됩니다.

연결이 열리면 서버에서 들어오는 메시지가 이벤트 형태로 코드에 전달됩니다. 수신 메시지에 이벤트 필드가 있는 경우 트리거된 이벤트는 이벤트 필드 값과 동일합니다. 이벤트 필드가 없는 경우 일반 메시지 이벤트가 발생합니다([소스](#)).

```
const eventSource = new EventSource('/sse');
eventSource.onmessage = ({ data }) => {
  console.log('새 메시지', JSON.parse(data));
};
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

인증

인증은 대부분의 애플리케이션에서 필수적인 부분입니다. 인증을 처리하기 위한 다양한 접근 방식과 전략이 있습니다. 프로젝트마다 특정 애플리케이션 요구 사항에 따라 접근 방식이 달라집니다. 이 장에서는 다양한 요구 사항에 맞게 조정할 수 있는 인증에 대한 몇 가지 접근 방식을 소개합니다.

요구 사항을 구체화해 봅시다. 이 사용 사례에서 클라이언트는 사용자 이름과 비밀번호로 인증하는 것으로 시작합니다. 인증이 완료되면 서버는 인증을 증명하기 위해 후속 요청 시 인증 헤더에 [베어리 토큰으로](#) 전송할 수 있는 JWT를 발급합니다. 또한 유효한 JWT가 포함된 요청만 액세스할 수 있는 보호된 경로를 생성합니다.

첫 번째 요구 사항인 사용자 인증부터 시작하겠습니다. 그런 다음 JWT를 발행하여 이를 확장합니다. 마지막으로 요청에 대해 유효한 JWT를 검사하는 보호된 경로를 생성합니다.

인증 모듈 만들기

먼저 [AuthModule](#)을 생성하고 그 안에 AuthService와 AuthController를 생성하겠습니다. AuthService를 사용하여 인증 로직을 구현하고 AuthController를 사용하여 인증 엔드포인트를 노출할 것입니다.

```
nest g 모듈 인증  
nest g 컨트롤러 인증  
nest g 서비스 인증
```

AuthService를 구현하면서 사용자 작업을 [UsersService](#)에 캡슐화하는 것이 유용하다는 것을 알게 될 것이므로 이제 해당 모듈과 서비스를 생성해 보겠습니다:

```
nest g 모듈 사용자  
nest g 서비스 사용자
```

생성된 파일의 기본 내용을 아래와 같이 바꿉니다. 샘플 앱의 경우, 사용자 서비스는 단순히 하드코딩된 인메모리 사용자 목록과 사용자 이름으로 사용자를 검색하는 찾기 메서드를 유지 관리합니다. 실제 앱에서는 선택한 라이브러리(예: TypeORM, Sequelize, 몽구스 등)를 사용하여 사용자 모델과 지속성 레이어를 빌드할 수 있습니다.

@@파일명(사용자/사용자.서비스)

'@nestjs/common'에서 { Injectable }을 임포트합니다;

// 사용자 엔티티 내보내기 유형 User = any를 나타내는 실제 클래스/인터페이스여야 합니다;

@Injectable()

```
export 클래스 UsersController {
```

```
    private 읽기 전용 사용자 = [
```

```

{
  userId: 1, 사용자명:
  'john',
  비밀번호: 'changeme',
},
{
  userId: 2, 사용자 이
  름: 'maria', 비밀번호
  : 'guess',
},
];
}

async findOne(username: 문자열): Promise<사용자 | 정의되지 않음> {
  return this.users.find(사용자 => 사용자.사용자이름 === 사용자이름);
}
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트합니다;

@Injectable()
내보내기 클래스 UsersService {
  constructor() {
    this.users = [
      {
        userId: 1, 사용자명:
        'john',
        비밀번호: 'changeme',
      },
      {
        userId: 2, 사용자 이
        름: 'maria', 비밀번호
        : 'guess',
      },
    ];
  }

  async findOne(username) {
    반환 이.사용자.찾기(사용자 => 사용자.사용자 이름 === 사용자 이름);
  }
}

```

UsersModule에서 필요한 유일한 변경 사항은 UsersController를

모듈 데코레이터를 추가하여 이 모듈 외부에서 볼 수 있도록 합니다(곧 AuthService에서 사용하게 될 것입니다).

@@파일명(사용자/사용자.모듈)

'@nestjs/common'에서 { Module }을 가져오고,
'./users.service'에서 { UserService }를 가져옵니다;

모듈({

제공자: [UserService], 내보내기:
[UserService],

```
})
사용자 모듈 클래스 {} @@스위치 내보내기
'@nestjs/common'에서 { Module }을 가져오고,
'./users.service'에서 { UserService }를 가져옵니다;
다;
```

```
모듈 ({
  제공자: [UserService], 내보내기:
  [UserService],
})
사용자 모듈 클래스 {} 내보내기
```

"로그인" 엔드포인트 구현하기

AuthService는 사용자를 검색하고 비밀번호를 확인하는 작업을 수행합니다. 이를 위해 `signIn()` 메서드를 생성합니다. 아래 코드에서는 편리한 ES6 스프레드 연산자를 사용하여 사용자 객체에서 비밀번호 속성을 제거한 후 반환합니다. 이는 사용자 객체를 반환할 때 비밀번호나 기타 보안 키와 같은 민감한 필드를 노출하지 않으려는 일반적인 관행입니다.

@@파일명(auth/auth.service)

'@nestjs/common'에서 { Injectable, UnauthorizedException }을 임포트하고,
'./users/users.service'에서 { UserService }를 임포트합니다;

@Injectable()

내보내기 클래스 AuthService {
constructor(private userService: UserService) {}

```
async signIn(사용자 이름: 문자열, 패스: 문자열): Promise<any> {  
    const user = await this.userService.findOne(username); if  
(user?.password !== pass) {  
        새로운 UnauthorizedException()을 던집니다;  
    }  
    const { password, ...result } = 사용자;  
    // TODO: JWT를 생성하여 여기에 반환합니다.  
    // 대신 사용자 객체 반환 결과를 반  
    환합니다;  
}
```

}

@@switch

'@nestjs/common'에서 { Injectable, Dependencies, UnauthorizedException }을 임포
트합니다;

'./users/users.service'에서 { UserService }를 가져옵니다;

주입 가능()

@Dependencies(UserService) 내

보내기 클래스 AuthService {

```
constructor(usersService) {  
    this.userService = usersService;  
}
```

```
async signIn(username: 문자열, pass: 문자열) {
```

```
const user = await this.usersService.findOne(username);
if (user?.password !== pass) {
    새로운 UnauthorizedException()을 던집니다;
}
const { password, ...result } = 사용자;
// TODO: JWT를 생성하여 여기에 반환합니다.
// 대신 사용자 객체 반환 결과를 반
환합니다;
}
}
```

경고 경고 물론 실제 애플리케이션에서는 비밀번호를 일반 텍스트로 저장하지 않습니다. 대신 솔트 처리된 단방향 해시 알고리즘이 포함된 `bcrypt` 같은 라이브러리를 사용할 것입니다. 이 접근 방식을 사용하면 해시된 비밀번호만 저장한 다음 저장된 비밀번호를 들어오는 비밀번호의 해시된 버전과 비교하므로 사용자 비밀번호를 일반 텍스트로 저장하거나 노출하지 않습니다. 샘플 앱을 단순하게 유지하기 위해 이러한 절대적인 의무를 위반하고 일반 텍스트를 사용했습니다. 실제 앱에서는 이렇게 하지 마세요!

이제 `AuthModule`을 업데이트하여 `UsersModule`을 가져옵니다.

@@파일명(auth/auth.module)

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다;  
다;  
'./auth.controller'에서 { AuthController }를 임포트하고,  
' ../users/users.module'에서 { UsersModule }을 임포트합니다;
```

모듈({

```
    임포트: [UsersModule], 공급자:  
        [AuthService], 컨트롤러:  
        [AuthController],  
    })  
내보내기 클래스 AuthModule {}  
@@switch  
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다;  
다;  
'./auth.controller'에서 { AuthController }를 임포트하고,  
' ../users/users.module'에서 { UsersModule }을 임포트합니다;
```

모듈({

```
    임포트: [UsersModule], 공급자:  
        [AuthService], 컨트롤러:  
        [AuthController],  
    })  
내보내기 클래스 AuthModule {}
```

이제 **AuthController**를 열고 **signIn()** 메서드를 추가해 보겠습니다. 이 메서드는 클라이언트에서 사용자를 인증하기 위해 호출됩니다. 이 메서드는 요청 본문에서 사용자 이름과 비밀번호를 받고, 사용자가 인증되면 JWT 토큰을 반환합니다.

```

@@파일명(auth/auth.controller)
'@nestjs/common'에서 { Body, Controller, Post, HttpStatusCode, HttpStatus }를
임포트합니다;
'./auth.service'에서 { AuthService }를 가져옵니다;

컨트롤러('auth')
내보내기 클래스 AuthController {
  constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @Post('login')
  signIn(@Body() signInDto: Record<string, any>) {
    return this.authService.signIn(signInDto.username,
      signInDto.password);
  }
}

```

정보 힌트 이상적으로는 `Record<string, any>` 유형을 사용하는 대신 DTO 클래스를 사용하여 요청 본문의 모양을 정의하는 것이 좋습니다. 자세한 내용은 [유효성 검사 챕터](#)를 참조하세요.

JWT 토큰

이제 인증 시스템의 JWT 부분으로 넘어갈 준비가 되었습니다. 요구 사항을 검토하고 구체화해 보겠습니다:

- 사용자가 사용자 이름/비밀번호로 인증할 수 있도록 허용하여 보호된 API 엔드포인트에 대한 후속 호출에서 사용할 수 있도록 JWT를 반환합니다. 이 요구 사항을 충족하기 위한 작업은 순조롭게 진행 중입니다. 이를 완료하려면 JWT를 발급하는 코드를 작성해야 합니다.
- 무기명 토큰으로 유효한 JWT의 존재를 기반으로 보호되는 API 경로 만들기 JWT 요구 사항을 지원하

```
npm install --save @nestjs/jwt
```

정보 힌트 `@nestjs/jwt` 패키지(자세한 내용은 [여기](#)를 참조하세요)는 JWT 조작에 도움이 되는 유틸리티 패키지입니다. 여기에는 JWT 토큰 생성 및 확인이 포함됩니다.

서비스를 깔끔하게 모듈화하기 위해 `AuthService`에서 JWT 생성을 처리합니다. `auth` 폴더에서 `auth.service.ts` 파일을 열고 `JwtService`를 삽입한 다음 `로그인` 메서드를 업데이트하여 아래와 같이

JWT 토큰을 생성합니다:

@@파일명(auth/auth.service)

'@nestjs/common'에서 { Injectable, UnauthorizedException }을 임포트하고,
'./users/users.service'에서 { UsersService }를 임포트합니다;
'@nestjs/jwt'에서 { JwtService }를 가져옵니다;

@Injectable()

내보내기 클래스 AuthService {

```

생성자(
    비공개 usersService: UsersService, 비
    공개 jwtService: JwtService
) {}

async signIn(username, pass) {
    const user = await this.usersService.findOne(username);
    if (user?.password !== pass) {
        새로운 UnauthorizedException()을 던집니다;
    }
    const payload = { sub: user.userId, username: user.username };
    return {
        access_token: await this.jwtService.signAsync(payload),
    };
}
@@switch
'@nestjs/common'에서 { Injectable, Dependencies, UnauthorizedException }을 임포트
합니다;

'./users/users.service'에서 { UsersService }를 임포트하고,
'@nestjs/jwt'에서 { JwtService }를 임포트합니다;

@Dependencies(UsersService, JwtService)
@Injectable()
export class AuthService {
    constructor(usersService, jwtService) {
        this.usersService = usersService;
        this.jwtService = jwtService;
    }

    async signIn(username, pass) {
        const user = await this.usersService.findOne(username);
        if (user?.password !== pass) {
            새로운 UnauthorizedException()을 던집니다;
        }
        const payload = { username: user.username, sub: user.userId };
        return {
            access_token: await this.jwtService.signAsync(payload),
        };
    }
}

```

사용자 객체 프로퍼티의 하위 집합에서 JWT를 생성하기 위해 `signAsync()` 함수를 제공하는 `@nestjs/jwt` 라이브러리를 사용하고 있으며, 이 함수는 단일 `access_token` 프로퍼티가 있는 간단한 객체로 반환합니다. 참고: JWT 표준과 일관성을 유지하기 위해 `sub`라는 속성 이름을 선택하여 `userId` 값을 보유합니다. 인증 서비스에 `JwtService` 공급자를 삽입하는 것을 잊지 마세요.

이제 새 종속성을 가져오고 `JwtModule`을 구성하기 위해 `AuthModule`을 업데이트해야 합니다. 먼저 `auth` 폴더에 `constants.ts`를 생성하고 다음 코드를 추가합니다:



```
@@파일명(auth/constants) export
const jwtConstants = {
  비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게
  보관하세요.',
};

@@switch
export const jwtConstants = {
  비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게
  보관하세요.',
};
```

이 키를 사용하여 JWT 서명 및 확인 단계 간에 키를 공유합니다.

경고 경고 이 키를 공개적으로 노출하지 마세요. 여기서는 코드가 수행하는 작업을 명확히 하기 위해 공개 했지만, 프로덕션 시스템에서는 시크릿 볼트, 환경 변수 또는 구성 서비스 등의 적절한 조치를 사용하여 이 키를 보호해야 합니다.

이제 인증 폴더에서 `auth.module.ts`를 열고 다음과 같이 업데이트합니다:

@@파일명(auth/auth.module)

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다  
다;  
  
'./users/users.module'에서 { UsersModule }을 임포트하고,  
'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;  
import { AuthController } from './auth.controller';  
import { jwtConstants } './constants'에서 임포트합니다;
```

모듈({ import:

```
[  
  UsersModule,  
  JwtModule.register({  
    global: true,  
    비밀: jwtConstants.secret, signOptions:  
    { expiresIn: '60s' },  
  }),  
,  
제공자: [AuthService], 컨트롤러:  
[AuthController], 내보내기:  
[AuthService],  
})
```

내보내기 클래스 AuthModule {}

@@switch

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다  
다;  
  
'./users/users.module'에서 { UsersModule }을 임포트하고,  
'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;  
import { AuthController } from './auth.controller';  
import { jwtConstants } './constants'에서 임포트합니다;
```

모듈({ import:

```
[
```

```
UsersModule,  
JwtModule.register({  
  global: true,  
  비밀: jwtConstants.secret, signOptions:  
  { expiresIn: '60s' },  
}),  
],  
공급자: [AuthService], 컨트롤러:  
[AuthController], 내보내기:  
[AuthService],  
})  
  
내보내기 클래스 AuthModule {}
```

힌트 힌트 작업을 더 쉽게 하기 위해 `JwtModule`을 전역으로 등록하고 있습니다. 즉, 애플리케이션의 다른 곳에서는 `JwtModule`을 임포트할 필요가 없습니다.

구성 객체를 전달하여 `register()`를 사용하여 `JwtModule`을 구성합니다. Nest `JwtModule`에 대한 자세한 내용은 [여기를](#), 사용 가능한 구성 옵션에 대한 자세한 내용은 [여기를](#) 참조하세요.

계속해서 cURL을 사용하여 경로를 다시 테스트해 보겠습니다. `UserService`에 하드코딩된 모든 `사용자` 객체로 테스트할 수 있습니다.

```
인증/로그인에 $ # POST 보내기  
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",  
"password": "changeme"}' -H "Content-Type: application/json"  
{"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
```

참고: 위 JWT는 잘렸습니다.

인증 가드 구현

이제 마지막 요구 사항인 요청에 유효한 JWT가 있어야 엔드포인트를 보호하는 문제를 해결할 수 있습니다. 경로를 보호하는 데 사용할 수 있는 `AuthGuard`를 생성하여 이를 수행합니다.

@@파일명(auth/auth.guard)

```
import {  
    CanActivate,  
    ExecutionContext,  
    Injectable,  
    UnauthorizedException,  
} from '@nestjs/common';  
import { JwtService } from '@nestjs/jwt';  
import { jwtConstants } from './constants';  
  
import { Request } from 'express';  
  
@Injectable()  
export class AuthGuard implements CanActivate {  
    constructor(private jwtService: JwtService) {}  
  
    async canActivate(context: ExecutionContext): Promise<boolean> {
```

```
const request = context.switchToHttp().getRequest();
const token = this.extractTokenFromHeader(request);
if (!token) {
    새로운 UnauthorizedException()을 던집니다;
}
try {
    const payload = await this.jwtService.verifyAsync(
        token,
        {
            비밀: jwtConstants.secret
        }
    );
    // □ 여기서 페이로드를 요청 객체에 할당합니다.

    //를 추가하여 라우트 핸들러 요청['user'] = 페이로드에서 액세스할 수
    있도록 합니다;
} catch {
    새로운 UnauthorizedException()을 던집니다;
}
참을 반환합니다;

}

private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    반환 유형 === '무기명' ? 토큰 : undefined;
}
}
```

이제 보호 경로를 구현하고 이를 보호하기 위해 [AuthGuard](#)를 등록할 수 있습니다.

`auth.controller.ts` 파일을 열고 아래와 같이 업데이트합니다:

@@파일명(auth.controller) 가져오

기 {

본문, 컨트롤러,

가져오기,

HttpCode,

HttpStatus,

게시, 요청, 사

용가드

}를 '@nestjs/common'에서 가져옵니다;

'./auth.guard'에서 { AuthGuard }를 가져오고,

'./auth.service'에서 { AuthService }를 가져옵니

다;

컨트롤러('auth')

```
내보내기 클래스 AuthController {
    constructor(private authService: AuthService) {}

    @HttpCode(HttpStatus.OK)
    @Post('login')
    signIn(@Body() signInDto: Record<string, any>) {
```

```

    this.authService.signIn(signInDto.사용자이름, signInDto.비밀번호)을 반환
합니다;
}

UseGuards(AuthGuard)
@Get('profile')
getProfile(@Request() req) {
    req.user를 반환합니다;
}
}

```

방금 생성한 AuthGuard를 `GET /profile` 경로에 적용하여 보호되도록 합니다. 앱이 실행 중인지 확인하고

`cURL`을 사용하여 경로를 테스트합니다.

```

$ # GET /profile
curl http://localhost:3000/auth/profile
{"statusCode":401,"message":"승인되지 않음"}

$ # POST /auth/login
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
{"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."}

```

이전 단계에서 무기명 코드로 반환된 `access_token`을 사용하여 `/profile`을 GET합니다.

```

curl http://localhost:3000/auth/profile -H "인증: 무기명
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm...
{"sub":1,"username":"john","iat":..., "exp":...}

```

인증 모듈에서 JWT의 만료 시간을 **60초**로 구성했습니다. 이는 너무 짧은 만료 시간이며 토큰 만료 및 새로 고침에 대한 세부 사항을 다루는 것은 이 문서의 범위를 벗어납니다. 하지만 JWT의 중요한 특성을 보여주기 위해 이 방법을 선택했습니다. 인증 후 60초를 기다렸다가 `GET /auth/프로필` 요청을 시도하면 **401 권한 없음** 응답을 받게 됩니다. 이는 `@nestjs/jwt`가 자동으로 JWT의 만료 시간을 확인하므로 애플리케이션에서 직접 확인해야 하는 수고를 덜어주기 때문입니다.

이제 JWT 인증 구현이 완료되었습니다. 이제 자바스크립트 클라이언트(예: Angular/React/Vue) 및 기타 자바스크립트 앱이 트위터의 API 서버와 안전하게 인증하고 통신할 수 있습니다.

전 세계적으로 인증 사용

대부분의 엔드포인트를 기본적으로 보호해야 하는 경우, 인증 가드를 **전역 가드로** 등록하고 각 컨트롤러 위에

`@UseGuards()` 데코레이터를 사용하는 대신 어떤 경로를 공개해야 하는지 플래그를 지정하면 됩니다.

먼저, 다음 구성을 사용하여 `AuthGuard`을 전역 가드로 등록합니다(예: `AuthModule` 등 모든 모듈에서):

```
제공자: [
  {
    제공: APP_GUARD, useClass:
    AuthGuard,
  },
],
```

이 설정이 완료되면 Nest는 모든 엔드포인트에 `AuthGuard`를 자동으로 바인딩합니다.

이제 경로를 공개로 선언하는 메커니즘을 제공해야 합니다. 이를 위해 `SetMetadata` 데코레이터 팩토리 함수를 사용하여 사용자 정의 데코레이터를 만들 수 있습니다.

```
import { SetMetadata } from '@nestjs/common';

export const IS_PUBLIC_KEY = 'isPublic';
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

위 파일에서 두 개의 상수를 내보냈습니다. 하나는 `IS_PUBLIC_KEY`라는 메타데이터 키이고, 다른 하나는 `Public`이라고 부를 새 데코레이터 자체입니다(프로젝트에 맞는 다른 이름을 지정할 수 있습니다).

이제 사용자 정의 `@Public()` 데코레이터가 생겼으므로 다음과 같이 모든 메서드를 데코레이션하는 데 사용할 수 있습니다:

```
Public()
@Get()
findAll() {
  반환 [];
}
```

마지막으로, `"isPublic"` 메타데이터가 발견되면 `AuthGuard`가 참을 반환하도록 해야 합니다. 이를 위해 `Reflector` 클래스를 사용하겠습니다(자세한 내용은 [여기](#)를 참조하세요).

```
@Injectable()
내보내기 클래스 AuthGuard 구현 CanActivate { 생성자(private jwtService:
JwtService, 비공개 리플렉터:
리플렉터) {}

async canActivate(context: ExecutionContext): Promise<boolean> {
  const isPublic = this.reflector.getAllAndOverride<boolean>
(is_public_key, [
    context.getHandler(),
    context.getClass(),
  ]);
  if (isPublic) {
    // 이 조건이 참을 반환하는
    것을 참조하십시오;
```

```
}

const request = context.switchToHttp().getRequest();
const token = this.extractTokenFromHeader(request);
if (!token) {
    새로운 UnauthorizedException()을 던집니다;
}
try {
    const payload = await this.jwtService.verifyAsync(token, {
        secret: jwtConstants.secret,
    });
    // □ 여기서 페이로드를 요청 객체에 할당합니다.

    //를 추가하여 라우트 핸들러 요청['user'] = 페이로드에서 액세스할 수
    있도록 합니다;
} catch {
    새로운 UnauthorizedException()을 던집니다;
}
참을 반환합니다;

}

private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    반환 유형 === '무기명' ? 토큰 : undefined;
}
}
```

여권 통합

Passport는 커뮤니티에서 잘 알려져 있고 많은 프로덕션 애플리케이션에서 성공적으로 사용되는 가장 인기 있는 node.js 인증 라이브러리입니다. 이 라이브러리를 Nest 애플리케이션과 통합하는 방법은 [@nestjs/passport](#) 모듈을 사용하면 간단합니다.

Passport와 NestJS를 통합하는 방법을 알아보려면 이 [챕터를](#) 확인하세요. 예제

이 장의 전체 코드 버전은 [여기에서](#) 확인할 수 있습니다.

권한 부여

권한 부여는 사용자가 수행할 수 있는 작업을 결정하는 프로세스를 말합니다. 예를 들어 관리 사용자는 글을 작성, 편집 및 삭제할 수 있습니다. 관리자가 아닌 사용자에게는 게시물을 읽을 수 있는 권한만 부여됩니다.

권한 부여는 인증과 직교하며 독립적입니다. 그러나 권한 부여에는 인증 메커니즘이 필요합니다.

인증을 처리하는 방법과 전략에는 여러 가지가 있습니다. 모든 프로젝트에 대해 취하는 접근 방식은 특정 애플리케이션 요구 사항에 따라 다릅니다. 이 장에서는 다양한 요구 사항에 맞게 조정할 수 있는 몇 가지 권한 부여 접근 방식을 소개합니다.

기본 RBAC 구현

역할 기반 액세스 제어(RBAC)는 역할과 권한을 중심으로 정의된 정책 중립적인 액세스 제어 메커니즘입니다. 이 섹션에서는 Nest [가드를](#) 사용하여 매우 기본적인 RBAC 메커니즘을 구현하는 방법을 보여드리겠습니다.

먼저 시스템에서 역할을 나타내는 [역할](#) 열거형을 만들어 보겠습니다:

```
@@파일 이름(role.enum)
내보내기 열거형 Role {
    사용자 = '사용자',
    관리자 = '관리자',
}
```

정보 힌트 보다 정교한 시스템에서는 데이터베이스 내에 역할을 저장하거나 외부 인증 공급자로부터 역할을 가져올 수 있습니다.

이제 [@Roles\(\)](#) 데코레이터를 만들 수 있습니다. 이 데코레이터를 사용하면 특정 리소스에 액세스하는 데 필요한 역할을 지정할 수 있습니다.

@@파일명(역할.데코레이터)

'@nestjs/common'에서 { SetMetadata }를 가져오고,

'./enums/role.enum'에서 { Role }을 가져옵니다;

```
export const ROLES_KEY = 'roles';
export const Roles = (...roles: Role[]) => SetMetadata(ROLES_KEY, roles);
```

@@switch

'@nestjs/common'에서 { SetMetadata }를 가져옵니다;

```
export const ROLES_KEY = 'roles';
export const Roles = (...roles) => SetMetadata(ROLES_KEY, roles);
```

이제 사용자 지정 `@Roles()` 데코레이터가 있으므로 이를 사용하여 모든 라우트 핸들러를 데코레이션할 수 있습니다.

```
@@파일명(cats.controller) @Post()
@Roles(Role.Admin)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@스위치

@포스트()
@Roles(Role.Admin)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

마지막으로, 현재 사용자에게 할당된 역할을 현재 처리 중인 경로에 필요한 실제 역할과 비교하는 `RolesGuard` 클래스를 생성합니다. 경로의 역할(사용자 정의 메타데이터)에 액세스하기 위해 프레임워크에서 기본 제공되고 `@nestjs/core` 패키지에서 노출되는 `Reflector` 헬퍼 클래스를 사용합니다.

@@파일명(roles.guard)

'@nestjs/common'에서 { Injectable, CanActivate, ExecutionContext }를 임포트합니다;

'@nestjs/core'에서 { Reflector }를 가져옵니다;

```
@Injectable()
export class RolesGuard implements CanActivate {
    constructor(private reflector: Reflector) {}

    canActivate(context: ExecutionContext): boolean {
        const requiredRoles = this.reflector.getAllAndOverride<Role[]>(ROLES_KEY, [
            context.getHandler(),
            context.getClass(),
        ]);
        if (!requiredRoles) { 참을 반환합니다;
        }
        const { user } = context.switchToHttp().getRequest();
        반환 requiredRoles.some((role) => user.roles?.includes(role));
    }
}
@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 임포트하고,
'@nestjs/core'에서 { Reflector }를 임포트합니다;
```

주입 가능() @의존성(반사기) 내보내기

```
클래스 RolesGuard {
    constructor(reflector) {
        this.reflector = reflector;
```

```

    }

    canActivate(context) {
      const requiredRoles = this.reflector.getAllAndOverride(ROLES_KEY, [
        context.getHandler(),
        context.getClass(),
      ]);
      if (!requiredRoles) { 참을
        반환합니다;
      }
      const { user } = context.switchToHttp().getRequest();
      반환 requiredRoles.일부((role) => user.roles.includes(role));
    }
}

```

정보 힌트 상황에 맞는 방식으로 리플렉터를 활용하는 방법에 대한 자세한 내용은 실행 컨텍스트 장

의 [리플렉션 및 메타데이터](#) 섹션을 참조하세요.

경고 이 예제는 라우트 핸들러 수준에서 역할의 존재 여부만 확인하므로 "기본"으로 명명되었습니다. 실제 애플리케이션에서는 여러 작업을 포함하는 엔드포인트/핸들러가 있을 수 있으며, 각 작업에는 특정 권한 집합이 필요할 수 있습니다. 이 경우 비즈니스 로직 내 어딘가에 역할을 확인하는 메커니즘을 제공해야 하며, 권한을 특정 작업과 연결하는 중앙 집중식 위치가 없기 때문에 유지 관리가 다소 어려워집니다.

이 예제에서는 `요청.user`에 사용자 인스턴스와 허용된 역할이 포함되어 있다고 가정했습니다(

`역할` 속성). 앱에서는 사용자 지정 인증 가드에서 해당 연결을 만들 수 있습니다.

- 자세한 내용은 [인증](#) 챕터를 참조하세요.

이 예제가 제대로 작동하려면 `사용자` 클래스의 모양이 다음과 같아야 합니다:

```

사용자 클래스 {
  // ... 다른 속성 역할: Role[];
}

```

마지막으로, 컨트롤러 수준에서 또는 전역적으로 RolesGuard를 등록하세요:

```

제공자: [
  {
    제공: APP_GUARD, useClass:
    RolesGuard,
  },
],

```

권한이 부족한 사용자가 엔드포인트를 요청하면 Nest는 자동으로 다음과 같은 응답을 반환합니다:

```
{
  "상태코드": 403,
  "메시지": "금지된 리소스", "오류": "금지됨"
}
```

정보 힌트 다른 오류 응답을 반환하려면 부울 값을 반환하는 대신 고유한 특정 예외를 던져야 합니다.

클레임 기반 권한 부여

ID가 생성되면 신뢰할 수 있는 당사자가 발급한 하나 이상의 클레임이 할당될 수 있습니다. 클레임은 주체가 무엇인지가 아니라 주체가 할 수 있는 일을 나타내는 이름-값 쌍입니다.

Nest에서 클레임 기반 권한 부여를 구현하려면 위의 [RBAC](#) 섹션에서 설명한 것과 동일한 단계를 따르되 한 가지 중요한 차이점이 있는데, 특정 역할을 확인하는 대신 권한을 비교해야 한다는 점입니다. 모든 사용자에게는 일련의 권한이 할당됩니다. 마찬가지로 각 리소스/엔드포인트는 해당 리소스/엔드포인트에 액세스하는 데 필요한 권한을 정의합니다(예: 전용 [@RequirePermissions\(\)](#) 데코레이터를 통해).

```
@@파일명(cats.controller) @Post()
@RequirePermissions(Permission.CREATE_CAT)
create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}

@@스위치

@포스트()
@RequirePermissions(Permission.CREATE_CAT)
@Bind(Body())
create(createCatDto) {
  this.catsService.create(createCatDto);
}
```

정보 힌트 위의 예에서 [권한](#)(RBAC 섹션에서 설명한 역할과 유사)은 시스템에서 사용 가능한 모든 권한을 포함하는 TypeScript 열거형입니다.

CASL 통합

CASL은 특정 클라이언트가 액세스할 수 있는 리소스를 제한하는 동형 권한 부여 라이브러리입니다. 점진적으로 채택할 수 있도록 설계되었으며 간단한 클레임 기반부터 완전한 기능을 갖춘 주제 및 속성 기반 권한 부여까지 쉽

게 확장할 수 있습니다.

시작하려면 먼저 `@casl/ability` 패키지를 설치하세요:

```
$ npm i @casl/ability
```

정보 힌트 이 예제에서는 CASL을 선택했지만 액세스제어나
acl을 사용할 수 있습니다.

설치가 완료되면 CASL의 메커니즘을 설명하기 위해 두 개의 엔티티 클래스를 정의하겠습니다: User와 Article입니다.

```
사용자 클래스 {  
    id: 숫자;  
    isAdmin: 부울입니다;  
}
```

사용자 클래스는 고유한 사용자 식별자인 id와 사용자에게 관리자 권한이 있는지 여부를 나타내는 isAdmin의 두 가지 속성으로 구성됩니다.

```
기사 클래스 { id: 숫자;  
    isPublished: 부울; authorId: 숫자;  
}
```

문서 클래스에는 각각 id, isPublished, authorId라는 세 가지 속성이 있습니다. id는 고유한 문서 식별자이고, isPublished는 문서가 이미 게시되었는지 여부를 나타내며, authorId는 문서를 작성한 사용자의 ID입니다.

이제 이 예제에 대한 요구 사항을 검토하고 구체화해 보겠습니다:

- 관리자는 모든 엔티티를 관리(생성/읽기/업데이트/삭제)할 수 있습니다.
- 사용자는 모든 항목에 읽기 전용 액세스 권한이 있습니다.
- 사용자는 자신의 글을 업데이트할 수 있습니다(`article.authorId === userId`).
- 이미 게시된 글은 삭제할 수 없습니다(`article.isPublished === true`).

이를 염두에 두고 사용자가 엔티티로 수행할 수 있는 모든 가능한 작업을 나타내는 Action 열거형을 만드는 것으로 시작할 수 있습니다:

```
export enum Action {  
    Manage = '관리',  
    Create = '만들기',  
    Read = '읽기',  
    Update = '업데이트',  
    Delete = '삭제',
```

경고 알림 관리는 CASL에서 "모든" 작업을 나타내는 특수 키워드입니다.

CASL 라이브러리를 캡슐화하기 위해 이제 CaslModule과 CaslAbilityFactory를 생성해 보겠습니다.

```
nest g 모듈 casl
nest g 클래스 casl/casl-ability.factory
```

이렇게 하면 `CaslAbilityFactory`에서 `createForUser()` 메서드를 정의할 수 있습니다. 이 메서드는 주어진 사용자에 대한 어빌리티 객체를 생성합니다:

```
유형 Subjects = InferSubjects<기사 유형 | 사용자 유형> | '모두'; 내보내기 유형
```

```
AppAbility = Ability<[Action, Subjects]>;
```

```
@Injectable()
export class CaslAbilityFactory {
  createForUser(user: User) {
    const { 할 수 있다, 할 수 없다, 빌드 } = 새로운 AbilityBuilder<
      Ability<[액션, 서브젝트]>
    >(어빌리티를 어빌리티클래스<앱어빌리티>로);

    if (user.isAdmin) {
      can(Action.Manage, 'all'); // 모든 것에 대한 읽기-쓰기 액세스 권한
    } else {
      can(Action.Read, 'all'); // 모든 것에 대한 읽기 전용 액세스
    }

    can(Action.Update, Article, { authorId: user.id });
    cannot(Action.Delete, Article, { isPublished: true });

    반환 빌드({
      // 자세한 내용은 https://casl.js.org/v5/en/guide/subject-type-detection#use-classes-as-subject-types를 참조하세요.
      detectSubjectType: (item) =>
        item.constructor은 ExtractSubjectType<Subjects>로 설정합니다,
        경고 모든는 CASL에서 '모든 주제'를 나타내는 특수 키워드입니다.
    });
  }
}
```

정보 힌트 어빌리티, 어빌리티빌더, 어빌리티클래스, 추출주제유형 클래스는 `@casl/ability` 패키지에 서 내보냅니다.
정보 힌트 `detectSubjectType` 옵션을 사용하면 CASL이 객체에서 주제 유형을 가져오는 방법을 이해 할 수 있습니다. 자세한 내용은 [CASL 설명서](#)를 참조하세요.

위의 예시에서는 `AbilityBuilder` 클래스를 사용하여 `Ability` 인스턴스를 만들었습니다. 짐작하셨겠지만, 수

락할 수 있는 것과 수락할 수 없는 것은 같은 인수를 받지만 의미가 다르며, 수락할 수 있는 것은 지정된 대상에 대해 작업을 수행할 수 있고 금지할 수 없습니다. 둘 다 최대 4개의 인수를 받을 수 있습니다. 이러한 함수에 대해 자세히 알아보려면 공식 [CASL 문서](#)를 참조하세요.

마지막으로, `CaslAbilityFactory`를 제공자 및 내보내기 배열에 추가해야 합니다.

`CaslModule` 모듈 정의:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./casl-ability.factory'에서 { CaslAbilityFactory }를 가져옵니다;

모듈({
    제공자: [CaslAbilityFactory], 내보내기:
    [CaslAbilityFactory],
})

내보내기 클래스 CaslModule {}
```

이렇게 하면 호스트 컨텍스트에서 `CaslModule`을 임포트하기만 하면 표준 생성자 주입을 사용하여 모든 클래스에 `CaslAbilityFactory`를 주입할 수 있습니다:

```
constructor(private caslAbilityFactory: CaslAbilityFactory) {}
```

그런 다음 다음과 같이 수업에서 사용하세요.

```
const ability = this.caslAbilityFactory.createForUser(user);
if (ability.can(Action.Read, 'all')) {
    // "사용자"는 모든 항목에 대한 읽기 권한이 있습니다.
}
```

정보 힌트 공식 CASL 문서에서 `어빌리티` 클래스에 대해 자세히 알아보세요.

예를 들어 관리자가 아닌 사용자가 있다고 가정해 보겠습니다. 이 경우 사용자는 문서를 읽을 수 있어야 하지만 새 문서를 만들거나 기존 문서를 삭제하는 것은 금지되어야 합니다.

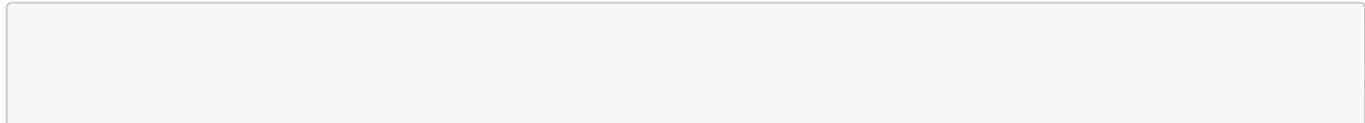
```
const user = new User();
user.isAdmin = false;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Read, Article); // 참
ability.can(Action.Delete, Article); // 거짓
ability.can(Action.Create, Article); // 거짓
```

정보 힌트 Ability와 `AbilityBuilder` 클래스 모두 할 수 있는 메서드와 할 수 없는 메서드를 제공하지

만 용도가 다르고 약간 다른 인수를 허용합니다.

또한 요구사항에 명시한 대로 사용자가 문서를 업데이트할 수 있어야 합니다:



```
const user = new User();
user.id = 1;

const article = new Article();
article.authorId = user.id;

const ability = this.caslAbilityFactory.createForUser(user);
ability.can(Action.Update, article); // true

article.authorId = 2;
ability.can(Action.Update, article); // false
```

보시다시피 **Ability** 인스턴스를 사용하면 매우 읽기 쉬운 방식으로 권한을 확인할 수 있습니다. 마찬가지로 **AbilityBuilder**를 사용하면 비슷한 방식으로 권한을 정의하고 다양한 조건을 지정할 수 있습니다. 더 많은 예제를 보려면 공식 문서를 참조하세요.

고급: 정책 가드 구현하기

이 섹션에서는 메서드 수준에서 구성할 수 있는 특정 권한 부여 정책을 사용자가 충족하는지 확인하는 다소 정교한 가드를 구축하는 방법을 보여드리겠습니다(클래스 수준에서 구성된 정책도 존중하도록 확장할 수 있습니다). 이 예제에서는 예시를 보여주기 위해 CASL 패키지를 사용하지만 이 라이브러리를 반드시 사용해야 하는 것은 아닙니다. 또한 이전 섹션에서 생성한 **CaslAbilityFactory** 프로바이더를 사용할 것입니다.

먼저 요구 사항을 구체화해 보겠습니다. 목표는 라우트 핸들러별로 정책 검사를 지정할 수 있는 메커니즘을 제공하는 것입니다. 보다 간단한 검사와 함수형 코드를 선호하는 사용자를 위해 객체와 함수를 모두 지원할 것입니다.

정책 처리기를 위한 인터페이스를 정의하는 것부터 시작하겠습니다:

```
'.../casl/casl-ability.factory'에서 { AppAbility } 임포트; 인터페이스 IPolicyHandler {
    handle(ability: AppAbility): boolean;
}

유형 PolicyHandlerCallback = (ability: AppAbility) => boolean;

export 유형 PolicyHandler = IPolicyHandler | PolicyHandlerCallback;
```

위에서 언급했듯이 정책 처리기를 정의하는 두 가지 가능한 방법, 즉 객체(`IPolicyHandler` 인터페이스를 구현하는 클래스의 인스턴스)와 함수(`PolicyHandlerCallback` 유형을 층족하는)를 제공했습니다.

이를 통해 `@CheckPolicies()` 데코레이터를 만들 수 있습니다. 이 데코레이터를 사용하면 특정 리소스에 액세스하기 위해 어떤 정책을 층족해야 하는지 지정할 수 있습니다.

```
export const CHECK_POLICIES_KEY = 'check_policy';
export const CheckPolicies = (...handlers: PolicyHandler[]) =>
  SetMetadata(CHECK_POLICIES_KEY, handlers);
```

이제 라우트 핸들러에 바인딩된 모든 정책 핸들러를 추출하고 실행하는 `PoliciesGuard`를 만들어 보겠습니다.

```
@Injectable()
내보내기 클래스 PoliciesGuard 구현 CanActivate { 생성자(
  개인 리플렉터: 리플렉터,
  private caslAbilityFactory: CaslAbilityFactory,
) {}

async canActivate(context: ExecutionContext): Promise<boolean> {
  const policyHandlers =
    this.reflector.get<정책핸들러[]>(
      CHECK_POLICIES_KEY,
      컨텍스트.getHandler(),
    ) || [];

  const { user } = context.switchToHttp().getRequest();
  const ability = this.caslAbilityFactory.createForUser(user);

  반환 정책 핸들러.every((핸들러) => this.execPolicyHandler(핸들러, 능력),
);
}

private execPolicyHandler(handler: PolicyHandler, ability: AppAbility) {
  if (typeof handler === 'function') {
    반환 핸들러(어빌리티);
  }
  핸들러.핸들(ability)을 반환합니다;
}
}
```

정보 힌트 이 예제에서는 `요청.user`에 사용자 인스턴스가 포함되어 있다고 가정했습니다. 앱에서는 사용자 지정 인증 가드에서 해당 연결을 만들 수 있습니다(자세한 내용은 [인증](#) 챕터를 참조하세요).

이 예제를 자세히 살펴봅시다. 정책 핸들러는 `@CheckPolicies()` 데코레이터를 통해 메서드에 할당된 핸들러의 배열입니다. 다음으로, 사용자가 특정 작업을 수행할 수 있는 충분한 권한을 가지고 있는지 확인할 수 있도록 `Ability` 객체를 구성하는 `CaslAbilityFactory#create` 메서드를 사용합니다. 이 객체를 정책 핸들러에 전달하는데, 이 핸들러는 함수이거나 `IPolicyHandler`를 구현하는 클래스의 인스턴스이며, 부울을 반환하는 `handle()` 메서드를 노출합니다. 마지막으로, 모든 핸들러가 `참` 값을 반환하는지 확인하기 위해 `Array#every`

메서드를 사용합니다.

마지막으로 이 가드를 테스트하려면 다음과 같이 라우트 핸들러에 바인딩하고 인라인 정책 핸들러(기능적 접근 방식)를 등록합니다:

```
Get()
@UseGuards(PoliciesGuard)
CheckPolicies((ability: AppAbility) => ability.can(Action.Read, Article))
findAll() {
    이.기사서비스.모두 찾기()를 반환합니다;
}
```

또는 `IPolicyHandler` 인터페이스를 구현하는 클래스를 정의할 수도 있습니다:

```
export class ReadArticlePolicyHandler implements IPolicyHandler {
    handle(ability: AppAbility) {
        ability.can(Action.Read, Article)을 반환합니다;
    }
}
```

그리고 다음과 같이 사용하세요:

```
Get()
@UseGuards(PoliciesGuard)
CheckPolicies(new ReadArticlePolicyHandler())
findAll() {
    이.기사서비스.모두 찾기()를 반환합니다;
}
```

경고 주의 `새` 키워드를 사용하여 정책 핸들러를 제자리에 인스턴스화해야 하므로

`ReadArticlePolicyHandler` 클래스는 종속성 주입을 사용할 수 없습니다. 이 문제는 `ModuleRef#get` 메서드를 사용하여 해결할 수 있습니다(자세한 내용은 [여기](#)를 참조하세요). 기본적으로

`@CheckPolicies()` 데코레이터를 통해 함수와 인스턴스를 등록하는 대신 `Type<IPolicyHandler>`

전달을 허용해야 합니다. 그런 다음 가드 내부에서 유형 참조를 사용하여 인스턴스를 검색할 수 있습니다: `moduleRef.get(YOUR_HANDLER_TYPE)` 또는 `ModuleRef#create` 메서드를 사용하여 인스턴스를 동적으로 인스턴스화할 수도 있습니다.

암호화 및 해싱

암호화는 정보를 인코딩하는 과정입니다. 이 프로세스는 일반 텍스트라고 하는 정보의 원래 표현을 암호 텍스트라고 하는 대체 형식으로 변환합니다. 이상적으로는 권한이 있는 당사자만 암호 텍스트를 다시 일반 텍스트로 해독하여 원본 정보에 액세스할 수 있습니다. 암호화는 그 자체로 간섭을 방지하는 것이 아니라 가로채려는 사람이 이해할 수 있는 콘텐츠를 거부합니다. 암호화는 양방향 기능이며, 암호화된 내용은 적절한 키를 사용하여 해독할 수 있습니다.

해싱은 주어진 키를 다른 값으로 변환하는 과정입니다. 해시 함수는 수학적 알고리즘에 따라 새로운 값을 생성하는데 사용됩니다. 해싱이 완료되면 출력에서 입력으로 변경하는 것이 불가능해야 합니다.

암호화

Node.js는 문자열, 숫자, 버퍼, 스트림 등을 암호화하고 해독하는 데 사용할 수 있는 내장 [암호화 모듈](#)을 제공합니다. Nest 자체는 불필요한 추상화를 피하기 위해 이 모듈 위에 추가 패키지를 제공하지 않습니다.

예를 들어 AES(고급 암호화 시스템) '[aes-256-ctr](#)' 알고리즘 CTR 암호화 모드를 사용하겠습니다.

```
'crypto'에서 { createCipheriv, randomBytes, scrypt }를 가져오고,  
'util'에서 { promiseify }를 가져옵니다;  
  
const iv = randomBytes(16);  
const password = '키 생성에 사용된 비밀번호';  
  
// 키 길이는 알고리즘에 따라 다릅니다.  
// 이 경우 aes256의 경우 32바이트입니다.  
const key = (await promiseify(scrypt)(password, 'salt', 32)) as  
Buffer; const cipher = createCipheriv('aes-256-ctr', key, iv);  
  
const textToEncrypt = 'Nest';  
const encryptedText = Buffer.concat([  
  cipher.update(textToEncrypt),  
  cipher.final(),  
]);
```

이제 암호화된 텍스트 값을 해독합니다:

'crypto'에서 { createDecipheriv }를 가져옵니다;

```
const decipher = createDecipheriv('aes-256-ctr', key, iv);
const decryptedText = Buffer.concat([
  decipher.update(encryptedText),
  decipher.final(),
]);
```

해싱

해싱의 경우, [bcrypt](#) 또는 [argon2](#) 패키지를 사용하는 것을 권장합니다. Nest 자체는 불필요한 추상화를 도입하지 않기 위해(학습 곡선을 짧게 만들기 위해) 이러한 모듈 위에 추가 래퍼를 제공하지 않습니다.

예를 들어, bcrypt를 사용하여 임의의 비밀번호를 해시해 보겠습니다.

먼저 필요한 패키지를 설치합니다:

```
$ npm i bcrypt
$ npm i -D @types/bcrypt
```

설치가 완료되면 다음과 같이 [해시](#) 함수를 사용할 수 있습니다:

```
import * as bcrypt from 'bcrypt';

const saltOrRounds = 10;
const password = 'random_password';
const hash = await bcrypt.hash(password, saltOrRounds);
```

소금을 생성하려면 [genSalt](#) 함수를 사용합니다:

```
const salt = await bcrypt.genSalt();
```

비밀번호를 비교/확인하려면 [비교](#) 기능을 사용하세요:

```
const isMatch = await bcrypt.compare(비밀번호, 해시);
```

사용 가능한 기능에 대한 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

헬멧

헬멧은 HTTP 헤더를 적절히 설정하여 잘 알려진 웹 취약점으로부터 앱을 보호할 수 있습니다. 일반적으로 Helmet은 보안 관련 HTTP 헤더를 설정하는 작은 미들웨어 함수의 모음에 불과합니다([자세히 보기](#)).

정보 힌트 헬멧을 전역으로 적용하거나 등록하는 것은 다른 호출보다 먼저 이루어져야 합니다.
앱.사용() 또는 앱.사용()을 호출할 수 있는 설정 함수를 사용할 수 없습니다. 이는 미들웨어/경로가 정의되는 순서가 중요한 기본 플랫폼(예: Express 또는 Fastify)의 작동 방식 때문입니다. 경로를 정의한 후에 헬멧이나 코어와 같은 미들웨어를 사용하는 경우 해당 미들웨어는 해당 경로에 적용되지 않으며 미들웨어 다음에 정의된 경로에만 적용됩니다.

Express와 함께 사용(기본값)

필요한 패키지를 설치하는 것으로 시작하세요.

```
npm i --save helmet
```

설치가 완료되면 글로벌 미들웨어로 적용합니다.

```
'헬멧'에서 헬멧을 가져옵니다;  
// 초기화 파일 어딘가에 앱.사용(헬멧());
```

경고 헬멧, [@apollo/server](#)(4.x) 및 Apollo 샌드박스를 사용할 때 Apollo 샌드박스에서 CSP에 문제가 있을 수 있습니다. 이 문제를 해결하려면 아래 그림과 같이 CSP를 구성하세요:

```
app.use(helmet({
  crossOriginEmbedderPolicy: false,
  contentSecurityPolicy: {
    지시어를 사용합니다: {
      imgSrc: [`'self'`, 'data:', 'apollo-server-landing-
page.cdn.apollographql.com'],
      scriptSrc: [`'self'`, `https: 'unsafe-inline'`],
      manifestSrc: [`'self'`, 'apollo-server-landing-
page.cdn.apollographql.com'],
      frameSrc: [`'self'`, 'sandbox.embed.apollographql.com'],
    },
  },
}));
```

Fastify와 함께 사용

FastifyAdapter를 사용하는 경우 [@fastify/helmet](#) 패키지를 설치합니다:

```
npm i --save @fastify/helmet
```

fastify-helmet을 미들웨어가 아닌 [Fastify 플러그인으로](#) 사용해야 합니다(즉, `app.register()` 사용):

```
'@fastify/helmet'에서 헬멧 가져오기
```

```
// 초기화 파일 어딘가에 await  
app.register(helmet)
```

경고 경고 [apollo-server-fastify](#) 및 [@fastify/helmet](#)을 사용하는 경우 GraphQL 플레이그라운드에서 [CSP](#)에 문제가 있을 수 있으며, 이 충돌을 해결하려면 아래와 같이 CSP를 구성하세요:

```
await app.register(fastifyHelmet, {  
  contentSecurityPolicy: {  
    지시어를 사용합니다: {  
      defaultSrc: [`'self'`, 'unpkg.com'],  
      styleSrc: [  
        `'self'`,  
        `unsafe-inline`,  
        'cdn.jsdelivr.net',  
        'fonts.googleapis.com',  
        'unpkg.com',  
      ],  
      fontSrc: [`'self'`, 'fonts.gstatic.com', 'data:'],  
      imgSrc: [`'self'`, 'data:', 'cdn.jsdelivr.net'],  
      scriptSrc: [  
        `'self'`,  
        `https: unsafe-inline`,  
        'cdn.jsdelivr.net',  
        'unsafe-val',  
      ],  
    },  
  },  
});  
  
// CSP를 전혀 사용하지 않을 경우 다음과 같이 사용할 수 있습니다: await  
app.register(fastifyHelmet, {  
  contentSecurityPolicy: false,  
});
```

CORS

CORS(교차 출처 리소스 공유)는 다른 도메인에서 리소스를 요청할 수 있는 메커니즘입니다. Nest는 내부적으로 Express cors 패키지를 사용합니다. 이 패키지는 요구 사항에 따라 사용자 정의할 수 있는 다양한 옵션을 제공합니다.

시작하기

CORS를 활성화하려면 Nest 애플리케이션 객체에서 `enableCors()` 메서드를 호출합니다.

```
const app = await NestFactory.create(AppModule);
app.enableCors();
await app.listen(3000);
```

`enableCors()` 메서드는 선택적 구성 객체 인수를 받습니다. 이 객체의 사용 가능한 속성은 공식 [CORS](#) 문서에 설명되어 있습니다. 또 다른 방법은 요청에 따라 비동기적으로(즉석에서) 구성 객체를 정의할 수 있는 [콜백 함수](#)를 전달하는 것입니다.

또는 `create()` 메서드의 옵션 객체를 통해 CORS를 활성화합니다. 기본 설정으로 CORS를 활성화하려면 `cors` 속성을 `true`로 설정합니다. 또는 [CORS 구성 객체](#) 또는 [콜백 함수](#)를 `cors` 속성 값으로 전달하여 동작을 사용자 지정할 수 있습니다.

```
const app = await NestFactory.create(AppModule, { cors: true });
await app.listen(3000);
```

CSRF 보호

크로스 사이트 요청 위조(CSRF 또는 XSRF라고도 함)는 웹 애플리케이션이 신뢰하는 사용자로부터 무단 명령이 전송되는 웹사이트의 악의적인 익스플로잇 유형입니다. 이러한 종류의 공격을 완화하기 위해 [csurf 패키지를](#) 사용할 수 있습니다.

Express와 함께 사용(기본값)

필요한 패키지를 설치하는 것으로 시작하세요:

```
$ npm i --save csurf
```

경고 경고 이 패키지는 더 이상 사용되지 않습니다. 자세한 내용은 [csurf 문서를](#) 참조하세요.

경고 경고 [csurf 문서에](#) 설명된 대로 이 미들웨어를 사용하려면 세션 미들웨어 또는 [쿠키 파서를](#) 먼저 초기화해야 합니다. 자세한 지침은 해당 문서를 참조하세요.

설치가 완료되면 [csurf](#) 미들웨어를 글로벌 미들웨어로 적용합니다.

```
'csurf'에서 *를 csurf로 가져옵니다;  
// ...  
// 초기화 파일 어딘가에 app.use(csurf());
```

Fastify와 함께 사용

필요한 패키지를 설치하는 것으로 시작하세요:

```
npm i --save @fastify/csrf-protection
```

설치가 완료되면 다음과 같이 [@fastify/csrf-protection](#) 플러그인을 등록합니다:

```
'@fastify/csrf-protection'에서 fastifyCsrf를 가져옵니다;  
// ...  
// 스토리지 플러그인을 등록한 후 초기화 파일 어딘가에 있습니다.  
경고 경고 여기 @fastify/csrf-protection 문서에 설명된 대로 이 플러그인을 사용하려면 먼저 스토리지 플러그인을 초기화해야 합니다. 자세한 지침은 해당 문서를 참조하세요.
```


속도 제한

무차별 대입 공격으로부터 애플리케이션을 보호하는 일반적인 기술은 속도 제한입니다. 시작하려면 [@nestjs/throttler](#) 패키지를 설치해야 합니다.

```
$ npm i --save @nestjs/throttler
```

설치가 완료되면, [스로틀러모듈](#)은 다른 네스트 패키지와 마찬가지로 `forRoot` 또는 `forRootAsync` 메서드를 사용하여 구성할 수 있습니다.

```
@@파일명(app.module) @Module({
  imports: [
    ThrottlerModule.forRoot({
      ttl: 60,
      limit: 10,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

위와 같이 보호되는 애플리케이션의 경로에 대한 `TTL`, 유효 시간 및 TTL 내의 최대 요청 수인 제한에 대한 전역 옵션을 설정합니다.

모듈을 가져온 다음에는 [스로틀러가드](#)를 바인딩할 방법을 선택할 수 있습니다. [가드](#) 섹션에서 언급한 바인딩 방식은 무엇이든 괜찮습니다. 예를 들어 가드를 전역적으로 바인딩하려면 이 공급자를 모든 모듈에 추가하여 바인딩 할 수 있습니다:

```
{
  provide: APP_GUARD,
  useClass: ThrottlerGuard
}
```

사용자 지정

가드를 컨트롤러에 바인딩하거나 전역적으로 바인딩하고 싶지만 하나 이상의 엔드포인트에 대해 속도 제한을 비활성화하려는 경우가 있을 수 있습니다. 이 경우 `@SkipThrottle()` 데코레이터를 사용하여 전체 클래스 또는

는 단일 경로에 대한 스토클러를 무효화할 수 있습니다. 모든 경로가 아닌 컨트롤러의 대부분을 제외하려는 경우 `@SkipThrottle()` 데코레이터는 부울을 받을 수도 있습니다.

```
스킵스로틀() @Controller('users')
```

사용자 컨트롤러 클래스 {} 내보내기

이 `@SkipThrottle()` 데코레이터는 경로 또는 클래스를 건너뛰거나 건너뛰는 클래스에서 경로 건너뛰기를 무효화하는 데 사용할 수 있습니다.

```
스킵스로틀() @Controller('users')
사용자 컨트롤러 클래스 내보내기 {
    // 이 경로에 속도 제한이 적용됩니다. 스kipstrottle(거짓)
    dontSkip() {
        "속도 제한이 있는 사용자 작업 목록."을 반환합니다;
    }
    // 이 경로는 속도 제한을 건너뜁니다. doSkip() {
        "속도 제한 없이 작업하는 사용자를 나열합니다."를 반환합니다;
    }
}
```

글로벌 모듈에 설정된 `제한` 및 `ttl`을 재정의하여 더 엄격하거나 느슨한 보안 옵션을 제공하는 데 사용할 수 있는 `@Throttle()` 데코레이터도 있습니다. 이 데코레이터는 클래스나 함수에도 사용할 수 있습니다. 이 데코레이터의 인수는 `limit`, `ttl` 순서이므로 순서가 중요합니다. 다음과 같이 구성해야 합니다:

```
// 속도 제한 및 기간에 대한 기본 구성을 재정의합니다. 스토틀(3, 60)
@GetMapping()
findAll() {
    반환 "사용자 지정 속도 제한으로 작동하는 사용자 목록.";
}
```

프록시

애플리케이션이 프록시 서버 뒤에서 실행되는 경우 특정 HTTP 어댑터 옵션(`express` 및 `fastify`)에서 `신뢰 프록시` 옵션을 확인하고 활성화하세요. 이렇게 하면 `X-Forwarded-For` 헤더에서 원래 IP 주소를 가져올 수 있으며, `getTracker()` 메서드를 재정의하여 `req.ip`가 아닌 헤더에서 값을 가져올 수 있습니다. 다음 예제는 `express` 와 `fastify` 모두에서 작동합니다:

```
// 스로틀러 비하인드-프록시.guard.ts  
'@nestjs/throttler'에서 { ThrottlerGuard }를 가져오고,  
'@nestjs/common'에서 { Injectable }을 가져옵니다;  
  
@Injectable()  
export class ThrottlerBehindProxyGard extends ThrottlerGard {  
    protected getTracker(req: Record<string, any>): string {  
        return req.ips.length ? req.ips[0] : req.ip; // 필요에 따라 IP 추출을  
    }  
}
```

개별화합니다.

```

}

// app.controller.ts
'./throttler-behind-proxy.guard'에서 { ThrottlerBehindProxyGuard }를 가
져옵니다;

```

사용가드(스로틀러비하인드프록시가드)

정보 힌트 익스프레스에 대한 요청 객체의 API는 [여기에서](#), 패스트파이브에 대한 요청 객체의 API는 [여기에서](#) 찾을 수 있습니다.

웹 소켓

이 모듈은 웹소켓과 함께 작동할 수 있지만 일부 클래스 확장이 필요합니다. 이 모듈을 확장하려면 `ThrottlerGuard`를 생성하고 `핸들 요청` 메서드를 다음과 같이 재정의합니다:

```

@Injectable()
내보내기 클래스 WsThrottlerGuard extends ThrottlerGuard {
  async handleRequest(context: ExecutionContext, limit: number, ttl: number): Promise<boolean> {
    const client = context.switchToWs().getClient();
    const ip = client._socket.remoteAddress
    const key = this.generateKey(context, ip);
    const { totalHits } = await this.storageService.increment(key, ttl);

    if (totalHits > limit) {
      새로운 ThrottlerException()을 던집니다;
    }

    참을 반환합니다;
  }
}

```

정보 힌트 `ws`를 사용하는 경우 `_socket`을 `conn`로 바꿔야 합니다.

웹소켓으로 작업할 때 몇 가지 유의해야 할 사항이 있습니다:

- 가드는 `앱_가드` 또는 `앱.사용글로벌가드()`에 등록할 수 없습니다.
- 한계에 도달하면 Nest는 `예외` 이벤트를 발생시키므로 이에 대비한 리스너가 있는지 확인하세요.

정보 힌트 `@nestjs/platform-ws` 패키지를 사용하는 경우 다음을 사용할 수 있습니다.

대신 `client._socket.remoteAddress`를 입력하세요.

GraphQL

스로틀러 가드는 GraphQL 요청을 처리하는 데에도 사용할 수 있습니다. 다시 말하지만, 가드를 확장할 수 있지 만 이번에는 `getHttpRequestResponse` 메서드가 재정의됩니다.

```

@Injectable()
export class GqlThrottlerGuard extends ThrottlerGuard {
  getRequestResponse(context: ExecutionContext) {
    const gqlCtx = GqlExecutionContext.create(context);
    const ctx = gqlCtx.getContext();
    반환 { req: ctx.req, res: ctx.res };
  }
}

```

구성

다음 옵션은 [스로틀러 모듈](#)에 유효합니다:

ttl 각 요청이 스토리지에서 지속되는 시간(초)

limit TTL 한도 내 최대 요청 횟수

에 관해서는 무시할 사용자 에이전트의 정규식 배열입니다.

무시 사용자 에이전트 스로틀링 요청

저장소 요청을 추적하는 방법에 대한 저장소 설정

비동기 구성

속도 제한 구성을 동기식이 아닌 비동기식으로 가져오고 싶을 수도 있습니다. 의존성 주입 및 [비동기](#) 메서드를 허용하는 [forRootAsync\(\)](#) 메서드를 사용할 수 있습니다.

한 가지 접근 방식은 팩토리 함수를 사용하는 것입니다:

```

모듈({ import:
  [
    ThrottlerModule.forRootAsync({
      import: [ConfigModule],
      inject: [ConfigService],
      useFactory: (config: ConfigService) =>
        ({ ttl: config.get('THRITTLE_TTL'),
          limit: config.get('THRITTLE_LIMIT'),
        }),
    }),
  ],
})
내보내기 클래스 AppModule {}

```

useClass 구문을 사용할 수도 있습니다:

```
모듈({ import:  
[  
    ThrottlerModule.forRootAsync({
```

```
    임포트합니다: [ConfigModule],  
    useClass: 스로틀러컨피그서비스,  
  },  
],  
}  
  
내보내기 클래스 AppModule {}
```

ThrottlerConfigService가 인터페이스를 구현하는 한 이 작업을 수행할 수 있습니다.

스로틀러옵션팩토리. 저장소

내장 스토리지는 메모리 내 캐시로, 요청이 특정 기준을 통과할 때까지 요청을 추적합니다.

글로벌 옵션으로 설정한 TTL. [저장소](#) 옵션에 고유한 저장소 옵션을 추가할 수 있습니다.

[ThrottlerStorage](#) 인터페이스를 구현하는 클래스만 있으면 됩니다.

분산 서버의 경우 [Redis](#)용 커뮤니티 스토리지 공급자를 사용하여 단일 데이터 소스를 확보할 수 있습니다.

정보 참고 ThrottlerStorage는 [@nestjs/throttler](#)에서 가져올 수 있습니다.

게이트웨이

종속성 주입, 데코레이터, 예외 필터, 파이프, 가드, 인터셉터 등 이 문서의 다른 곳에서 설명하는 대부분의 개념은 게이트웨이에도 동일하게 적용됩니다. Nest는 가능한 경우 구현 세부 사항을 추상화하여 동일한 구성 요소가 HTTP 기반 플랫폼, 웹 소켓 및 마이크로서비스에서 실행될 수 있도록 합니다. 이 섹션에서는 웹소켓에 특화된 Nest의 측면을 다룹니다.

Nest에서 게이트웨이는 [@WebSocketGateway\(\)](#) 데코레이터로 주석을 단 클래스일 뿐입니다. 기술적으로 게이트웨이는 플랫폼에 구애받지 않으므로 어댑터가 생성되면 모든 WebSockets 라이브러리와 호환됩니다. 기본적으로 지원되는 WS 플랫폼은 [socket.io](#)와 [ws](#)의 두 가지입니다. 필요에 가장 적합한 것을 선택할 수 있습니다. 또한 이 [가이드](#)에 따라 직접 어댑터를 만들 수도 있습니다.



정보 힌트 게이트웨이는 [프로바이더로](#) 취급될 수 있으며, 이는 클래스 생성자를 통해 종속성을 주입할 수 있음을 의미합니다. 또한 게이트웨이는 다른 클래스(프로바이더 및 컨트롤러)에서도 주입할 수 있습니다.

설치

웹소켓 기반 애플리케이션 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
@@파일명()
npm i --save @nestjs/웹소켓 @nestjs/플랫폼-소켓.io @@switch
npm i --save @nestjs/웹소켓 @nestjs/플랫폼-소켓.io
```

개요

일반적으로 앱이 웹 애플리케이션이 아니거나 포트를 수동으로 변경하지 않는 한 각 게이트웨이는 HTTP 서버와 동일한 포트에서 수신 대기합니다. 이 기본 동작은 [80이](#) 선택한 포트 번호인 [@WebSocketGateway\(80\)](#) 데코레이터에 인수를 전달하여 수정할 수 있습니다. 다음 구성을 사용하여 게이트웨이에서 사용하는 [네임스페이스](#)를 설정할 수도 있습니다:

[웹소켓게이트웨이\(80, { 네임스페이스: '이벤트' }\)](#)

경고 경고 게이트웨이는 기존 모듈의 공급자 배열에 참조될 때까지 인스턴스화되지 않습니다.

지원되는 모든 옵션을 소켓 생성자에 두 번째 인자로 전달할 수 있습니다.

WebSocketGateway() 데코레이터를 추가합니다:

```
웹소켓게이트웨이(81, { 트랜스포트: ['웹소켓'] })
```

이제 게이트웨이가 수신 대기 중이지만 아직 수신 메시지를 구독하지 않았습니다. [이벤트](#) 메시지를 구독하고 정확히 동일한 데이터로 사용자에게 응답하는 핸들러를 만들어 보겠습니다.

```
@@파일명(events.gateway) @SubscribeMessage('events')

handleEvent(@MessageBody() 데이터: 문자열): 문자열 {
    데이터를 반환합니다;
}

@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
    데이터를 반환합니다;
}
```

정보 힌트 `@SubscribeMessage()` 및 `@MessageBody()` 데코레이터는 다음에서 가져옵니다.
`nestjs/websockets` 패키지.

게이트웨이가 생성되면 모듈에 등록할 수 있습니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./events.gateway'에서 { EventsGateway }를 가져옵니다;

@@파일명(events.module)
@Module({
    공급자: [이벤트 게이트웨이]
})
이벤트 모듈 클래스 {} 내보내기
```

또한 데코레이터에 속성 키를 전달하여 수신 메시지 본문에서 속성 키를 추출할 수도 있습니다:

```
@@파일명(events.gateway) @SubscribeMessage('events')

handleEvent(@MessageBody('id') id: number): number {
    // id === messageBody.id
    반환 id;
}

@@switch
@Bind(MessageBody('id'))
@SubscribeMessage('events')
handleEvent(id) {
    // id === messageBody.id
    반환 id;
}
```

데코레이터를 사용하지 않으려면 다음 코드가 기능적으로 동일합니다:

```

@@파일명(events.gateway) @SubscribeMessage('events')
handleEvent(client: Socket, data: 문자열): 문자열 { 반
    환 데이터;
}
@@switch
@SubscribeMessage('events')
handleEvent(client, data) {
    데이터를 반환합니다;
}

```

위의 예에서 `handleEvent()` 함수는 두 개의 인수를 받습니다. 첫 번째 인수는 플랫폼별 **소켓** 인스턴스이고, 두 번째 인수는 클라이언트로부터 받은 데이터입니다. 하지만 이 접근 방식은 각 단위 테스트에서 **소켓** 인스턴스를 모킹해야 하므로 권장되지 않습니다.

이벤트 메시지가 수신되면 핸들러는 네트워크를 통해 전송된 것과 동일한 데이터가 포함된 확인을 보냅니다. 또한 클라이언트의 `emit()` 메서드를 사용하는 등 라이브러리별 접근 방식을 사용하여 메시지를 전송할 수도 있습니다. 연결된 소켓 인스턴스에 액세스하려면 `@ConnectedSocket()` 데코레이터를 사용합니다.

```

@@파일명(events.gateway)
@SubscribeMessage('events')
handleEvent(
    메시지 본문() 데이터: 문자열, @커넥티드 소켓() 클
    라이언트: Socket,
): 문자열 { 데이터를
    반환합니다;
}
@@switch
바인드(메시지바디(), 커넥티드소켓()) 구독 메시지('이
벤트') 핸들 이벤트(데이터, 클라이언트) {
    데이터를 반환합니다;
}

```

정보 힌트 `@ConnectedSocket()` 데코레이터는 `@nestjs/websockets` 패키지에서 가져옵니다.

하지만 이 경우 인터셉터를 활용할 수 없습니다. 사용자에게 응답하지 않으려면 **반환** 문을 건너뛰거나 명시적으로 "거짓" 값(예: 정의되지 않음)을 반환하면 됩니다.

이제 클라이언트가 다음과 같은 메시지를 전송합니다:

```
socket.emit('events', { name: 'Nest' });
```

`handleEvent()` 메서드가 실행됩니다. 위의 핸들러 내에서 전송된 메시지를 수신하려면 클라이언트는 해당 수신 확인 리스너를 첨부해야 합니다:

```
socket.emit('events', { name: 'Nest' }, (data) => console.log(data));
```

다중 응답

승인은 한 번만 발송됩니다. 또한 네이티브 웹소켓 구현에서는 지원되지 않습니다. 이 제한을 해결하기 위해 두 가지 프로퍼티로 구성된 객체를 반환할 수 있습니다. 방출된 이벤트의 이름인 이벤트와 클라이언트에 전달해야 하는 데이터입니다.

```
@@파일명(events.gateway) @SubscribeMessage('events')

handleEvent(@MessageBody() 데이터: 알 수 없음): WsResponse<unknown> {
    const event = 'events';
    반환 { 이벤트, 데이터 };
}

@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
handleEvent(data) {
    const event = 'events';
    return { event, data };
}
```

정보 힌트 `WsResponse` 인터페이스는 `@nestjs/websockets` 패키지에서 가져옵니다.

경고 경고 `데이터` 필드가 일반 JavaScript 객체 응답을 무시하므로 `데이터` 필드가 `ClassSerializerInterceptor`에 의존하는 경우 `WsResponse`를 구현하는 클래스 인스턴스를 반환해야 합니다.

클라이언트는 수신 응답을 수신 대기하기 위해 다른 이벤트 리스너를 적용해야 합니다.

```
socket.on('events', (data) => console.log(data));
```

비동기 응답

메시지 핸들러는 동기식 또는 비동기식으로 응답할 수 있습니다. 따라서 `비동기` 메서드가 지원됩니다. 또한 메시지 핸들러는 `Observable`을 반환할 수 있으며, 이 경우 스트림이 완료될 때까지 결과값이 방출됩니다.

```
@@파일명(events.gateway) @SubscribeMessage('events')

onEvent(@MessageBody() 데이터: 알 수 없음): Observable<WsResponse<number>> {
    const event = 'events';
    const response = [1, 2, 3];

    return from(response).pipe(
        map(data => ({ event, data })),
    );
}
```

```

    );
}

@@switch
@Bind(MessageBody())
@SubscribeMessage('events')
onEvent(data) {
  const event = 'events';
  const response = [1, 2, 3];

  return from(response).pipe(
    map(data => ({ event, data })),
  );
}

```

위의 예에서 메시지 처리기는 배열의 각 항목에 대해 3번 응답합니다. 라이프사이클 툭

유용한 라이프사이클 후크는 3가지가 있습니다. 모두 해당 인터페이스가 있으며 다음 문서에 설명되어 있습니다.

다음 표를 참조하세요:

OnGatewayInit	<code>afterInit()</code> 메서드를 강제로 구현합니다. 라이브러리별 서버 인스턴스 를 인수로 받습니다(필요한 경우 나머지는 스프레드합니다).
온게이트웨이 연결	<code>handleConnection()</code> 메서드를 구현하도록 강제합니다. 라이브러리별 클라이언트 소켓 인
OnGatewayDisconnect	스턴스를 인자로 받습니다. <code>handleDisconnect()</code> 메서드를 구현하도록 강제합니다. 라이브러리별 클라이언트 소켓 인스턴스를 인자로 받습니다.

정보 힌트 각 라이프사이클 인터페이스는 [@nestjs/websockets](#) 패키지에서 노출됩니다.

서버

때로는 네이티브 플랫폼별 서버 인스턴스에 직접 액세스하고 싶을 때가 있습니다. 이 객체에 대한 참조는 `afterInit()` 메서드(`OnGatewayInit` 인터페이스)에 인수로 전달됩니다. 또 다른 옵션은 `@WebSocketServer()` 데코레이터를 사용하는 것입니다.

`@WebSocketServer()` 서버:

서버;

경고 `@WebSocketServer()` 데코레이터는 [@nestjs/websockets](#)에서 가져온 것입니다.

패키지입니다.

Nest는 서버 인스턴스를 사용할 준비가 되면 이 프로퍼티에 서버 인스턴스를 자동으로 할당합니다.

예시

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

예외 필터

HTTP 예외 필터 계층과 해당 웹 소켓 계층의 유일한 차이점은 `HttpException`을 던지는 대신 `WsException`을 사용해야 한다는 점입니다.

새로운 `WsException`('잘못된 자격 증명입니다.')을 던집니다;

정보 힌트 `WsException` 클래스는 `@nestjs/websockets` 패키지에서 가져옵니다.

위의 샘플을 사용하면 Nest는 던져진 예외를 처리하고 다음과 같은 구조의 예외 메시지를 내보냅니다:

```
{  
  상태: '오류',  
  메시지가 표시됩니다: '잘못된 자격 증명입니다.'  
}
```

필터

웹 소켓 예외 필터는 HTTP 예외 필터와 동일하게 작동합니다. 다음 예제에서는 수동으로 인스턴스화된 메서드 범위 필터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 필터를 사용할 수도 있습니다(즉, 게이트웨이 클래스 앞에 `@UseFilters()` 데코레이터를 붙이면 됩니다).

```
사용필터(새로운 WsExceptionFilter()) 구독 메시지('이벤트')  
onEvent(클라이언트, 데이터: any): WsResponse<any>  
{ const event = 'events';  
  반환 { 이벤트, 데이터 };  
}
```

상속

일반적으로 애플리케이션 요구 사항을 충족하도록 완전히 사용자 정의된 예외 필터를 만듭니다. 그러나 핵심 예외 필터를 단순히 확장하고 특정 요인에 따라 동작을 재정의하려는 사용 사례가 있을 수 있습니다.

예외 처리를 기본 필터에 위임하려면 `BaseWsExceptionFilter`를 확장해야 합니다.

를 생성하고 상속된 `catch()` 메서드를 호출합니다.

@@파일명()

```
'@nestjs/common'에서 { Catch, ArgumentsHost }를 가져오고,  
'@nestjs/websockets'에서 { BaseWsExceptionFilter }를 가져옵니다  
;
```

```
@Catch()
export class AllExceptionsFilter extends BaseWsExceptionFilter {
  catch(exception: unknown, host: ArgumentsHost) {
    super.catch(예외, 호스트);
  }
}

@@switch
'@nestjs/common'에서 { Catch }를 가져옵니다;
'@nestjs/websockets'에서 { BaseWsExceptionFilter }를 가져옵니다;

@Catch()
export class AllExceptionsFilter extends BaseWsExceptionFilter {
  catch(exception, host) {
    super.catch(예외, 호스트);
  }
}
```

위의 구현은 접근 방식을 보여주는 셀일 뿐입니다. 확장 예외 필터의 구현에는 맞춤형 비즈니스 로직(예: 다양한 조건 처리)이 포함될 수 있습니다.

파이프

일반 파이프와 웹 소켓 파이프 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException`을 던지는 대신 `WsException`을 사용해야 한다는 것입니다. 또한 모든 파이프는 데이터 매개변수에만 적용됩니다(클라이언트 인스턴스의 유효성을 검사하거나 변환하는 것은 쓸모가 없으므로).

정보 힌트 `WsException` 클래스는 `@nestjs/websockets` 패키지에서 노출됩니다.

바인딩 파이프

다음 예제는 수동으로 인스턴스화된 메서드 범위 파이프를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 파이프를 사용할 수도 있습니다(즉, 게이트웨이 클래스에 `@UsePipes()` 데코레이터를 접두사로 붙이면 됩니다).

`@@파일명()`

사용 파이프(새로운 유효성 검사 파이프()) 구독 메시지('이벤트')

핸들 이벤트(클라이언트: 클라이언트, 데이터: 알 수 없음): `WsResponse<unknown>`

```
{ const event = 'events';
  반환 { 이벤트, 데이터 };
}
```

`@@switch`

사용파이프(새로운 유효성 검사 파이프()) 구독 메시지('

이벤트') 핸들 이벤트(클라이언트, 데이터) {

```
const event = 'events';
return { event, data };
}
```

경비병

웹 소켓 가드와 일반 HTTP 애플리케이션 가드 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException`을 던지는 대신 `WsException`을 사용해야 한다는 점입니다.

정보 힌트 `WsException` 클래스는 `@nestjs/websockets` 패키지에서 노출됩니다.

바인딩 가드

다음 예제는 메서드 범위 가드를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 가드를 사용할 수도 있습니다(즉, 게이트웨이 클래스 앞에 `@UseGuards()` 데코레이터를 붙이면 됩니다).

파일명() `@사용가드()` 인증가드()

`@가입메시지('이벤트')`

핸들 이벤트(클라이언트: 클라이언트, 데이터: 알 수 없음): `WsResponse<unknown>`

```
{ const event = 'events';
  반환 { 이벤트, 데이터 };
}
```

스위치 `@사용가드(AuthGuard)`

`@SubscribeMessage('events')`

```
핸들이벤트(클라이언트, 데이터) {
  const event = 'events';
  return { event, data };
}
```

인터셉터

일반 인터셉터와 웹 소켓 인터셉터 사이에는 차이가 없습니다. 다음 예제에서는 수동으로 인스턴스화된 메서드 범위 인터셉터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 게이트웨이 범위 인터셉터도 사용할 수 있습니다(즉, 게이트웨이 클래스 앞에 `@UseInterceptors()` 데코레이터).

```
@@파일명()
```

```
사용 인터셉터(새로운 트랜스폼 인터셉터()) 구독 메시지('이벤트')
```

```
핸들 이벤트(클라이언트: 클라이언트, 데이터: 알 수 없음): WsResponse<unknown>
```

```
{ const event = 'events';
```

```
반환 { 이벤트, 데이터 };
```

```
}
```

```
@@switch
```

```
사용 인터셉터(새로운 트랜스폼인터셉터())
```

```
@SubscribeMessage('events')
```

```
handleEvent(client, data) {
```

```
const event = 'events';
```

```
return { event, data };
```

```
}
```

어댑터

WebSockets 모듈은 플랫폼에 구애받지 않으므로 [WebSocketAdapter](#) 인터페이스를 사용하여 자체 라이브러리(또는 네이티브 구현)를 가져올 수 있습니다. 이 인터페이스는 다음 표에 설명된 몇 가지 메서드를 구현하도록 강제합니다:

create	전달된 인수를 기반으로 소켓 인스턴스를 생성합니다
bindClientConnect	클라이언트 연결 이벤트를 바인딩합니다. 클라이언트 연결 해제 이벤트를 바인딩합니다(선택 사항*).
바인드메시지 핸들러	수신 메시지를 해당 메시지 핸들러에 바인딩합니다.
close	서버 인스턴스 종료 <code>socket.io</code>

확장

`socket.io` 패키지는 [IoAdapter](#) 클래스로 래핑됩니다. 기본 기능을 향상시키려면 어떻게 해야 하나요? 기능이 필요한가요? 예를 들어, 기술 요구 사항에 따라 부하가 분산된 여러 웹 서비스 인스턴스에 걸쳐 이벤트를 브로드캐스트하는 기능이 필요합니다. 이를 위해 [IoAdapter](#)를 확장하고 새 `socket.io` 서버를 인스턴스화하는 단일 메서드를 재정의할 수 있습니다. 하지만 먼저 필요한 패키지를 설치해 보겠습니다.

경고 여러 로드 밸런싱 인스턴스에서 `socket.io`를 사용하려면 전송을 설정하여 폴링을 비활성화해야 합니다: `['websocket']`을 설정하여 폴링을 비활성화하거나 로드 밸런서에서 쿠키 기반 라우팅을 활성화해야 합니다. Redis만으로는 충분하지 않습니다. 자세한 내용은 [여기](#)를 참조하세요.

```
$ npm i --save redis socket.io @socket.io/redis-adapter
```

패키지가 설치되면 [RedisIoAdapter](#) 클래스를 생성할 수 있습니다.

```
'@nestjs/platform-socket.io'에서 { IoAdapter }를 임포트하고,  
'socket.io'에서 { ServerOptions }를 임포트합니다;  
'@socket.io/redis-adapter'에서 { createAdapter }를 가져오고,  
'redis'에서 { createClient }를 가져옵니다;
```

```
내보내기 클래스 RedisIoAdapter extends IoAdapter {  
    비공개 어댑터 생성자: 반환 유형<생성 어댑터 유형>;  
  
    비동기 connectToRedis(): Promise<void> {  
        const pubClient = createClient({ url: `redis://localhost:6379` });  
        const subClient = pubClient.duplicate();  
  
        await Promise.all([pubClient.connect(), subClient.connect()]);  
  
        this.adapterConstructor = createAdapter(pubClient, subClient);  
    }  
}
```

```
createI0Server(port: number, options?: ServerOptions): any {
  const server = super.createI0Server(port, options);
  server.adapter(this.adapterConstructor);
  서버를 반환합니다;
}
```

그런 다음 새로 생성한 Redis 어댑터로 전환하기만 하면 됩니다.

```
const app = await NestFactory.create(AppModule);
const redisIoAdapter = new RedisIoAdapter(app);
await redisIoAdapter.connectToRedis();

app.useWebSocketAdapter(redisIoAdapter);
```

Ws 라이브러리

사용 가능한 또 다른 어댑터는 프레임워크 사이에서 프록시처럼 작동하고 매우 빠르고 철저하게 테스트된 ws 라이브러리를 통합하는 [WsAdapter](#)입니다. 이 어댑터는 네이티브 브라우저 웹소켓과 완벽하게 호환되며 socket.io 패키지보다 훨씬 빠릅니다. 안타깝게도 바로 사용할 수 있는 기능이 훨씬 적습니다. 하지만 경우에 따라서는 꼭 필요하지 않을 수도 있습니다.

정보 힌트 ws 라이브러리는 네임스페이스([socket.io](#)에서 널리 사용되는 통신 채널)를 지원하지 않습니다. 그러나 어떻게든 이 기능을 모방하기 위해 서로 다른 경로에 여러 ws 서버를 마운트할 수 있습니다(

예시: [WebSocketGateway](#)({{ '{' }} 경로: '/users' {{ '}' }})).

ws를 사용하려면 먼저 필요한 패키지를 설치해야 합니다:

```
npm i --save @nestjs/platform-ws
```

패키지가 설치되면 어댑터를 전환할 수 있습니다:

```
const app = await NestFactory.create(AppModule);
app.useWebSocketAdapter(new WsAdapter(app));
```

정보 힌트 WsAdapter는 [@nestjs/platform-ws](#)에서 가져옵니다.

고급(사용자 지정 어댑터)

데모 목적으로 `ws` 라이브러리를 수동으로 통합해 보겠습니다. 앞서 언급했듯이 이 라이브러리에 대한 어댑터는 이미 생성되어 있으며 `@nestjs/platform-ws` 패키지에서 `WsAdapter` 클래스로 노출되어 있습니다. 다음은 단순화된 구현의 잠재적 모습입니다:

@@파일명 (ws-adapter)

'ws'에서 WebSocket으로 *를 가져옵니다;
'@nestjs/common'에서 { WebSocketAdapter, INestApplicationContext }를
가져옵니다;

'@nestjs/websockets'에서 { MessageMappingProperties }를 가져오고,
'rxjs'에서 { Observable, fromEvent, EMPTY }를 가져옵니다;
'rxjs/operators'에서 { mergeMap, filter }를 가져옵니다;

```
export class WsAdapter implements WebSocketAdapter {
  constructor(private app: INestApplicationContext) {}

  create(port: number, options: any = {}): any {
    return new WebSocket.Server({ port, ...options });
  }
}
```

```
bindClientConnect(server, callback: Function) {
  server.on('connection', callback);
}
```

바인드메시지핸들러(클라이언트:

```
  WebSocket,
  핸들러를 사용합니다: MessageMappingProperties[],
  프로세스: (데이터: any) => Observable<any>,
) {
  fromEvent(client, 'message')
    .pipe(
      mergeMap(data => this.bindMessageHandler(데이터, 핸들러, 프로세
스)),
      filter(결과 => 결과),
    )
    .subscribe(응답 => client.send(JSON.stringify(응답)));
}
```

```
bindMessageHandler(
  buffer,
  핸들러를 사용합니다: MessageMappingProperties[],
  프로세스: (데이터: any) => Observable<any>,
): 관찰가능<any> {
  const message = JSON.parse(buffer.data);
  const messageHandler = handlers.find(
    핸들러 => 핸들러.메시지 === 메시지.이벤트,
  );
  if (!messageHandler) {
    return EMPTY;
  }
}
```

```
반환 프로세스(메시지핸들러.콜백(메시지.데이터));  
}  
  
close(server) {  
    server.close();  
}  
}
```

정보 힌트 `ws` 라이브러리를 활용하려면 자체 라이브러리를 만드는 대신 기본 제공되는 `WsAdapter`를 사용하세요.

그런 다음 `useWebSocketAdapter()` 메서드를 사용하여 사용자 정의 어댑터를 설정할 수 있습니다:

`@@파일명`(메인)

```
const app = await NestFactory.create(AppModule);
app.useWebSocketAdapter(new WsAdapter(app));
```

예

`WsAdapter`를 사용하는 작업 예제는 [여기에서 확인할 수 있습니다.](#)

개요

Nest는 전통적인(모놀리식이라고도 하는) 애플리케이션 아키텍처 외에도 마이크로서비스 아키텍처 개발 스타일을 기본적으로 지원합니다. 종속성 주입, 데코레이터, 예외 필터, 파이프, 가드, 인터셉터 등 이 문서의 다른 곳에서 설명하는 대부분의 개념은 마이크로서비스에도 동일하게 적용됩니다. 가능한 경우 Nest는 구현 세부 사항을 추상화하여 동일한 구성 요소가 HTTP 기반 플랫폼, WebSocket 및 마이크로서비스에서 실행될 수 있도록 합니다. 이 섹션에서는 마이크로서비스에 특화된 Nest의 측면을 다룹니다.

Nest에서 마이크로서비스는 기본적으로 HTTP와는 다른 전송 계층을 사용하는 애플리케이션입니다.



Nest는 서로 다른 마이크로서비스 인스턴스 간의 메시지 전송을 담당하는 전송자라고 하는 여러 가지 기본 제공 전송 계층 구현을 지원합니다. 대부분의 전송기는 기본적으로 요청-응답 및 이벤트 기반 메시지 스타일을 모두 지원합니다. Nest는 요청-응답 및 이벤트 기반 메시징 모두에 대한 표준 인터페이스 뒤에 있는 각 전송기의 구현 세부 사항을 추상화합니다. 따라서 애플리케이션 코드에 영향을 주지 않고도 특정 전송 계층의 특정 안정성 또는 성능 기능을 활용하기 위해 한 전송 계층에서 다른 전송 계층으로 쉽게 전환할 수 있습니다.

설치

마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
npm i --save @nestjs/microservices
```

시작하기

마이크로서비스를 인스턴스화하려면 `NestFactory` 클래스의 `createMicroservice()` 메서드를 사용합니다:

@@파일명(메인)

'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/microservices'에서 { Transport, MicroserviceOptions }를 임포트하고,
'./app.module'에서 { AppModule }을 임포트합니다;

비동기 함수 부트스트랩()

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(  
  AppModule,  
  {  
    전송: Transport.TCP,  
  },  
);  
await app.listen();  
}
```

부트스트랩(); @@스위

치

'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/microservices'에서 { Transport }를 가져옵니다;

```
'./app.module'에서 { AppModule }을 가져옵니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.TCP,
  });
  await app.listen();
}

부트스트랩();
```

정보 힌트 마이크로서비스는 기본적으로 TCP 전송 계층을 사용합니다.

`createMicroservice()` 메서드의 두 번째 인수는 옵션 객체입니다. 이 객체는 두 개의 멤버로 구성될 수 있습니다:

전송자 전송자를 지정합니다(예: `Transport.NATS`).

옵션 트랜스포터 동작을 결정하는 트랜스포터별 옵션 객체입니다.

옵션 객체는 선택한 전송자에 따라 다릅니다. TCP 전송자는 아래에 설명된 속성을 노출합니다. 다른 전송자(예: Redis, MQTT 등)의 경우 사용 가능한 옵션에 대한 설명은 관련 챕터를 참조하세요.

호스트 연결 호스트 이름

포트 연결 포트

재시도 시도 메시지 재시도 횟수(기본값: 0) **재시도 지연** 메시지 재시도 시도

~~사이의 자연 시간(ms)(기본값: 0)~~

패턴

마이크로서비스는 패턴으로 메시지와 이벤트를 모두 인식합니다. 패턴은 리터럴 객체나 문자열과 같은 일반 값입니다. 패턴은 자동으로 직렬화되어 메시지의 데이터 부분과 함께 네트워크를 통해 전송됩니다. 이러한 방식으로 메시지 발신자와 수신자는 어떤 요청이 어떤 핸들러에 의해 소비되는지 조정할 수 있습니다.

요청-응답

요청-응답 메시지 스타일은 다양한 외부 서비스 간에 메시지를 교환해야 할 때 유용합니다. 이 패러다임을 사용하면

메시지 확인 프로토콜을 수동으로 구현할 필요 없이 서비스가 실제로 메시지를 수신했는지 확인할 수 있습니다. 하지만 요청-응답 패러다임이 항상 최선의 선택은 아닙니다. 예를 들어, 로그 기반 지속성을 사용하는 스트리밍 전송자(예: [카프카](#) 또는 [NATS 스트리밍](#))는 이벤트 메시징 패러다임에 더 적합한 다른 범위의 문제를 해결하는 데 최적화되어 있습니다(자세한 내용은 아래 [이벤트 기반 메시징](#) 참조).

요청-응답 메시지 유형을 활성화하기 위해 Nest는 두 개의 논리 채널을 생성합니다. 한 채널은 데이터 전송을 담당하고 다른 채널은 수신 응답을 대기합니다. [NATS와](#) 같은 일부 기본 전송의 경우 이 이중 채널 지원이 기본으로 제공됩니다. 다른 전송의 경우 Nest는 수동으로 보정합니다.

별도의 채널을 만들어야 합니다. 이 경우 오버헤드가 발생할 수 있으므로 요청-응답 메시지 스타일이 필요하지 않은 경우에는 이벤트 기반 방법을 사용하는 것이 좋습니다.

요청-응답 패러다임에 기반한 메시지 핸들러를 만들려면 `@nestjs/microservices` 패키지에서 가져온 `@MessagePattern()` 데코레이터를 사용합니다. 이 데코레이터는 `컨트롤러` 클래스가 애플리케이션의 진입점이므로 `컨트롤러` 클래스 내에서만 사용해야 합니다. 프로바이더 내부에서 사용하면 Nest 런타임에서 무시되므로 아무런 효과가 없습니다.

`@@파일명`(math.controller)

```
'@nestjs/common'에서 { Controller }를 가져옵니다;
'@nestjs/microservices'에서 { MessagePattern }을 가져옵니다;
```

`컨트롤러()`

```
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  accumulate(data: number[]): number {
    반환 (데이터 || []).reduce((a, b) => a + b);
  }
}
@@switch
'@nestjs/common'에서 { Controller }를 가져옵니다;
'@nestjs/microservices'에서 { MessagePattern }을 가져옵니다;
```

`컨트롤러()`

```
export class MathController {
  @MessagePattern({ cmd: 'sum' })
  accumulate(data) {
    반환 (데이터 || []).reduce((a, b) => a + b);
  }
}
```

위 코드에서 `accumulate()` 메시지 핸들러는 `{} {{ }} cmd: 'sum' {{ }} {}` 메시지 패턴을 충족하는 메시지를 수신합니다. 메시지 핸들러는 클라이언트에서 전달된 데이터라는 단일 인수를 받습니다. 이 경우 데이터는 누적될 숫자 배열입니다.

비동기 응답

메시지 핸들러는 동기식 또는 비동기식으로 응답할 수 있습니다. 따라서 `비동기` 메서드가 지원됩니다.

```
@@파일명()  
메시지 패턴({ cmd: '합계' })  
async accumulate(data: number[]): Promise<number> {  
    return (data || []).reduce((a, b) => a + b);  
}  
@@switch  
메시지패턴({ cmd: '합계' }) async  
accumulate(data) {
```

```
    반환(데이터 || []).reduce((a, b) => a + b);
}
```

메시지 핸들러는 `Observable`을 반환할 수도 있으며, 이 경우 스트림이 완료될 때까지 결과값이 방출됩니다.

`@@파일명()`

메시지 패턴({ cmd: '합계' }) 누적(데이터: 숫자[]):

```
Observable<number> {
  return from([1, 2, 3]);
}
@@switch
메시지 패턴({ cmd: '합계' }) 누적(데이터: 숫자[]):
```

```
Observable<number> {
  return from([1, 2, 3]);
}
```

위의 예에서 메시지 핸들러는 배열의 각 항목에 대해 3번 응답합니다. 이벤트 기반

요청-응답 방식은 서비스 간에 메시지를 교환하는 데 이상적이지만 다음과 같은 경우에는 적합하지 않습니다.

메시지 스타일이 이벤트 기반인 경우 - 응답을 기다리지 않고 이벤트만 게시하려는 경우. 이 경우 두 채널을 유지하기 위해 요청-응답에 필요한 오버헤드를 원하지 않을 수 있습니다.

시스템의 이 부분에서 특정 조건이 발생했음을 다른 서비스에 간단히 알리고 싶다고 가정해 보겠습니다. 이것이 이벤트 기반 메시지 스타일의 이상적인 사용 사례입니다.

이벤트 핸들러를 생성하기 위해 `@EventPattern()` 데코레이터를 사용합니다.

`nestjs/microservices` 패키지.

`@@파일명()`

```
@EventPattern('user_created')
async handleUserCreated(data: Record<string, unknown>) {
  // 비즈니스 로직
}
@@switch
@EventPattern('user_created')
async handleUserCreated(data) {
  // 비즈니스 로직
}
```

정보 힌트 단일 이벤트 패턴에 대해 여러 이벤트 핸들러를 등록할 수 있으며 모든 이벤트 핸들러가 자동으로 병렬로 트리거됩니다.

`handleUserCreated()` 이벤트 핸들러는 '`user_created`' 이벤트를 수신 대기합니다. 이벤트 핸들러는 클라이언트에서 전달된 `데이터`(이 경우 네트워크를 통해 전송된 이벤트 페이로드)를 단일 인수로 받습니다.

데코레이터

보다 정교한 시나리오에서는 수신 요청에 대한 자세한 정보에 액세스하고 싶을 수도 있습니다. 예를 들어 와일드카드 구독이 있는 NATS의 경우 제작자가 메시지를 보낸 원래 제목을 얻고 싶을 수 있습니다. 마찬가지로 Kafka에서 메시지 헤더에 액세스하고 싶을 수 있습니다. 이를 위해 다음과 같이 기본 제공 데코레이터를 사용할 수 있습니다:

```
@@파일명() @메시지패턴('time.us.*')
getDate(@Payload() 데이터: 숫자[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // 예: "time.us.east"
  return new Date().toLocaleTimeString(...);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*')
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // 예: "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

정보 힌트 `@Payload()`, `@Ctx()` 및 `NatsContext`는 [@nestjs/microservices](#)에서 가져옵니다.

정보 힌트 `@Payload()` 데코레이터에 속성 키를 전달하여 들어오는 페이로드 객체에서 특정 속성(예: `@Payload('id')`)을 추출할 수도 있습니다.

클라이언트

클라이언트 Nest 애플리케이션은 `ClientProxy` 클래스를 사용하여 메시지를 교환하거나 Nest 마이크로서비스에 이벤트를 게시할 수 있습니다. 이 클래스는 원격 마이크로서비스와 통신할 수 있는 `send()`(요청-응답 메시징용) 및 `emit()`(이벤트 중심 메시징용)과 같은 여러 메서드를 정의합니다. 다음 방법 중 하나를 사용하여 이 클래스의 인스턴스를 가져옵니다.

한 가지 기법은 정적 `register()` 메서드를 노출하는 `ClientsModule`을 임포트하는 것입니다. 이 메서드

는 마이크로서비스 전송자를 나타내는 객체 배열인 인수를 받습니다. 이러한 각 객체에는 이름 속성, 선택적 전송 속성(기본값은 `Transport.TCP`) 및 선택적 트랜스포터별 옵션 속성이 있습니다.

`name` 속성은 필요한 경우 `ClientProxy` 인스턴스를 주입하는 데 사용할 수 있는 주입 토큰 역할을 합니다. 이름 속성의 값은 인젝션 토큰으로서 [여기](#)에 설명된 대로 임의의 문자열 또는 JavaScript 심볼일 수 있습니다.

옵션 프로퍼티는 `createMicroservice()`에서 본 것과 동일한 프로퍼티를 가진 객체입니다. 메서드를 사용합니다.

```
모듈({ import: [
  ClientsModule.register([
    { name: 'MATH_SERVICE', transport: Transport.TCP },
  ]),
]
...
})
```

모듈을 임포트한 후에는 위에 표시된 'MATH_SERVICE' 트랜스포터 옵션을 통해 지정된 대로 구성된 ClientProxy 인스턴스를 @Inject() 데코레이터를 사용하여 주입할 수 있습니다.

```
생성자(
  @Inject('MATH_SERVICE') 비공개 클라이언트: ClientProxy,
) {}
```

정보 힌트 클라이언트 모듈과 클라이언트 프록시 클래스는 [nestjs/microservices](#) 패키지.

때로는 클라이언트 애플리케이션에서 트랜스포터 구성을 하드코딩하는 대신 다른 서비스(예: [컨피그서비스](#))에서 가져와야 할 수도 있습니다. 이를 위해 ClientProxyFactory 클래스를 사용하여 사용자 정의 프로바이더를 등록할 수 있습니다. 이 클래스에는 정적 create() 메서드가 있는데, 이 메서드는 트랜스포터 옵션 객체를 받아들이고 사용자 정의된 ClientProxy 인스턴스를 반환합니다.

```
모듈({ providers: [
  {
    제공: 'math_service',
    사용 팩토리: (config서비스: 구성 서비스) => {
      const mathSvcOptions = configService.getMathSvcOptions();
      return ClientProxyFactory.create(mathSvcOptions);
    },
    주입합니다: [구성 서비스],
  }
]
...
})
```

정보 힌트 ClientProxyFactory는 [@nestjs/microservices](#) 패키지에서 가져옵니다.

또 다른 옵션은 @Client() 속성 데코레이터를 사용하는 것입니다.

```
@Client({ 전송: Transport.TCP }) 클라이언
```

```
트: ClientProxy;
```

정보 힌트 `@Client()` 데코레이터는 `@nestjs/microservices` 패키지에서 가져옵니다.

클라이언트 인스턴스를 테스트하기 어렵고 공유하기 어렵기 때문에 `@Client()` 데코레이터를 사용하는 것은 선호되지 않습니다.

클라이언트 프록시는 게으르다. 즉시 연결을 시작하지 않습니다. 대신 첫 번째 마이크로서비스 호출 전에 설정된 다음 각 후속 호출에서 재사용됩니다. 그러나 연결이 설정될 때까지 애플리케이션 부트스트랩 프로세스를 지연시키려면 `OnApplicationBootstrap` 라이프사이클 후크 내에서 `ClientProxy` 객체의 `connect()` 메서드를 사용하여 수동으로 연결을 시작할 수 있습니다.

`@@파일명()`

```
async onApplicationBootstrap() {
    await this.client.connect();
}
```

연결을 만들 수 없는 경우 `connect()` 메서드는 해당 오류 객체와 함께 거부됩니다. 메시지 보내기

`ClientProxy`는 `send()` 메서드를 노출합니다. 이 메서드는 마이크로서비스를 호출하고 응저버블에 응답을 추가합니다. 따라서 방출된 값을 쉽게 구독할 수 있습니다.

`@@파일명()`

```
accumulate(): Observable<number> {
    const pattern = { cmd: 'sum' };
    const payload = [1, 2, 3];
    이.클라이언트.보내기<번호>(패턴, 페이로드)를 반환합니다;
}

@@switch
accumulate() {
    const pattern = { cmd: '합계' };
    const payload = [1, 2, 3];
    이.클라이언트.보내기(패턴, 페이로드)를 반환합니다;
}
```

`send()` 메서드는 패턴과 페이로드라는 두 개의 인수를 받습니다. 패턴은 `@MessagePattern()` 데코레이터에 정의된 패턴과 일치해야 합니다. 페이로드는 원격 마이크로서비스로 전송하려는 메시지입니다. 이 메서드는 콜드 응저버블을 반환하므로 메시지가 전송되기 전에 명시적으로 응저버블을 구독해야 합니다.

이벤트 게시

이벤트를 전송하려면 `ClientProxy` 객체의 `emit()` 메서드를 사용합니다. 이 메서드는 메시지 브로커에 이벤트를 게시합니다.

```

@@filename()
async publish()
{
  this.client.emit<number>('user_created', new UserCreatedEvent());
}
@@switch
async publish() {
  this.client.emit('user_created', new UserCreatedEvent());
}

```

`emit()` 메서드는 패턴과 페이로드라는 두 개의 인수를 받습니다. 패턴은 `@EventPattern()` 데코레이터에 정의된 패턴과 일치해야 합니다. 페이로드는 원격 마이크로서비스로 전송하려는 이벤트 페이로드입니다. 이 메서드는 `send()` 가 반환하는 콜드 옵저버블과 달리 핫 옵저버블을 반환하므로, 명시적으로 옵저버블을 구독하는지 여부와 관계없이 프록시는 즉시 이벤트를 전달하려고 시도합니다.

범위

다른 프로그래밍 언어 배경을 가진 사람들에게는 Nest에서 거의 모든 것이 들어오는 요청에서 공유된다는 사실이 의외로 느껴질 수 있습니다. 데이터베이스에 대한 연결 풀, 전역 상태를 가진 싱글톤 서비스 등이 있습니다. Node.js는 모든 요청이 별도의 스레드에서 처리되는 요청/응답 다중 스레드 상태 비저장 모델을 따르지 않는다는 점을 기억하세요. 따라서 싱글톤 인스턴스를 사용하는 것은 애플리케이션에 완전히 안전합니다.

그러나 GraphQL 애플리케이션의 요청별 캐싱, 요청 추적 또는 멀티테넌시와 같이 핸들러의 요청 기반 수명이 원하는 동작일 수 있는 예외적인 경우가 있습니다. [여기에서](#) 범위를 제어하는 방법을 알아보세요.

요청 범위가 지정된 핸들러와 프로바이더는 `@Inject()` 데코레이터를 `CONTEXT` 토큰과 함께 사용하여 `RequestContext`를 주입할 수 있습니다:

```

'@nestjs/common'에서 { Injectable, Scope, Inject }를 가져오고,
'@nestjs/microservices'에서 { CONTEXT, RequestContext }를 가져옵니다
;
```

`주입 가능({ 범위: Scope.REQUEST }) 내보내기`

```

클래스 CatsService {
  생성자(@Inject(CONTEXT) private ctx: RequestContext) {}
}
```

이렇게 하면 두 가지 프로퍼티가 있는 `RequestContext` 객체에 액세스할 수 있습니다:

```
export interface RequestContext<T = any> {  
    pattern: string | Record<string, any>;  
    data: T;  
}
```

데이터 속성은 메시지 제작자가 보낸 메시지 페이로드입니다. **패턴** 속성은 수신 메시지를 처리할 적절한 핸들러를 식별하는 데 사용되는 패턴입니다.

시간 초과 처리

분산 시스템에서는 때때로 마이크로서비스가 다운되거나 사용할 수 없는 경우가 있습니다. 무한히 오래 기다리지 않으려면 타임아웃을 사용할 수 있습니다. 타임아웃은 다른 서비스와 통신할 때 매우 유용한 패턴입니다. 마이크로서비스 호출에 타임아웃을 적용하려면 RxJS **타임아웃** 연산자를 사용하면 됩니다. 마이크로서비스가 특정 시간 내에 요청에 응답하지 않으면 예외가 발생하고, 이를 포착하여 적절히 처리할 수 있습니다.

이 문제를 해결하려면 rxjs 패키지를 사용해야 합니다. 파이프에서 **타임아웃** 연산자를 사용하기만 하면 됩니다:

```
@@파일명()
this.client
    .send<TResult, TInput>(패턴, 데이터)
    .pipe(timeout(5000));
@@switch
this.client
    .send(패턴, 데이터)
    .pipe(timeout(5000));
```

정보 힌트 **타임아웃** 연산자는 rxjs/operators 패키지에서 가져옵니다.

5초 후에도 마이크로서비스가 응답하지 않으면 오류가 발생합니다.

Redis

Redis 트랜스포터는 게시/구독 메시징 패러다임을 구현하고 Redis의 [게시/구독](#) 기능을 활용합니다. 게시된 메시지는 어떤 구독자(있는 경우)가 최종적으로 메시지를 수신할지 알 수 없이 채널로 분류됩니다. 각 마이크로서비스는 원하는 수의 채널에 구독할 수 있습니다. 또한 한 번에 하나 이상의 채널을 구독할 수 있습니다. 채널을 통해 교환되는 메시지는 다음과 같습니다.

즉, 메시지가 게시된 후 해당 메시지에 관심 있는 구독자가 없는 경우 메시지를 삭제합니다, 를 누르면 메시지가 삭제되어 복구할 수 없습니다. 따라서 다음과 같은 메시지가 삭제되지 않는다는 보장이 없습니다. 또는 이벤트가 하나 이상의 서비스에서 처리됩니다. 하나의 메시지를 여러 가입자가 구독하고 수신할 수 있습니다.



설치

Redis 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save ioredis
```

개요

Redis 트랜스포터를 사용하려면 `createMicroservice()` 메서드에 다음 옵션 객체를 전달합니다:

@@파일명(메인)

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>( AppModule, {
    transport: Transport.REDIS, 옵션:
    {
        호스트: 'localhost',
        포트: 6379,
    },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.REDIS,
    옵션: {
        호스트: 'localhost',
        포트: 6379,
    },
});
```

정부 힌트 `Transport` 열거형은 [@nestjs/microservices](#) 패키지에서 가져옵니다.

옵션

옵션 속성은 선택한 트랜스포터에 따라 다릅니다. Redis 트랜스포터는 아래에 설명된 속성을 노출합니다.

호스트	연결 URL
포트	연결 포트
재시도 시도	메시지 재시도 횟수(기본값: 0)
재시도 지연	메시지 재시도 시도
사이의 자연 시간(ms)(기본값: 0)	Redis 월카드 구독을 활성화하여 트랜스포터가 다음을 사용하도록 지시합니다.
와일드카드	구독/메시지 내부를 살펴봅니다. (기본값: false)

공식 [아이오레디스](#) 클라이언트에서 지원하는 모든 프로퍼티는 이 트랜스포터에서도 지원됩니다. 클

라이언트

다른 마이크로서비스 전송기와 마찬가지로, Redis `ClientProxy` 인스턴스를 생성하기 위한 [몇 가지 옵션이 있습니다](#).

인스턴스를 생성하는 한 가지 방법은 `ClientsModule`을 사용하는 것입니다. `ClientsModule`을 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 `register()` 메서드를 사용하여 위에서 `createMicroservice()` 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 이름 속성을 전달합니다. 클라이언트 모듈에 대한 자세한 내용은 [여기](#)를 참조하세요.

```
모듈({ import: [
  ClientsModule.register([
    {
      이름: 'MATH_SERVICE',
      transport: Transport.REDIS, 옵션
        : {
          호스트: 'localhost', 포
            트: 6379,
        }
      },
    ],
  ]),
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. Redis 트랜잭터 사용 시, `RedisContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RedisContext)
{
    console.log(`채널: ${context.getChannel()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
    console.log(`채널: ${context.getChannel()}`);
}
```

정보 힌트 `@Payload()`, `@Ctx()` 및 `RedisContext`는

`nestjs/microservices` 패키지.

MQTT

MQTT(메시지 큐 텔레메트리 전송)는 짧은 지연 시간에 최적화된 오픈 소스 경량 메시징 프로토콜입니다. 이 프로토콜은 게시/구독 모델을 사용하여 장치를 연결하는 확장 가능하고 비용 효율적인 방법을 제공합니다. MQTT를 기반으로 구축된 통신 시스템은 퍼블리싱 서버, 브로커 및 하나 이상의 클라이언트로 구성됩니다. 제한된 장치와 낮은 대역폭, 높은 지연 시간 또는 불안정한 네트워크를 위해 설계되었습니다.

설치

MQTT 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save mqtt
```

개요

MQTT 트랜스포터를 사용하려면 `createMicroservice()` 메서드에 다음 옵션 객체를 전달합니다:

```
@@파일명 (메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
    전송: Transport.MQTT, 옵션: {
        url: 'mqtt://localhost:1883',
    },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.MQTT,
    옵션: {
        url: 'mqtt://localhost:1883',
    },
});
```

정보 힌트 `Transport` 열거형은 [@nestjs/microservices](#) 패키지에서 가져옵니다.

옵션

옵션 객체는 선택한 트랜스포터에 따라 다릅니다. MQTT 트랜스포터는 [여기에](#) 설명된 속성을 노출합니다.

클라이언트

다른 마이크로서비스 전송기와 마찬가지로 MQTT `ClientProxy` 인스턴스를 생성하는 데는 여러 가지 옵션이 있습니다.

인스턴스를 생성하는 한 가지 방법은 `ClientsModule`을 사용하는 것입니다. `ClientsModule`을 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 `register()` 메서드를 사용하여 위에서 `createMicroservice()` 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 이름 속성을 전달합니다. 클라이언트 모듈에 대한 자세한 내용은 [여기](#)를 참조하세요.

```
모듈({ import: [
  ClientsModule.register([
    {
      이름: 'MATH_SERVICE',
      전송: Transport.MQTT, 옵션: {
        url: 'mqtt://localhost:1883',
      }
    },
  ]),
]
...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. MQTT 트랜스포터를 사용하는 경우 `MqttContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
  console.log(`Topic: ${context.getTopic()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(`Topic: ${context.getTopic()}`);
}
```

`@@파일명()` `@메시지패턴('알림')` 정보 힌트 `@Payload()`, `@Ctx()` 및 `MqttContext`는

원본 mqtt 패킷에 액세스하려면 다음과 같이 `MqttContext` 객체의 `getPacket()` 메서드를 사용합니다:

```
getNotifications(@Payload() data: number[], @Ctx() context: MqttContext) {
    console.log(context.getPacket());
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
    console.log(context.getPacket());
}
```

와일드카드

구독은 명시적인 주제에 대한 구독이거나 와일드카드를 포함할 수 있습니다. 와일드카드는 +와 # 두 가지를 사용할 수 있습니다.

+는 단일 레벨 와일드카드이고 #는 여러 주제 레벨을 포괄하는 다중 레벨 와일드카드입니다.

```
@@ 파일 이름 () @ 메시지 패턴('센서/+ / 온도/+')
getTemperature(@Ctx() context: MqttContext) {
    console.log(`Topic: ${context.getTopic()}`);
}
@@switch
@Bind(Ctx())
@MessagePattern('sensors/+temp/') getTemp(context)
{
    console.log(`Topic: ${context.getTopic()}`);
}
```

레코드 빌더

메시지 옵션을 구성하려면(QoS 수준 조정, Retain 또는 DUP 플래그 설정, 페이로드에 추가 속성 추가 등)

`MqttRecordBuilder` 클래스를 사용하면 됩니다. 예를 들어, QoS를 2로 설정하려면 다음과 같이 `setQoS` 메서드를 사용합니다:

```
const userProperties = { 'x-version': '1.0.0' };
const record = new MqttRecordBuilder(':cat:')
    .setProperties({ userProperties })
    .setQoS(1)
    .build();
client.send('replace-이모티콘', record).subscribe(...);
```

@@파일명() @메시지패턴('이모티콘 대체')

정보 힌트 `MqttRecordBuilder` 클래스는 [@nestjs/microservices](#) 패키지에서 내보내집니다.

또한 서버 측에서도 [MqttContext](#)에 액세스하여 이러한 옵션을 읽을 수 있습니다.

```

replaceEmoji(@Payload() 데이터: 문자열, @Ctx() 컨텍스트: MqttContext): 문자열
{
  const { properties: { userProperties } } = context.getPacket();
  return userProperties['x-version'] === '1.0.0' ? '□' : '☒';
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const { properties: { userProperties } } = context.getPacket();
  return userProperties['x-version'] === '1.0.0' ? '□' : '☒';
}

```

여러 요청에 대해 사용자 속성을 구성해야 하는 경우도 있는데, 이러한 옵션을 [ClientProxyFactory](#)에 전달할 수 있습니다.

```

'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/microservices'에서 { ClientProxyFactory, Transport }를 가져옵니다;

모듈({ providers:
  [
    {
      제공: 'API_v1', useFactory:
        () =>
          ClientProxyFactory.create({
            transport: Transport.MQTT,
            options: {
              url: 'mqtt://localhost:1833',
              userProperties: { 'x-version': '1.0.0' },
            },
          }),
    },
  ],
})
내보내기 클래스 ApiModule {}

```

NATS

NATS는 클라우드 네이티브 애플리케이션, IoT 메시징 및 마이크로서비스 아키텍처를 위한 간단하고 안전한 고성능 오픈 소스 메시징 시스템입니다. NATS 서버는 Go 프로그래밍 언어로 작성되었지만 서버와 상호 작용하는 클라이언트 라이브러리는 수십 개의 주요 프로그래밍 언어로 사용할 수 있습니다. NATS는 최대 한 번 전송과 최소 한 번 전송을 모두 지원합니다. 대규모 서버와 클라우드 인스턴스, 엣지 게이트웨이, 심지어 사물 인터넷 장치에 이르기까지 어디서나 실행할 수 있습니다.

설치

NATS 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save nats
```

개요

NATS 트랜스포터를 사용하려면 `createMicroservice()` 메서드에 다음 옵션 객체를 전달합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.NATS, 옵션:
  {
    서버: ['nats://localhost:4222'],
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.NATS,
  옵션: {
    서버: ['nats://localhost:4222'],
  },
});
```

정보 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

옵션 개체는 선택한 트랜스포터에 따라 다릅니다. NATS 트랜스포터는 [여기에](#) 설명된 속성을 노출합니다. 또한

서버가 구독해야 하는 큐의 이름을 지정할 수 있는 **큐 속성**이 있습니다(이 설정을 무시하려면 정의되지 않은 상태로 두세요). 아래에서 NATS 큐 그룹에 대해 자세히 알아보세요.

클라이언트

다른 마이크로서비스 전송기와 마찬가지로 NATS `ClientProxy` 인스턴스를 생성하는 데는 [몇 가지 옵션이 있습니다](#).

인스턴스를 생성하는 한 가지 방법은 `ClientsModule`을 사용하는 것입니다. `ClientsModule`을 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 `register()` 메서드를 사용하여 위에서 `createMicroservice()` 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 [이름](#) 속성을 전달합니다. 클라이언트 모듈에 대한 자세한 내용은 [여기](#)를 참조하세요.

```
모듈({ import: [
  ClientsModule.register([
    {
      이름: 'MATH_SERVICE',
      transport: Transport.NATS, 옵션:
      {
        서버: ['nats://localhost:4222'],
      }
    },
  ]),
]
...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

요청-응답

요청-응답 메시지 스타일([자세히 읽기](#))의 경우, NATS 전송자는 NATS 기본 제공 [요청-응답](#) 메커니즘을 사용하지 않습니다. 대신, "요청"은 `게시()` 함수를 사용하여 지정된 주제에 게시됩니다.

메서드에 고유한 회신 제목 이름을 지정하면 응답자는 해당 제목을 수신하고 회신 제목에 응답을 보냅니다. 회신 제목은 양쪽 당사자의 위치에 관계없이 요청자에게 동적으로 다시 전달됩니다.

이벤트 기반

이벤트 기반 메시지 스타일([자세히 읽기](#))의 경우 NATS 트랜스포터는 NATS에 내장된 [게시-구독](#) 메커니즘을 사용합니다. 게시자는 주제에 대한 메시지를 보내고 수신 중인 활성 구독자가 있는 경우를 입력하면 해당 주체가 메시지를 받습니다. 또한 구독자는 정규식처럼 작동하는 와일드카드 주제에 관심을 등록할 수도 있습니다. 이러한 일대다 패턴을 팬아웃이라고도 합니다.

대기열 그룹

NATS는 [분산 큐라는](#) 기본 제공 부하 분산 기능을 제공합니다. 큐 구독을 만들려면 다음과 같이 [큐 속성을 사용](#)합니다:

@@파일명 (메인)

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>
  (AppModule, {
```

```
transport: Transport.NATS, 옵션:
{
  서버를 추가합니다:
  [ 'nats://localhost:4222' ], queue:
  'cats_queue',
},
});
```

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. NATS 트랜스포터를 사용하는 경우 `NatsContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  console.log(`Subject: ${context.getSubject()}`);
}
```

정보 힌트 `@Payload()`, `@Ctx()` 및 `NatsContext`는 `nestjs/microservices` 패키지.

와일드카드

구독은 명시적인 주제에 대한 구독일 수도 있고 와일드카드를 포함할 수도 있습니다.

```
@@파일명() @메시지패턴('time.us.*')
getDate(@Payload() 데이터: 숫자[], @Ctx() context: NatsContext) {
  console.log(`Subject: ${context.getSubject()}`); // 예: "time.us.east"
  return new Date().toLocaleTimeString(...);
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*')
getDate(data, context) {
  console.log(`Subject: ${context.getSubject()}`); // 예: "time.us.east"
  return new Date().toLocaleTimeString(...);
}
```

레코드 빌더

메시지 옵션을 구성하려면 [NatsRecordBuilder](#) 클래스를 사용할 수 있습니다(참고: 이벤트 기반 흐름에서도 이 작업이 가능합니다). 예를 들어, [x 버전](#) 헤더를 추가하려면 다음과 같이 [setHeaders](#) 메서드를 사용합니다:

```
'nats'에서 *를 nats로 가져옵니다;

// 코드 어딘가에 const headers =
nats.headers();
headers.set('x-version', '1.0.0');

const record = new NatsRecordBuilder(':cat:').setHeaders(headers).build();
this.client.send('replace-emoji', record).subscribe(...);
```

정보 힌트 [NatsRecordBuilder](#) 클래스는 [@nestjs/microservices](#) 패키지에서 내보내집니다.

서버 측에서도 다음과 같이 [NatsContext](#)에 액세스하여 이러한 헤더를 읽을 수 있습니다:

```
@@파일명() @@메시지패턴('이모티콘 대체')
replaceEmoji(@Payload() 데이터: 문자열, @Ctx() context: NatsContext): 문자열
{
  const headers = context.getHeaders();
  return headers['x-version'] === '1.0.0' ? '□' : '🐱';
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const headers = context.getHeaders();
  return headers['x-version'] === '1.0.0' ? '□' : '🐱';
}
```

경우에 따라 여러 요청에 대한 헤더를 구성하고 싶을 수 있으며, 이를 [ClientProxyFactory](#)에 옵션으로 전달 할 수 있습니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/microservices'에서 { ClientProxyFactory, Transport }를 가져옵니다;
```

```
모듈({ providers:  
[  
  {  
    제공: 'API_v1', useFactory:  
      () =>  
        ClientProxyFactory.create({  
          transport: Transport.NATS,  
          options: {  
            서버: ['nats://localhost:4222'],  
            headers: { 'x-version': '1.0.0' },  
          },  
        },  
  },  
]
```

```
    } ) ,  
    } ,  
] ,  
}  
내보내기 클래스 ApiModule {}
```

RabbitMQ

RabbitMQ는 여러 메시징 프로토콜을 지원하는 오픈소스 경량 메시지 브로커입니다. 분산 및 연합 구성으로 배포하여 대규모, 고가용성 요구 사항을 충족할 수 있습니다. 또한, 전 세계적으로 소규모 스타트업과 대기업에서 가장 널리 배포된 메시지 브로커입니다.

설치

RabbitMQ 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save amqplib amqp-connection-manager
```

개요

RabbitMQ 트랜스포터를 사용하려면 `createMicroservice()`에 다음 옵션 객체를 전달합니다.

메서드를 사용합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  transport: Transport.RMQ, 옵션: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      내구성: 거짓
    },
  },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.RMQ,
  옵션: {
    urls: ['amqp://localhost:5672'],
    queue: 'cats_queue',
    queueOptions: {
      내구성: 거짓
    },
  },
});
```

정보 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

옵션 속성은 선택한 트랜스포터에 따라 다릅니다. RabbitMQ 트랜스포터는 아래에 설명된 속성을 노출합니다.

URL	연결 URL
큐	서버가 수신 대기할 대기열 이름 프리페치 카운트
운트	채널에 대한 프리페치 횟수를 설정합니다.
<hr/>	
채널별 프리페치 활성화	
noAck	거짓이면 수동 승인 모드가 활성화됩니다.
queueOptions	추가 대기열 옵션(자세한 내용은 여기 를 참조하세요)
하세요) 소켓 옵션	추가 소켓 옵션(여기에서 자세히 읽기) 헤더
<hr/>	
모든 메시지와 함께 전송할 헤더	

클라이언트

다른 마이크로서비스 전송기와 마찬가지로 RabbitMQ [ClientProxy](#)를 생성하기 위한 몇 가지 옵션이 있습니다. 인스턴스입니다.

인스턴스를 생성하는 한 가지 방법은 [ClientsModule](#)을 사용하는 것입니다. [ClientsModule](#)을 사용하여 클라이언트 인스턴스를 만들려면 인스턴스를 임포트한 후 [register\(\)](#) 메서드를 사용하여 위에서 [createMicroservice\(\)](#) 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 주입 토큰으로 사용할 이름 속성을 전달합니다. 클라이언트 모듈에 대한 자세한 내용은 [여기](#)를 참조하세요.

```
모듈({ import: [
  ClientsModule.register([
    {
      이름: 'MATH_SERVICE',
      transport: Transport.RMQ, 옵션:
      {
        urls: ['amqp://localhost:5672'],
        queue: 'cats_queue',
        queueOptions: {
          내구성: 거짓
        },
      },
      ],
    ],
  ]),
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 자세한 정보에 액세스하고 싶을 수 있습니다. RabbitMQ 트랜스포터 사용 시, `RmqContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
    console.log(`패턴: ${context.getPattern()}`);
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
    console.log(`패턴: ${context.getPattern()}`);
}
```

정보 힌트 `@Payload()`, `@Ctx()` 및 `RmqContext`는 `nestjs/microservices` 패키지.

속성, 필드 및 콘텐츠가 포함된 원본 RabbitMQ 메시지에 액세스하려면

`getMessage()` 메서드를 다음과 같이 호출합니다:

```
@@파일명() @메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
    console.log(context.getMessage());
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
    console.log(context.getMessage());
}
```

RabbitMQ 채널에 대한 참조를 검색하려면 `RmqContext`의 `getChannelRef` 메서드를 사용합니다.

객체를 다음과 같이 설정합니다:

```
@@파일명() @@메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
    console.log(context.getChannelRef());
}
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
    console.log(context.getChannelRef());
}
```

메시지 확인

메시지가 손실되지 않도록 하기 위해 RabbitMQ는 [메시지 확인을](#) 지원합니다. 확인은 소비자가 다시 전송하여 특정 메시지가 수신, 처리되었으며 RabbitMQ가 자유롭게 삭제할 수 있음을 RabbitMQ에 알립니다. 소비자가 응답을 보내지 않고 죽으면(채널이 닫히거나, 연결이 끊기거나, TCP 연결이 끊어지면) RabbitMQ는 메시지가 완전히 처리되지 않았다는 것을 이해하고 다시 대기열에 넣습니다.

수동 승인 모드를 사용하려면 `noAck` 속성을 `false`로 설정합니다:

```
옵션: {
  urls: ['amqp://localhost:5672'],
  queue: 'cats_queue',
  noAck: false,
  queueOptions: {
    내구성: 거짓
  },
},
```

수동 소비자 확인이 켜져 있는 경우 작업자가 작업을 완료했음을 알리기 위해 적절한 확인을 보내야 합니다.

```
@@파일명() @@메시지패턴('알림')
getNotifications(@Payload() data: number[], @Ctx() context: RmqContext) {
  const channel = context.getChannelRef();
  const originalMsg = context.getMessage();

  channel.ack(originalMsg);
}

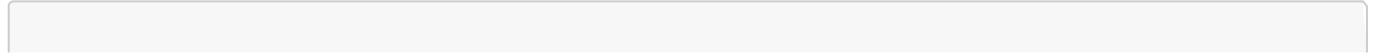
@@switch
@Bind(Payload(), Ctx())
메시지 패턴('알림') getNotifications(데이터,
컨텍스트) {
  const channel = context.getChannelRef();
  const originalMsg = context.getMessage();

  channel.ack(originalMsg);
}
```

레코드 빌더

메시지 옵션을 구성하려면 `RmqRecordBuilder` 클래스를 사용할 수 있습니다(참고: 이벤트 기반 플로우에서도

이 작업을 수행할 수 있음). 예를 들어 [헤더](#) 및 [우선순위](#) 속성을 설정하려면 다음과 같이 [setOptions](#) 메서드를 사용합니다:



```
const message = ':cat:';
const record = new RmqRecordBuilder(message)
  .setOptions({
    headers: {
      ['x-version']: '1.0.0',
    },
    우선순위: 3,
  })
  .build();
```

```
이 클라이언트 보내기('이모티콘 교체', 레코드).구독(...);
```

정보 힌트 `RmqRecordBuilder` 클래스는 [@nestjs/microservices](#) 패키지에서 내보내집니다.

또한 서버 측에서도 다음과 같이 `RmqContext`에 액세스하여 이러한 값을 읽을 수 있습니다:

```
@@파일명() @@메시지패턴('이모티콘 대체')
replaceEmoji(@Payload() 데이터: 문자열, @Ctx() 컨텍스트: RmqContext): 문자열
{
  const { 속성: { 헤더 } } = context.getMessage(); return
  headers['x-version'] === '1.0.0' ? '□' : '喵';
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('replace-emoji')
replaceEmoji(data, context) {
  const { 속성: { 헤더 } } = context.getMessage(); return
  headers['x-version'] === '1.0.0' ? '□' : '喵';
}
```

카프카

카프카는 오픈 소스 분산 스트리밍 플랫폼으로, 세 가지 주요 기능을 갖추고 있습니다:

- 메시지 큐 또는 엔터프라이즈 메시징 시스템과 유사하게 레코드 스트림을 게시하고 구독합니다.
- 내결합성 내구성 있는 방식으로 기록 스트림을 저장하세요.
- 레코드 스트림이 발생하는 대로 처리합니다.

카프카 프로젝트는 실시간 데이터 피드를 처리하기 위해 처리량이 많고 지연 시간이 짧은 통합 플랫폼을 제공하는 것을 목표로 합니다. 실시간 스트리밍 데이터 분석을 위해 Apache Storm 및 Spark와 매우 잘 통합됩니다.

설치

Kafka 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i -- 저장 카프카즈
```

개요

다른 Nest 마이크로서비스 전송 계층 구현과 마찬가지로, 아래와 같이 `createMicroservice()` 메서드에 전달된 옵션 객체의 `전송` 속성을 옵션 `옵션` 속성과 함께 사용하여 Kafka 전송기 메커니즘을 선택합니다:

```
@@파일명(메인)
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  운송: Transport.KAFKA, 옵션: {
    클라이언트: {
      브로커: ['localhost:9092'],
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  transport: Transport.KAFKA,
  옵션: { client:
    {
      브로커: ['localhost:9092'],
    }
  }
})
```

정부 힌트 `Transport` 열거형은 `@nestjs/microservices` 패키지에서 가져옵니다.

옵션

옵션 속성은 선택한 트랜스포터에 따라 다릅니다. 카프카 트랜스포터는 아래에 설명된 속성을 노출합니다.

클라이언트	클라이언트 구성 옵션(여기에서 자세히 읽기) 소비
자	소비자 구성 옵션(여기에서 자세히 읽기) 실행
	실행 구성 옵션(여기에서 자세히 읽기) 구독하기
	구독 구성 옵션(여기에서 자세히 읽기) 생산자
	프로듀서 구성 옵션(여기에서 자세히 읽기) 보내기
	보내기 구성 옵션(여기에서 자세히 읽기)
프로듀서 전용 모드	소비자 그룹 등록을 건너뛰고 생산자 역할만 수행하는 기능 플래그 (부울)
postfixId	clientId 값의 접미사 변경(문자열)

클라이언트

다른 마이크로서비스 전송자와 비교했을 때 Kafka에는 약간의 차이가 있습니다. 대신

[ClientProxy](#) 클래스를 사용합니다.

다른 마이크로서비스 전송자와 마찬가지로, [ClientKafka](#) 인스턴스를 생성하는 데는 [몇 가지 옵션이](#) 있습니다. 인스턴스를 생성하는 한 가지 방법은 [ClientsModule](#)을 사용하는 것입니다. 클라이언트 인스턴스를 생성하려면

클라이언트 모듈을 가져온 다음 [register\(\)](#) 메서드를 사용하여 위에서 [createMicroservice\(\)](#) 메서드에 표시된 것과 동일한 속성을 가진 옵션 객체와 인젝션 토큰으로 사용할 이름 속성을 전달합니다. [클라이언트 모듈](#)에 대한 자세한 내용은 [여기를](#) 참조하세요.

```
모듈({ import: [
  ClientsModule.register([
    {
      이름: 'HERO_SERVICE',
      운송: Transport.KAFKA, 옵션: {
        클라이언트: {
          clientId: 'hero',
          브로커: ['localhost:9092'],
        },
        소비자: {
          groupId: 'hero-consumer'
        }
      },
    ],
  ]),
  ...
})
```

클라이언트를 생성하는 다른 옵션(`ClientProxyFactory` 또는 `@Client()`)도 사용할 수 있습니다. 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

다음과 같이 `@Client()` 데코레이터를 사용합니다:

```
@Client({  
    운송: Transport.KAFKA, 옵션: {  
        클라이언트: {  
            clientId: 'hero',  
            브로커: ['localhost:9092'],  
        },  
        소비자: {  
            groupId: 'hero-consumer'  
        }  
    }  
})  
클라이언트: ClientKafka;
```

메시지 패턴

Kafka 마이크로서비스 메시지 패턴은 요청 채널과 응답 채널에 두 개의 토픽을 사용합니다.

`ClientKafka#send()` 메서드는 [상관관계 ID](#), 회신 토픽, 회신 파티션을 요청 메시지와 연결하여 [반환 주소가 포함된](#) 메시지를 전송합니다. 이를 위해서는 메시지를 보내기 전에 `ClientKafka` 인스턴스가 회신 토픽에 가입되어 있어야 하고 적어도 하나의 파티션에 할당되어야 합니다.

따라서 실행 중인 모든 Nest 애플리케이션에 대해 회신 주제 파티션이 하나 이상 있어야 합니다. 예를 들어 4개의 Nest 애플리케이션을 실행 중인데 답장 주제에 3개의 파티션만 있는 경우에는 메시지를 보내려고 할 때 Nest 애플리케이션 중 1개에서 오류가 발생합니다.

새 `ClientKafka` 인스턴스가 시작되면 소비자 그룹에 가입하고 해당 토픽을 구독합니다. 이 프로세스는 소비자 그룹의 소비자에게 할당된 토픽 파티션의 재밸런싱을 트리거합니다.

일반적으로 주제 파티션은 애플리케이션 실행 시 무작위로 설정된 소비자 이름별로 정렬된 소비자 컬렉션에 주제 파티션을 할당하는 라운드 로빈 파티셔너를 사용하여 할당됩니다. 그러나 새 소비자가 소비자 그룹에 가입하면 새 소비자는 소비자 컬렉션 내의 어느 곳에나 배치될 수 있습니다. 따라서 기존 소비자가 새 소비자 다음에 배치될 때 기존 소비자에게 다른 파티션이 할당될 수 있는 조건이 만들어집니다. 결과적으로 다른 파티션이 할당된 소비자는 리

밸런싱 전에 전송된 요청에 대한 응답 메시지를 잃게 됩니다.

`ClientKafka` 소비자들이 응답 메시지를 잃지 않도록 하기 위해 Nest에 내장된 사용자 정의 파티셔너가 사용됩니다. 이 사용자 정의 파티셔너는 애플리케이션 시작 시 설정된 고해상도 타임스탬프(`process.hrtime()`)를 기준으로 정렬된 소비자 컬렉션에 파티션을 할당합니다.

메시지 응답 구독

경고 참고 이 섹션은 `요청-응답` 메시지 스타일을 사용하는 경우에만 해당됩니다(

`메시지패턴` 데코레이터와 `ClientKafka#send` 메서드). 응답 구독하기

토픽은 [이벤트 기반](#) 통신에 필요하지 않습니다(@EventPattern 데코레이터 및 메소드를 전송합니다).

[ClientKafka](#) 클래스는 `subscribeToResponseOf()` 메서드를 제공합니다. `subscribeToResponseOf()` 메서드는 요청의 토픽 이름을 인수로 받아 파생된 응답 토픽 이름을 응답 토픽 컬렉션에 추가합니다. 이 메서드는 메시지 패턴을 구현할 때 필요합니다.

```
@@filename(heroes.controller)
onModuleInit() {
    this.client.subscribeToResponseOf('hero.kill.dragon');
}
```

[ClientKafka](#) 인스턴스가 비동기적으로 생성된 경우, `connect()` 메서드를 호출하기 전에 `subscribeToResponseOf()` 메서드를 호출해야 합니다.

```
@@filename(heroes.controller)
async onModuleInit() {
    this.client.subscribeToResponseOf('hero.kill.dragon');
    await this.client.connect();
}
```

수신

Nest는 들어오는 Kafka 메시지를 키, 값, 헤더 속성이 있는 객체로 수신하며, 그 값은 [Buffer](#) 유형입니다. 그런 다음 Nest는 버퍼를 문자열로 변환하여 이러한 값을 구문 분석합니다. 문자열이 "object like"인 경우 Nest는 문자열을 [JSON으로](#) 구문 분석하려고 시도합니다. 그런 다음 해당 값은 연결된 핸들러로 전달됩니다.

발신

Nest는 이벤트를 게시하거나 메시지를 보낼 때 직렬화 프로세스를 거친 후 발신 Kafka 메시지를 보냅니다. 이는 [ClientKafka emit\(\)](#) 및 `send()` 메서드에 전달된 인수 또는 @MessagePattern 메서드에서 반환된 값에 대해 발생합니다. 이 직렬화는 `JSON.stringify()` 또는 `toString()` 프로토타입 메서드를 사용하여 문자열이나 버퍼가 아닌 객체를 "문자열화"합니다.

```
@@파일명(heroes.controller) @Controller()
export class HeroesController {
    @MessagePattern('hero.kill.dragon')
    killDragon(@Payload() message: KillDragonMessage): any {
        const dragonId = message.dragonId;
        const items = [
            { id: 1, name: '신화검' },
            { id: 2, name: '던전 열쇠' },
        ];
    }
}
```

```
    품목 반품;  
}  
}
```

정보 힌트 `@Payload()`는 `@nestjs/microservices`에서 가져옵니다.

`키` 및 `값` 속성이 있는 개체를 전달하여 발신 메시지에 키를 지정할 수도 있습니다. 메시지 키 지정은 [공동 파티셔닝 요구 사항을](#) 충족하는 데 중요합니다.

```
@@파일명(heroes.controller) @Controller()  
export class HeroesController {  
  @MessagePattern('hero.kill.dragon')  
  killDragon(@Payload() message: KillDragonMessage): any {  
    const 영역 = '동자';  
    const heroId = message.heroId;  
    const dragonId = message.dragonId;  
  
    const items = [  
      { id: 1, name: '신화검' },  
      { id: 2, name: '던전 열쇠' },  
    ];  
  
    반환 { headers:  
      {  
        영역  
      },  
      키: heroId,  
      값: items  
    }  
  }  
}
```

또한 이 형식으로 전달되는 메시지에는 `헤더` 해시 속성에 설정된 사용자 지정 헤더도 포함될 수 있습니다. 헤더 해시 속성 값은 `문자열` 또는 `버퍼` 유형 중 하나여야 합니다.

```
@@파일명(heroes.controller) @Controller()
export class HeroesController {
    @MessagePattern('hero.kill.dragon')
    killDragon(@Payload() message: KillDragonMessage): any {
        const 영역 = '동지';
        const heroId = message.heroId;
        const dragonId = message.dragonId;

        const items = [
            { id: 1, name: '신화검' },
            { id: 2, name: '던전 열쇠' },
        ];
    }
}
```

```
반환 { headers:  
    {  
        kafka_nestRealm: 영역  
    },  
    키: heroId,  
    값: items  
}  
}  
}
```

이벤트 기반

요청-응답 방식은 서비스 간에 메시지를 주고받는 데는 이상적이지만, 메시지 스타일이 이벤트 기반인 경우, 즉 응답을 기다리지 않고 이벤트를 게시하려는 경우에는 적합하지 않습니다(카프카에 이상적). 이 경우 두 개의 토픽을 유지 관리하기 위해 요청-응답에 필요한 오버헤드를 원하지 않을 수 있습니다.

이에 대해 자세히 알아보려면 다음 두 섹션을 확인하세요: [개요: 이벤트 기반](#) 및 [개요: 이벤트 게시하기](#).

컨텍스트

보다 정교한 시나리오에서는 들어오는 요청에 대한 더 많은 정보에 액세스하고 싶을 수 있습니다. Kafka 트랜스포터 사용 시, `KafkaContext` 객체에 액세스할 수 있습니다.

```
@@파일명() @메시지패턴('hero.kill.dragon')

killDragon(@Payload() 메시지: KillDragonMessage, @Ctx() context:
KafkaContext) {
    console.log(`Topic: ${context.getTopic()}`);
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('hero.kill.dragon')
killDragon(message, context) {
    console.log(`Topic: ${context.getTopic()}`);
}
```

정보 힌트 `@Payload()`, `@Ctx()`, `KafkaContext`는 `nestjs/microservices` 패키지.

원본 카프카 인커밍메시지 객체에 접근하기 위해서는 `KafkaContext` 객체를 다음과 같이 생성합니다:

```
@@파일명() @메시지패턴('hero.kill.dragon')
killDragon(@Payload() 메시지: KillDragonMessage, @Ctx() 컨텍스트:
```

```

KafkaContext) {
    const originalMessage = context.getMessage();
    const partition = context.getPartition();
    const { 헤더, 타임스탬프 } = originalMessage;
}

@@switch
@Bind(Payload(), Ctx())
@MessagePattern('hero.kill.dragon')
killDragon(message, context) {
    const originalMessage = context.getMessage();
    const partition = context.getPartition();
    const { 헤더, 타임스탬프 } = originalMessage;
}

```

여기서 IncomingMessage는 다음 인터페이스를 충족합니다:

인터페이스 IncomingMessage { 주제:

문자열;

파티션: 숫자; 타임스탬

프: 문자열; 크기: 숫자

; 속성: 숫자; 오프셋:

문자열; 키: any;

값: any;

헤더를 추가합니다: 레코드<스트링, 임의>;

}

핸들러에서 수신된 각 메시지의 처리 시간이 느린 경우 하트비트 콜백을 사용하는 것이 좋습니다. 하트비트 함수를 검색하려면 다음과 같이 KafkaContext의 getHeartbeat() 메서드를 사용하세요:

```

@@파일명() @메시지패턴('hero.kill.dragon')

비동기 퀄드래곤(@Payload() 메시지: KillDragonMessage, @Ctx() context:
KafkaContext) {
    const heartbeat = context.getHeartbeat();

    // 느린 처리 대기 doWorkPart1()을
    수행합니다;

    // 세션타임아웃을 초과하지 않도록 하트비트를 전송합니다;

    // 느린 처리를 다시 대기하면서
    doWorkPart2()를 기다립니다;
}

```

이름 지정 규칙

Kafka 마이크로서비스 구성 요소는 Nest 마이크로서비스 클라이언트와 서버 구성 요소 간의 충돌을 방지하기 위해 `client.clientId` 및 `consumer.groupId` 옵션에 각자의 역할에 대한 설명을 추가합니다. 기본적으로 `ClientKafka` 구성 요소는 이 두 옵션에 `-client`을 추가하고 `ServerKafka` 구성 요소는 `-server`를 추가합니다. 아래에 제공된 값이 이러한 방식으로 어떻게 변환되는지 참고하세요(주석에 표시된 대로).

@@파일명(메인)

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
    운송: Transport.KAFKA, 옵션: {
        클라이언트: {
            clientId: 'hero', // 히어로 서버 브
            로커: ['localhost:9092'],
        },
        소비자: {
            groupId: 'hero-consumer' // hero-consumer-server
        },
    }
});
```

그리고 클라이언트를 위해서도요:

@@파일명(heroes.controller)

```
@Client({
    운송: Transport.KAFKA, 옵션: {
        클라이언트: {
            clientId: 'hero', // 히어로-클라이언
            브로커: ['localhost:9092'],
        },
        소비자: {
            groupId: 'hero-consumer' // hero-consumer-client
        }
    }
})
```

클라이언트: `ClientKafka`;

정보 힌트 카프카 클라이언트 및 소비자 이름 지정 규칙을 확장하여 사용자 지정할 수 있습니다.

`ClientKafka` 및 `KafkaServer`를 사용자 지정 공급자에 추가하고 생성자를 재정의하세요.

Kafka 마이크로서비스 메시지 패턴은 요청 채널과 응답 채널에 대해 두 개의 토픽을 사용하므로, 응답 패턴은 요청 토픽에서 파생되어야 합니다. 기본적으로 응답 토픽의 이름은 요청 토픽 이름에 `.reply`가 추가된 합성어입니다.

```
@@filename(heroes.controller)
onModuleInit() {
    this.client.subscribeToResponseOf('hero.get'); // hero.get.reply
}
```

정보 힌트 카프카 응답 주제 명명 규칙은 사용자 지정 공급자에서 ClientKafka를 확장하고

getResponsePatternName 메서드를 재정의하여 사용자 지정할 수 있습니다.

검색 가능한 예외

다른 전송기와 마찬가지로, 처리되지 않은 모든 예외는 자동으로 RpcException으로 래핑되어 "사용자 친화적인" 형식으로 변환됩니다. 그러나 이 메커니즘을 우회하여 예외를 kafkaJS 드라이버가 대신 사용하도록 하고 싶은 예외적인 경우가 있습니다. 메시지를 처리할 때 예외를 던지면 kafkaJS가 메시지를 재시도(재전송)하도록 지시하므로 메시지(또는 이벤트) 핸들러가 트리거되었더라도 오프셋이 Kafka에 커밋되지 않습니다.

경고 경고 이벤트 핸들러(이벤트 기반 통신)의 경우 처리되지 않은 모든 예외는 기본적으로 검색 가능한 예외로 간주됩니다.

이를 위해 다음과 같이 KafkaRetriableException이라는 전용 클래스를 사용할 수 있습니다:

새로운 KafkaRetriableException('...')을 던집니다;

정보 힌트 KafkaRetriableException 클래스는 @nestjs/microservices에서 내보내집니다.
패키지입니다.

커밋 오프셋

카프카로 작업할 때는 오프셋 커밋이 필수입니다. 기본적으로 메시지는 특정 시간이 지나면 자동으로 커밋됩니다. 자세한 내용은 [KafkaJS 문서](#)를 참조하세요. ClientKafka는 기본 KafkaJS 구현처럼 작동하는 오프셋을 수동으로 커밋하는 방법을 제공합니다.

```
@@파일명() @EventPattern('user.created')

비동기 처리 사용자 생성(@Payload() 데이터: IncomingMessage, @Ctx() context:
KafkaContext) {
    // 비즈니스 로직

    const { 오프셋 } = context.getMessage(); const
    partition = context.getPartition(); const
    topic = context.getTopic();
    await this.client.commitOffsets([{ topic, partition, offset }])
}

@@switch
@Bind(Payload(), Ctx())
@EventPattern('user.created')

async handleUserCreated(데이터, 컨텍스트) {
```

```
// 비즈니스 로직

const { 오프셋 } = context.getMessage(); const
partition = context.getPartition(); const
topic = context.getTopic();
await this.client.commitOffsets([{ topic, partition, offset }])
}
```

메시지의 자동 커밋을 비활성화하려면 다음과 같이 실행 구성에서 `autoCommit: false`를 설정합니다:

@@파일명(메인)

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>
(AppModule, {
  운송: Transport.KAFKA, 옵션: {
    클라이언트: {
      브로커: ['localhost:9092'],
    },
    run: {
      자동 커밋: false
    }
  }
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
  транспорт: Transport.KAFKA,
  опции: { клиент:
    {
      брокер: ['localhost:9092'],
    },
    run: {
      автоматический коммит: false
    }
  }
});
```

gRPC

gRPC는 모든 환경에서 실행할 수 있는 최신 오픈 소스 고성능 RPC 프레임워크입니다. 로드 밸런싱, 추적, 상태 확인 및 인증에 대한 플러그형 지원을 통해 데이터센터 안팎의 서비스를 효율적으로 연결할 수 있습니다.

많은 RPC 시스템과 마찬가지로 gRPC는 원격으로 호출할 수 있는 함수(메서드)로 서비스를 정의하는 개념을 기반으로 합니다. 각 메서드에 대해 매개변수와 반환 유형을 정의합니다. 서비스, 매개변수 및 반환 유형은 Google의 오픈 소스 언어 중립 [프로토콜 버퍼](#) 메커니즘을 사용하여 `.proto` 파일에 정의됩니다.

Nest는 `.proto` 파일을 사용하여 클라이언트와 서버를 동적으로 바인딩함으로써 원격 프로시저 호출을 쉽게 구현하고 구조화된 데이터를 자동으로 직렬화 및 역직렬화할 수 있도록 합니다.

설치

gRPC 기반 마이크로서비스 구축을 시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save @grpc/grpc-js @grpc/proto-loader
```

개요

다른 Nest 마이크로서비스 전송 계층 구현과 마찬가지로, `createMicroservice()` 메서드에 전달된 옵션 객체의 [전송](#) 속성을 사용하여 gRPC 전송기 메커니즘을 선택합니다. 다음 예제에서는 영웅 서비스를 설정하겠습니다. [옵션](#) 속성은 해당 서비스에 대한 메타데이터를 제공하며, 그 속성은 [아래에](#) 설명되어 있습니다.

@@파일명(메인)

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(AppModule, {
    transport: Transport.GRPC, 옵션:
    {
        패키지: '영웅',
        protoPath: join(__dirname, 'hero/hero.proto'),
    },
});
@@switch
const app = await NestFactory.createMicroservice(AppModule, {
    transport: Transport.GRPC,
    옵션을 추가합니다: {
        package: 'hero',
        protoPath: join(__dirname, 'hero/hero.proto'),
    },
});
```

정보 힌트 `join()` 함수는 `path` 패키지에서 가져오고, `Transport` 열거형은

`@nestjs/microservices` 패키지에서 가져옵니다.

`nest-cli.json` 파일에 TypeScript가 아닌 파일을 배포할 수 있는 자산 속성을 추가하고, 모든 TypeScript가 아닌 자산 감시를 켜기 위해 `watchAssets` 속성을 추가합니다. 이 경우 `.proto` 파일이 `dist` 폴더에 자동으로 복사되도록 합니다.

```
{
  "컴파일러옵션": {
    "assets": [
      "**/*.proto",
      "watchAssets": true
    ]
  }
}
```

옵션

gRPC 트랜스포터 옵션 개체는 아래에 설명된 속성을 노출합니다.

패키지	Protobuf 패키지 이름(.proto 파일의 패키지 설정과 일치). 필수
protoPath	.proto 파일의 절대(또는 루트 디렉터리에 상대) 경로입니다. 필수
url	연결 URL. IP 주소/dns 이름:포트 형식의 문자열(예, 'localhost:50051') 트랜스포터가 연결을 설정하는 주소/포트를 정의합니다. 선택 사항입니다. 기본값은 'localhost:5000'입니다.
protoLoader	.proto 파일을 로드할 유ти리티의 NPM 패키지 이름입니다. 선택 사항입니다. 기본값은 <code>'@grpc/proto-loader'</code>
로더	grpc/proto-loader 옵션. 이를 통해 다음과 같은 동작을 세부적으로 제어할 수 있습니다. .proto 파일. 선택 사항입니다. 자세한 내용은 여기 를 참조하세요.
자격 증명	서버 자격 증명. 선택 사항입니다. 여기 에서 자세히 알아보기

샘플 gRPC 서비스

`HeroesService`라는 샘플 gRPC 서비스를 정의해 보겠습니다. 위의 옵션 오브젝트에서 `protoPath` 프로퍼티는 `.proto` 정의 파일 `hero.proto`의 경로를 설정합니다. `hero.proto` 파일은 [프로토콜 버퍼](#)를 사용하여

구조화되어 있습니다. 파일은 다음과 같습니다:

```
// hero/hero.proto 구문
= "proto3";

패키지 영웅;

서비스 히어로즈서비스 {
    rpc FindOne (HeroById) 반환 (Hero) {}
}
```

```
메시지 HeroById {
    int32 id = 1;
}
```

```
메시지 Hero { int32
    id = 1; 문자열
    name = 2;
}
```

히어로서비스는 `FindOne()` 메서드를 노출합니다. 이 메서드는 `HeroById` 유형의 입력 인수를 받고 `Hero` 메시지를 반환합니다(프로토콜 버퍼는 `메시지` 요소를 사용하여 매개변수 유형과 반환 유형을 모두 정의합니다).

다음으로 서비스를 구현해야 합니다. 이 정의를 충족하는 핸들러를 정의하기 위해 아래와 같이 컨트롤러에서 `@GrpcMethod()` 데코레이터를 사용합니다. 이 데코레이터는 메서드를 gRPC 서비스 메서드로 선언하는 데 필요한 메타데이터를 제공합니다.

정보 힌트 이전 마이크로서비스 챕터에서 소개한 `@MessagePattern()` 데코레이터([자세히 읽기](#))는 gRPC 기반 마이크로서비스에는 사용되지 않습니다. gRPC 기반 마이크로서비스에서는

```
@@파일명(heroes.controller) @Controller()
export class HeroesController {
    @GrpcMethod('HeroesService', 'FindOne')
    findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: ServerUnaryCall<any,
    any>): Hero {
        const items = [
            { id: 1, name: 'John' },
            { id: 2, name: 'Doe' },
        ];
        반환 항목.찾기(({ id }) => id === 데이터.id);
    }
}
@@스위치
@Controller()
export class HeroesController {
    @GrpcMethod('HeroesService', 'FindOne')
    findOne(data, metadata, call) {
        const items = [
            { id: 1, name: 'John' },
            { id: 2, name: 'Doe' },
        ];
        반환 항목.찾기(({ id }) => id === 데이터.id);
    }
}
```

정보 힌트 `@GrpcMethod()` 데코레이터는 [@nestjs/microservices](#) 패키지에서, 메타데이터와

`ServerUnaryCall`은 `grpc` 패키지에서 가져옵니다.

위에 표시된 데코레이터는 두 개의 인수를 받습니다. 첫 번째 인수는 `hero.proto`의 히어로즈서비스 서비스 정의에 해당하는 서비스 이름(예: '`HeroesService`')입니다. 두 번째 인자(문자열 '`FindOne`')는 `hero.proto` 파일의 히어로즈서비스 내에 정의된 `FindOne()` rpc 메서드에 해당합니다.

`findOne()` 핸들러 메서드에는 호출자가 전달한 데이터, gRPC 요청 메타데이터를 저장하는 메타데이터, 클라이언트에 메타데이터를 전송하기 위한 `sendMetadata`와 같은 `GrpcCall` 객체 속성을 가져오기 위한 `호출 등` 세 가지 인수가 필요합니다.

두 `@GrpcMethod()` 데코레이터 인수는 모두 선택 사항입니다. 두 번째 인수(예: '`FindOne`') 없이 호출하면 Nest는 처리기 이름을 낙타 대문자로 변환하여 `.proto` 파일 rpc 메서드를 처리기와 자동으로 연결합니다(예: `findOne` 처리기는 `FindOne` rpc 호출 정의와 연결됨). 이는 아래와 같습니다.

```
@@파일명(heroes.controller) @Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: ServerUnaryCall<any,
  any>): Hero {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}

@@스위치
@Controller()
export class HeroesController {
  @GrpcMethod('HeroesService')
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}
```

첫 번째 `@GrpcMethod()` 인수를 생략할 수도 있습니다. 이 경우 Nest는 처리기가 정의된 클래스 이름을 기준으로 처리기를 프로토 정의 파일의 서비스 정의와 자동으로 연결합니다. 예를 들어, 다음 코드에서 히어로즈서비스 클래스는 '히어로즈서비스'라는 이름의 일치 여부에 따라 핸들러 메서드를 `hero.proto` 파일의 히어로즈서비스

서비스 정의와 연결합니다.

```
@@파일명(heroes.controller) @Controller()
내보내기 클래스 HeroesService {
```

```

@GrpcMethod()
findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: ServerUnaryCall<any,
any>): Hero {
  const items = [
    { id: 1, name: 'John' },
    { id: 2, name: 'Doe' },
  ];
  반환 항목.찾기(({ id }) => id === 데이터.id);
}

@@스위치

@Controller()
내보내기 클래스 HeroesService {
  @GrpcMethod()
  findOne(data, metadata, call) {
    const items = [
      { id: 1, name: 'John' },
      { id: 2, name: 'Doe' },
    ];
    반환 항목.찾기(({ id }) => id === 데이터.id);
  }
}

```

클라이언트

Nest 애플리케이션은 `.proto` 파일에 정의된 서비스를 소비하는 gRPC 클라이언트로 작동할 수 있습니다.

`ClientGrpc` 객체를 통해 원격 서비스에 액세스합니다. 여러 가지 방법으로 `ClientGrpc` 객체를 얻을 수 있습니다.

선호하는 기법은 `ClientsModule`을 임포트하는 것입니다. `register()` 메서드를 사용하여 `.proto` 파일에 정의된 서비스 패키지를 인젝션 토큰에 바인딩하고 서비스를 구성할 수 있습니다. 이를 속성은 인젝션 토큰입니다. gRPC 서비스의 경우 `전송`을 사용합니다: `Transport.GRPC`를 사용합니다. 옵션 속성은 `위에서` 설명한 것과 동일한 속성을 가진 개체입니다.

```
임포트합니다: [  
  ClientsModule.register([  
    {  
      이름: 'HERO_PACKAGE',  
      transport: Transport.GRPC, 옵션: {  
        패키지: '영웅',  
        protoPath: join(__dirname, 'hero/hero.proto'),  
      },  
    },  
  ]),  
];
```

정보 힌트 `register()` 메서드는 객체 배열을 받습니다. 쉼표로 구분된 등록 객체 목록을 제공하여 여러 패키지를 등록할 수 있습니다.

등록이 완료되면 `@Inject()`을 사용하여 구성된 `ClientGrpc` 객체를 삽입할 수 있습니다. 그런 다음 `ClientGrpc` 객체의 `getService()` 메서드를 사용하여 아래와 같이 서비스 인스턴스를 검색할 수 있습니다.

```
@Injectable()
export class AppService implements OnModuleInit {
    private heroesService: HeroesService;

    constructor(@Inject('HERO_PACKAGE') private client: ClientGrpc) {}

    onModuleInit() {}
        this.heroesService = this.client.getService<HeroesService>
        ('히어로즈서비스');
    }

    getHero(): Observable<string> {
        return this.heroesService.findOne({ id: 1 });
    }
}
```

오류 경고 gRPC 클라이언트는 프로토 로더 구성(마이크로서비스 트랜스포터 구성의 `options.loader.keepcase`)에서 `keepCase` 옵션이 `true`로 설정되어 있지 않으면 이름에 밑줄 `_이` 포함된 필드를 전송하지 않습니다.

다른 마이크로서비스 전송 메서드에 사용된 기술과 비교하면 약간의 차이가 있습니다. `ClientProxy` 클래스 대신 `getService()` 메서드를 제공하는 `ClientGrpc` 클래스를 사용합니다. `getService()` 일반 메서드는 서비스 이름을 인자로 받아 해당 인스턴스(사용 가능한 경우)를 반환합니다.

또는 다음과 같이 `@Client()` 데코레이터를 사용하여 `ClientGrpc` 객체를 인스턴스화할 수 있습니다:

```
@Injectable()
내보내기 클래스 AppService 구현 OnModuleInit {
    @Client({
        transport: Transport.GRPC, 옵션
    : {
        패키지: '영웅',
        protoPath: join(__dirname, 'hero/hero.proto'),
    },
})
클라이언트: ClientGrpc;

private heroesService: HeroesService;

onModuleInit() {
    this.heroesService = this.client.getService<HeroesService>
('HeroesService');
}

getHero(): Observable<string> {
    return this.heroesService.findOne({ id: 1 });
}
```

```

    }
}

```

마지막으로, 더 복잡한 시나리오의 경우 동적으로 구성된 클라이언트를 삽입할 수 있습니다.

`ClientProxyFactory` 클래스입니다.

두 경우 모두 `히어로즈서비스` 프록시 객체에 대한 참조가 생성되며, 이 객체는 `.proto` 파일에 정의된 것과 동일한 메서드 집합을 노출합니다. 이제 이 프록시 객체(즉, 영웅 서비스)에 액세스하면 gRPC 시스템이 자동으로 요청을 직렬화하여 원격 시스템으로 전달하고 응답을 반환한 후 응답을 역직렬화합니다. gRPC는 이러한 네트워크 통신 세부 정보로부터 우리를 보호하기 때문에 `heroesService`는 로컬 공급자처럼 보이고 작동합니다.

모든 서비스 메서드는 언어의 자연스러운 규칙을 따르기 위해 대소문자를 구분합니다. 예를 들어, `.proto` 파일 `히어로즈서비스` 정의에는 `FindOne()` 함수가 포함되어 있지만, `히어로즈서비스` 인스턴스는 `findOne()` 메서드를 제공합니다.

```

인터페이스 히어로즈 서비스 {
  findOne(data: { id: number }): Observable<any>;
}

```

메시지 핸들러는 `Observable`을 반환할 수도 있으며, 이 경우 스트림이 완료될 때까지 결과값이 방출됩니다.

```

@@파일명(heroes.controller) @Get()
호출(): Observable<any> {
  return this.heroesService.findOne({ id: 1 });
}
@@switch
@Get()
call() {
  return this.heroesService.findOne({ id: 1 });
}

```

요청과 함께 gRPC 메타데이터를 보내려면 다음과 같이 두 번째 인수를 전달할 수 있습니다:

```
호출(): Observable<any> {
  const metadata = new Metadata();
  metadata.add('Set-Cookie', 'yummy_cookie=choco');

  return this.heroesService.findOne({ id: 1 }, 메타데이터);
}
```

정보 힌트 **메타데이터** 클래스는 **grpc** 패키지에서 가져옵니다.

이를 위해서는 앞서 몇 단계에 걸쳐 정의한 [하이로즈서비스](#) 인터페이스를 업데이트해야 한다는 점에 유의하세요.

예

작동 예제는 [여기에서](#) 확인할 수 있습

니다. gRPC 스트리밍

gRPC 자체는 일반적으로 [스트림으로](#) 알려진 장기 라이브 연결을 지원합니다. 스트림은 다음과 같습니다. 채팅, 관찰 또는 청크 데이터 전송과 같은 경우에 유용합니다. 자세한 내용은 [여기에서](#) 공식 문서를 참조하세요.

Nest는 두 가지 방법으로 GPRC 스트리밍 핸들러를 지원합니다:

- RxJS [Subject](#) + [Observable](#) 핸들러: 컨트롤러 메서드 내부에서 바로 응답을 작성하거나 [Subject/Observable](#) 소비자에게 전달할 때 유용합니다.
- 순수 GPRC 호출 스트리밍 핸들러: 노드 표준 [듀플렉스](#) 스트리밍 핸들러의 나머지 디스패치를 처리할 일부 실행자에게 전달하는 데 유용할 수 있습니다.

스트리밍 샘플

[HelloService](#)라는 새로운 샘플 gRPC 서비스를 정의해 보겠습니다. [hello.proto](#) 파일은 [프로토콜 버퍼](#)를 사용하여 구조화되어 있습니다. 파일은 다음과 같습니다:

```
// hello/hello.proto 구  
문 = "proto3";  
  
패키지 안녕하세요;  
  
서비스 HelloService {  
    rpc BidiHello(스트림 HelloRequest) 반환 (스트림 HelloResponse);  
    rpc LotsOfGreetings(스트림 HelloRequest) 반환 (HelloResponse);  
}  
  
메시지 HelloRequest {  
    문자열 greeting = 1;  
}
```

정보필드 HelloResponse { 메서드는 `@GrpcMethod`을 사용하여 간단하게 구현할 수 있습니다.

문자열 reply = 1;
데코레이터를 사용해야 합니다(위의 예제에서처럼). 반환된 스트림은 여러 값을 방출할 수 있기 때문입니다.
}

이 .proto 파일을 기반으로 HelloService 인터페이스를 정의해 보겠습니다:

```
인터페이스 HelloService {  
    bidiHello(업스트림: Observable<HelloRequest>):  
    Observable<HelloResponse>;  
    lotsOfGreetings(  
        업스트림: 관찰 가능<HelloRequest>,  
    ): 관찰 가능 <헬로 응답>;  
}
```

```
인터페이스 HelloRequest { 인사말: 문  
    자열;  
}
```

```
인터페이스 HelloResponse { 회신:  
    문자열;  
}
```

정보 힌트 프로토 인터페이스는 [ts-proto](#) 패키지에 의해 자동으로 생성될 수 있으며, [여기에서](#) 자세히 알아보세요.

주제 전략

[GrpcStreamMethod\(\)](#) 데코레이터는 함수 파라미터를 RxJS [옵저버블로](#) 제공합니다. 따라서 여러 메시지를 수신하고 처리할 수 있습니다.

```
@GrpcStreamMethod()  
비디헬로(메시지: Observable<any>, 메타데이터: 메타데이터, 호출:  
ServerDuplexStream<any, any>): Observable<any> {  
  const subject = new Subject();  
  
  const onNext = message => {  
    console.log(message);  
    subject.next({  
      답장합니다: '안녕, 세상아!'  
    });  
  };  
  const onComplet = () => subject.complete();  
  messages.subscribe({  
    다음: on다음, 완료: on완  
    료,  
  });  
  
  subject.asObservable()을 반환합니다;  
}  
경고 경고 @GrpcStreamMethod() 데코레이터와 전이중 상호 작용을 지원하려면 컨트롤러 메서드가  
RxJS Observable을 반환해야 합니다.
```

정보 힌트 메타데이터 및 서버유나리콜 클래스/인터페이스는 grpc에서 가져옵니다.

패키지입니다.

서비스 정의(.proto 파일)에 따르면, `BidiHello` 메서드는 요청을 서비스로 스트리밍해야 합니다. 클라이언트에서 스트림으로 여러 개의 비동기 메시지를 전송하기 위해 RxJS `ReplaySubject` 클래스를 활용합니다.

```
const helloService = this.client.getService<HelloService>('HelloService');
const helloRequest$ = new ReplaySubject<HelloRequest>();

helloRequest$.next({ greeting: 'Hello (1)!' });
helloRequest$.next({ greeting: 'Hello (2)!' });
helloRequest$.complete();
```

헬로서비스.비디헬로(헬로리퀘스트\$)를 반환합니다;

위의 예제에서는 스트림에 두 개의 메시지를 쓰고(`next()` 호출) 데이터 전송이 완료되었음을 서비스에 알렸습니다(`complete()` 호출).

통화 스트림 핸들러

메서드 반환값이 스트림으로 정의된 경우 `@GrpcStreamCall()` 데코레이터는 함수 매개변수를 다음과 같은 표준 메서드를 지원하는 `grpc.ServerDuplexStream`으로 제공합니다.

`.on('data', callback), .write(message)` 또는 `.cancel()`. 사용 가능한 메서드에 대한 전체 문서는 [여기에서](#) 확인할 수 있습니다.

또는 메서드 반환값이 스트림이 아닌 경우 `@GrpcStreamCall()` 데코레이터는 두 개의 함수 매개변수 `grpc.ServerReadableStream`(자세한 내용은 [여기](#)를 참조하세요)과 콜백을 각각 제공합니다.

먼저 전이중 상호 작용을 지원해야 하는 `BidiHello`를 구현해 보겠습니다.

```
@GrpcStreamCall()
bidiHello(requestStream: any) {
  requestStream.on('data', message => {
    console.log(message);
    requestStream.write({
      답장합니다: '안녕, 세상아!'
    });
  });
}
```

정보 힌트 이 데코레이터는 특정 반환 매개변수를 제공할 필요가 없습니다. 스트림은 다른 표준 스트림 유형

위의 예제에서는 `write()` 메서드를 사용하여 응답 스트림에 객체를 썼습니다. `.on()` 메서드에 두 번째 매개변수로 전달된 콜백은 서비스가 새 데이터 청크를 수신할 때마다 호출됩니다.

LotsOfGreetings 메서드를 구현해 보겠습니다.

```
@GrpcStreamCall()
lotsOfGreetings(requestStream: any, callback: (err: unknown, value:
HelloResponse) => void) {
  requestStream.on('data', message => {
    console.log(message);
  });
  requestStream.on('end', () => callback(null, { reply: '안녕하세요, 세상!' }));
}
```

여기서는 콜백 함수를 사용하여 요청 스트림 처리가 완료되면 응답을 보냈습니다.

gRPC 메타데이터

메타데이터는 키-값 쌍의 목록 형태로 된 특정 RPC 호출에 대한 정보로, 키는 문자열이고 값은 일반적으로 문자열이지만 바이너리 데이터일 수도 있습니다. 메타데이터는 클라이언트가 서버에 호출과 관련된 정보를 제공할 수 있고 그 반대의 경우도 마찬가지입니다. 메타데이터에는 인증 토큰, 모니터링 목적의 요청 식별자 및 태그, 데이터 세트의 레코드 수와 같은 데이터 정보가 포함될 수 있습니다.

GrpcMethod() 핸들러에서 메타데이터를 읽으려면 두 번째 인수(메타데이터)를 사용하세요.
메타데이터(grpc 패키지에서 가져온 것).

핸들러에서 메타데이터를 다시 보내려면 ServerUnaryCall#sendMetadata() 메서드(세 번째 핸들러 인수)를 사용하세요.

```
@@파일명(heroes.controller) @Controller()  
내보내기 클래스 HeroesService {  
    @GrpcMethod()  
    findOne(데이터: HeroById, 메타데이터: 메타데이터, 호출: ServerUnaryCall<any,  
any>): Hero {  
        const serverMetadata = new Metadata();  
        const items = [  
            { id: 1, name: 'John' },  
            { id: 2, name: 'Doe' },  
        ];  
  
        serverMetadata.add('Set-Cookie', 'yummy_cookie=choco');  
        call.sendMetadata(serverMetadata);  
  
        반환 항목.찾기(({ id }) => id === 데이터.id);  
    }  
}  
@@스위치  
@Controller()
```

```

내보내기 클래스 HeroesService {
    @GrpcMethod()
    findOne(데이터, 메타데이터, 호출) {
        const serverMetadata = new Metadata();
        const items = [
            { id: 1, name: 'John' },
            { id: 2, name: 'Doe' },
        ];

        serverMetadata.add('Set-Cookie', 'yummy_cookie=choco');
        call.sendMetadata(serverMetadata);

        반환 항목.찾기(({ id }) => id === 데이터.id);
    }
}

```

마찬가지로 `@GrpcStreamMethod()` 핸들러([주제 전략](#))로 주석이 달린 핸들러에서 메타데이터를 읽으려면 두 번째 인수(메타데이터)를 사용하는데, 이 인수는 (`grpc` 패키지에서 가져온) [메타데이터](#) 유형입니다.

핸들러에서 메타데이터를 다시 보내려면 `ServerDuplexStream#sendMetadata()` 메서드(세 번째 핸들러 인수)를 사용하세요.

[호출 스트림 핸들러](#) 내에서 메타데이터를 읽으려면(`@GrpcStreamCall()`로 주석 처리된 핸들러) 다음과 같이 하세요. 데코레이터)를 사용하여 다음과 같이 요청 스트림 참조의 [메타데이터](#) 이벤트를 수신합니다:

```

requestStream.on('metadata', (metadata: 메타데이터) => {
    const meta = metadata.get('X-Meta');
});

```

맞춤형 운송업체

Nest는 개발자가 새로운 맞춤형 전송 전략을 구축할 수 있는 API뿐만 아니라 다양한 전송기를 기본으로 제공합니다. 전송기를 사용하면 플러그 가능한 통신 계층과 매우 간단한 애플리케이션 수준 메시지 프로토콜을 사용하여 네트워크를 통해 구성 요소를 연결할 수 있습니다(전체 [기사 읽기](#)).

정보 힌트 Nest로 마이크로서비스를 구축한다고 해서 반드시 [@nestjs/microservices](#) 패키지를 사용해야 하는 것은 아닙니다. 예를 들어 외부 서비스(다른 언어로 작성된 다른 마이크로서비스)와 통신하려는 경우 [@nestjs/microservice](#) 라이브러리에서 제공하는 모든 기능이 필요하지 않을 수 있습니다. 실제로 구독자를 선언적으로 정의할 수 있는 데코레이터([@EventPattern](#) 또는 [@MessagePattern](#))가 필요하지 않은 경우에는 [독립형 애플리케이션을](#) 실행하고 채널에 대한 연결/구독을 수동으로 유지하는 것으로 대부분의 사용 사례에 충분하며 더 많은 유연성을 제공할 수 있습니다.

사용자 지정 전송기를 사용하면 모든 메시징 시스템/프로토콜(Google Cloud Pub/Sub, Amazon Kinesis 등)을 통합하거나 기존 전송기를 확장하여 추가 기능(예: MQTT용 [QoS](#))을 추가할 수 있습니다.

정보 힌트 Nest 마이크로서비스의 작동 방식과 기존 전송기의 기능을 확장하는 방법을 더 잘 이해하려면 [NestJS 마이크로서비스 실제 사용 및 고급 NestJS 마이크로서비스](#) 문서 시리즈를 읽어보시기 바랍니다.

전략 만들기

먼저 커스텀 트랜스포터에 해당하는 클래스를 정의해 보겠습니다.

```
'@nestjs/microservices'에서 { CustomTransportStrategy, Server } 임포트; 클  
래스 GoogleCloudPubSubServer  
서버 확장  
CustomTransportStrategy를 구현합니다 {  
  /**  
   * 이 메서드는 "app.listen()"을 실행할 때 트리거됩니다.  
   */  
  listen(callback: () => void) {  
    callback();  
  }  
  
  /**  
   * 이 메서드는 애플리케이션 종료 시 트리거됩니다.  
   */  
  close() {}  
}
```

경고 경고 이 장에서는 모든 기능을 갖춘 Google Cloud Pub/Sub 서버를 구현하지 않으므로 트랜스포터에 대한 구체적인 기술 세부 사항을 살펴봐야 합니다.

위의 예제에서는 `GoogleCloudPubSubServer` 클래스를 선언하고 `CustomTransportStrategy` 인터페이스에 의해 시행되는 `listen()` 및 `close()` 메서드를 제공했습니다. 또한 이 클래스는 `@nestjs/microservices` 패키지에서 가져온 `Server` 클래스를 확장하여 Nest 런타임에서 메시지 핸들러를 등록하는 데 사용되는 메서드와 같은 몇 가지 유용한 메서드를 제공합니다. 또는 기존 전송 전략의 기능을 확장하려는 경우 해당 서버 클래스(예: `ServerRedis`)를 확장할 수 있습니다. 일반적으로 메시지/이벤트 구독(및 필요한 경우 응답)을 담당하기 때문에 클래스에 "서버" 접미사를 추가했습니다.

이제 기본 제공 트랜스포터 대신 다음과 같이 사용자 지정 전략을 사용할 수 있습니다:

```
const app = await NestFactory.createMicroservice<MicroserviceOptions>(
  AppModule,
  {
    전략: 새로운 구글클라우드펍서브서버(),
  },
);
```

기본적으로 `전송` 및 `옵션` 속성이 있는 일반 트랜스포터 옵션 객체를 전달하는 대신, 사용자 지정 트랜스포터 클래스의 인스턴스 값인 `전략이라는` 단일 속성을 전달합니다.

`GoogleCloudPubSubServer` 클래스로 돌아가서, 실제 애플리케이션에서는 메시지 브로커/외부 서비스에 대한 연결을 설정하고 `listen()` 메서드에서 구독자를 등록하고 특정 채널을 수신한 다음 `close()` 해체 메서드에서 구독을 제거하고 연결을 닫을 수 있지만, Nest 마이크로서비스가 서로 통신하는 방식을 잘 이해해야 하므로 이 [문서 시리즈를](#) 읽어보시기를 권장합니다. 대신 이 장에서는 `Server` 클래스가 제공하는 기능과 이를 활용하여 사용자 지정 전략을 구축하는 방법에 중점을 두겠습니다.

예를 들어 애플리케이션 어딘가에 다음과 같은 메시지 핸들러가 정의되어 있다고 가정해 보겠습니다:

```
메시지 패턴('에코') echo(@Payload()) 데
이터: 객체) {
  데이터를 반환합니다;
}
```

이 메시지 핸들러는 Nest 런타임에 의해 자동으로 등록됩니다. `Server` 클래스를 사용하면 어떤 메시지 패턴이 등록되었는지 확인하고 해당 패턴에 할당된 실제 메서드에 액세스하여 실행할 수 있습니다. 이를 테스트하기 위해 `콜백` 함수가 호출되기 전에 `listen()` 메서드 안에 간단한 `console.log`를 추가해 보겠습니다:

```
listen(callback: () => void) {
  console.log(this.messageHandlers);
  callback();
}
```

애플리케이션이 다시 시작되면 터미널에 다음 로그가 표시됩니다:

```
Map { 'echo' => [AsyncFunction] { isEventHandler: false } } }
```

정보 힌트 `@EventPattern` 데코레이터를 사용하면 동일한 출력을 볼 수 있지만 `isEventHandler` 속성을 `true`로 설정합니다.

보시다시피, 메시지 핸들러 속성은 패턴이 키로 사용되는 모든 메시지(및 이벤트) 핸들러의 맵 컬렉션입니다. 이제 키(예: "echo")를 사용하여 메시지 핸들러에 대한 참조를 받을 수 있습니다:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  console.log(await echoHandler('Hello world!'));
  callback();
}
```

임의의 문자열(여기서는 "Hello world!")을 인자로 전달하는 `echoHandler`를 실행하면 콘솔에서 해당 문자열을 볼 수 있습니다:

```
안녕하세요!
```

즉, 메서드 핸들러가 제대로 실행되었다는 뜻입니다.

인터셉터와 함께 커스텀 트랜스포트 전략을 사용할 때 핸들러는 RxJS 스트림으로 래핑됩니다. 즉, 스트림의 기본 로직을 실행하려면 스트림을 구독해야 합니다(예: 인터셉터가 실행된 후 컨트롤러 로직으로 계속 진행).

이에 대한 예는 아래에서 확인할 수 있습니다:

```
async listen(callback: () => void) {
  const echoHandler = this.messageHandlers.get('echo');
  const streamOrResult = await echoHandler('Hello World');
  if (isObservable(streamOrResult)) {
    streamOrResult.subscribe();
  }
  콜백();
}
```

클라이언트 프록시

첫 번째 섹션에서 언급했듯이, 마이크로서비스를 생성할 때 반드시 [@nestjs/microservices](#) 패키지를 사용할 필요는 없지만, 그렇게 하기로 결정하고 사용자 정의 전략을 통합해야 하는 경우 "클라이언트" 클래스도 제공해야 합니다.

정보 힌트 다시 한 번 말씀드리지만, 모든 [@nestjs/마이크로서비스](#) 기능(예: 스트리밍)과 호환되는 완전한 기능을 갖춘 클라이언트 클래스를 구현하려면 프레임워크에서 사용하는 통신 기술에 대한 충분한 이해가 필요합니다. 자세한 내용은 이 [문서를](#) 참조하세요.

외부 서비스와 통신하거나 메시지(또는 이벤트)를 전송 및 게시하려면 다음과 같이 라이브러리별 SDK 패키지를 사용하거나 [ClientProxy](#)를 확장하는 사용자 지정 클라이언트 클래스를 구현할 수 있습니다:

'[@nestjs/microservices](#)'에서 { ClientProxy, ReadPacket, WritePacket }을 가져옵니다;

```
GoogleCloudPubSubClient 클래스 ClientProxy 확장 {  
    async connect(): Promise<any> {}  
    async close() {}  
    async dispatchEvent(packet: ReadPacket<any>): Promise<any> {}  
    publish(  
        패킷을 읽습니다: ReadPacket<any>,  
        콜백: (패킷: WritePacket<any>) => void,  
    ): 함수 {}  
}
```

경고 경고 이 장에서는 모든 기능을 갖춘 Google Cloud Pub/Sub 클라이언트를 구현하지 않으므로 트랜스포터에 대한 구체적인 기술 세부 사항을 살펴봐야 합니다.

보시다시피 [ClientProxy](#) 클래스는 연결 설정 및 종료, 메시지([publish](#)) 및 이벤트([dispatchEvent](#)) 게시를 위한 몇 가지 메서드를 제공해야 합니다. 요청-응답 통신 스타일 지원이 필요하지 않은 경우 [publish\(\)](#) 메서드를 비워두면 됩니다. 마찬가지로 이벤트 기반 통신을 지원할 필요가 없는 경우 [dispatchEvent\(\)](#) 메서드를 건너뛰면 됩니다.

이러한 메서드가 언제, 무엇을 실행하는지 관찰하기 위해 다음과 같이 여러 개의 [console.log](#) 호출을 추가해 보겠습니다:

```
GoogleCloudPubSubClient 클래스 ClientProxy 확장 {
    async connect(): Promise<any> {
        콘솔 로그('연결');
    }

    async close() {
        console.log('close');
    }

    async dispatchEvent(packet: ReadPacket<any>): Promise<any> {
        return console.log('event to dispatch: ', packet);
    }
}

게시(
    패킷을 읽습니다: ReadPacket<any>,
    콜백: (패킷: WritePacket<any>) => void,
): 함수 { console.log('message:', packet);

// 실제 애플리케이션에서 "콜백" 함수는 다음과 같아야 합니다.
```

실행됨

```
// 응답자로부터 페이로드를 전송합니다. 여기서는 간단히 시뮬레이션(5초 지연)해 보겠습니다.

// 원래 전달한 것과 동일한 "데이터"를 전달하여 응답을 보냈습니다.
setTimeout(() => callback({ response: packet.data }), 5000);

반환 () => 콘솔.로그('teardown');

}

}
```

이제 `GoogleCloudPubSubClient` 클래스의 인스턴스를 생성하고 `send()` 함수를 실행해 보겠습니다. 메서드(이전 장에서 보셨을 것입니다)를 사용하여 반환된 관찰 가능한 스트림을 구독합니다.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .subscribe((응답) => console.log(응답));
```

이제 터미널에 다음과 같은 출력이 표시됩니다:

연결

```
메시지: { pattern: 'pattern', data: 'Hello world!' }
Hello world! // <-- 5초 후
```

`제시()` 메서드가 반환하는 "teardown" 메서드가 제대로 실행되는지 테스트하기 위해 스트림에 타임아웃 연산자를 적용하고 2초로 설정하여 `setTimeout`이 `콜백` 함수를 호출하는 것보다 일찍 종료되도록 해 보겠습니다.

```
const googlePubSubClient = new GoogleCloudPubSubClient();
googlePubSubClient
  .send('pattern', 'Hello world!')
  .pipe(timeout(2000))
  .subscribe(
    (응답) => 콘솔.로그(응답), (오류) => 콘솔.오류(오류.메시지),
  );
```

정보 힌트 타임아웃 연산자는 `rxjs/operators` 패키지에서 가져옵니다.

타임아웃 연산자를 적용하면 터미널 출력은 다음과 같이 표시됩니다:

연결

```
메시지: { pattern: 'pattern', data: 'Hello world!' }
```

```
해체 // <-- 해체 시간 초과가 발생
```

했습니다.

메시지를 보내는 대신 이벤트를 디스패치하려면 `emit()` 메서드를 사용합니다:

```
googlePubSubClient.emit('event', 'Hello world!');
```

이것이 바로 콘솔에 표시되는 내용입니다:

연결

```
event to dispatch: { pattern: 'event', data: 'Hello world!' }
```

메시지 직렬화

클라이언트 측에서 응답 직렬화에 대한 사용자 정의 로직을 추가해야 하는 경우 `ClientProxy` 클래스 또는 그 하위 클래스 중 하나를 확장하는 사용자 정의 클래스를 사용할 수 있습니다. 성공적인 요청을 수정하려면

`serializeResponse` 메서드를 재정의하고, 이 클라이언트를 통과하는 모든 오류를 수정하려면

`serializeError` 메서드를 재정의할 수 있습니다. 이 커스텀 클래스를 사용하려면 `customClass` 속성을 사용하여 클래스 자체를 `ClientsModule.register()` 메서드에 전달하면 됩니다. 아래는 각 오류를 `RpcException`으로 직렬화하는 커스텀 `ClientProxy`의 예시입니다.

`@@파일명`(오류처리.프록시)

```
'@nestjs/microservices'에서 { ClientTcp, RpcException }을 가져옵니다;
```

```
class ErrorHandlingProxy extends ClientTCP {
  serializeError(err: 오류) {
    새로운 RpcException(err)을 반환합니다;
  }
}
```

를 생성한 다음 `ClientsModule`에서 다음과 같이 사용합니다:

@@파일명 (app.module)

```
@Module({  
    imports: [  
        ClientsModule.register({  
            name: 'CustomProxy',  
            customClass: 오류처리 프록시,  
        }),  
    ],  
})
```

AppModule 클래스 내보내기

정보 힌트 이것은 클래스의 인스턴스가 아니라 `customClass`에 전달되는 클래스 자체입니다. Nest는 사용자를 위해 내부적으로 인스턴스를 생성하고 옵션 프로퍼티에 지정된 모든 옵션을 새 `ClientProxy`에 전달합니다.

예외 필터

HTTP 예외 필터 계층과 해당 마이크로서비스 계층의 유일한 차이점은 `HttpException`을 던지는 대신 `RpcException`을 사용해야 한다는 점입니다.

새로운 `RpcException`('잘못된 자격 증명.')을 던집니다;

정보 힌트 `RpcException` 클래스는 `@nestjs/microservices` 패키지에서 가져옵니다.

위의 샘플을 사용하면 Nest가 던져진 예외를 처리하고 다음과 같은 구조의 오류 객체를 반환합니다:

```
{  
  "상태": "오류",  
  "메시지": "잘못된 자격 증명입니다."  
}
```

필터

마이크로서비스 예외 필터는 HTTP 예외 필터와 유사하게 작동하지만 한 가지 작은 차이가 있습니다. 마이크로서비스 예외 필터의

`catch()` 메서드는 `옵저버블`을 반환해야 합니다.

@@파일명 (rpc-exception.filter)

'@nestjs/common'에서 { Catch, RpcExceptionFilter, ArgumentsHost }를 임포트하고,
'rxjs'에서 { Observable, throwError }를 임포트합니다;
'@nestjs/microservices'에서 { RpcException }을 가져옵니다;

@Catch(RpcException)

```
export class ExceptionFilter 구현 RpcExceptionFilter<RpcException> {
    catch(exception: RpcException, host: ArgumentsHost): Observable<any> {
        반환 throwError(() => exception.getError());
    }
}
```

@@switch

'@nestjs/common'에서 { Catch }를 임포트하

고, 'rxjs'에서 { throwError }를 임포트합니

다;

@Catch(RpcException)

```
export class ExceptionFilter {
    catch(exception, host) {
        반환 throwError(() => exception.getError());
    }
}
```

경고 [하이브리드 애플리케이션을](#) 사용할 때 글로벌 마이크로서비스 예외 필터는 기본적으로 사용하도록 설정되어 있지 않습니다.

다음 예제는 수동으로 인스턴스화된 메서드 범위 필터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 필터를 사용할 수도 있습니다(즉, 컨트롤러 클래스 앞에 `@UseFilters()` 데코레이터를 붙이면 됩니다).

```
@@파일명()  
사용필터(새로운 예외필터()) 메시지 패턴({ cmd:  
    '합계' }) 누적(데이터: 숫자[]): 숫자 {  
    반환 (데이터 || []).reduce((a, b) => a + b);  
}  
@@switch  
사용필터(새로운 예외필터()) 메시지패턴({ cmd:  
    '합계' }) 누적(데이터) {  
    반환 (데이터 || []).reduce((a, b) => a + b);  
}
```

상속

일반적으로 애플리케이션 요구 사항을 충족하도록 완전히 사용자 정의된 예외 필터를 만듭니다. 그러나 핵심 예외 필터를 단순히 확장하고 특정 요인에 따라 동작을 재정의하려는 사용 사례가 있을 수 있습니다.

예외 처리를 기본 필터에 위임하려면 `BaseExceptionFilter`를 확장해야 합니다.

를 생성하고 상속된 `catch()` 메서드를 호출합니다.

@@파일명()

```
'@nestjs/common'에서 { Catch, ArgumentsHost }를 가져옵니다;  
'@nestjs/microservices'에서 { BaseRpcExceptionFilter }를 가져옵니다;  
  
@Catch()  
export class AllExceptionsFilter extends BaseRpcExceptionFilter {  
  catch(exception: any, host: ArgumentsHost) {  
    반환 super.catch(예외, 호스트);  
  }  
}  
@@switch  
'@nestjs/common'에서 { Catch }를 가져옵니다;  
'@nestjs/microservices'에서 { BaseRpcExceptionFilter }를 가져옵니다;  
  
@Catch()  
export class AllExceptionsFilter extends BaseRpcExceptionFilter {  
  catch(exception, host) {  
    반환 super.catch(예외, 호스트);  
  }  
}
```

위의 구현은 접근 방식을 보여주는 셀일 뿐입니다. 확장 예외 필터의 구현에는 맞춤형 비즈니스 로직(예: 다양한 조건 처리)이 포함될 수 있습니다.

파이프

일반 파이프와 마이크로서비스 파이프 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException` 을 던지는 대신 `RpcException`을 사용해야 한다는 것입니다.

정보 힌트 `RpcException` 클래스는 `@nestjs/microservices` 패키지에서 노출됩니다.

바인딩 파이프

다음 예제는 수동으로 인스턴스화된 메서드 범위 파이프를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 파이프를 사용할 수도 있습니다(즉, 컨트롤러 클래스의 접두사에 `@UsePipes()` 데코레이터).

```
@@파일명()
@UsePipes(new ValidationPipe()) 메시지 패
턴({ cmd: '합계' }) 누적(데이터: 숫자[]) : 숫
자 {
    반환 (데이터 || []).reduce((a, b) => a + b);
}
@@switch
@UsePipes(new ValidationPipe())
메시지패턴({ cmd: '합계' }) 누적(데
이터) {
    반환 (데이터 || []).reduce((a, b) => a + b);
}
```

경비병

마이크로서비스 가드와 일반 HTTP 애플리케이션 가드 사이에는 근본적인 차이가 없습니다. 유일한 차이점은 `HttpException`을 던지는 대신 `RpcException`을 사용해야 한다는 점입니다.

정보 힌트 `RpcException` 클래스는 `@nestjs/microservices` 패키지에서 노출됩니다.

바인딩 가드

다음 예제는 메서드 범위 가드를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 가드를 사용할 수도 있습니다(즉, 컨트롤러 클래스에 `@UseGuards()` 데코레이터를 앞에 붙이면 됩니다).

```
@@파일명() @사용가드(AuthGuard) @메
시지패턴({ cmd: '합계' })
accumulate(data: number[]): number {
    반환 (데이터 || []).reduce((a, b) => a + b);
}

@@스위치 @사용가드
(AuthGuard)
메시지 패턴({ cmd: '합계' }) 누적(데이터) {
    반환 (데이터 || []).reduce((a, b) => a + b);
}
```

인터셉터

일반 인터셉터와 마이크로서비스 인터셉터 간에는 차이가 없습니다. 다음 예제에서는 수동으로 인스턴스화된 메서드 범위 인터셉터를 사용합니다. HTTP 기반 애플리케이션과 마찬가지로 컨트롤러 범위 인터셉터도 사용할 수 있습니다(즉, 컨트롤러 클래스 앞에 `@UseInterceptors()` 데코레이터).

`@@파일명()`

```
사용 인터셉터(새로운 트랜스폼인터셉터())
@MessagePattern({ cmd: 'sum' })
accumulate(data: number[]): number {
    반환 (데이터 || []).reduce((a, b) => a + b);
}
@@switch
사용 인터셉터(새로운 트랜스폼인터셉터()) 메시지패턴({
cmd: '합계' }) 누적(데이터) {
    반환 (데이터 || []).reduce((a, b) => a + b);
}
```

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 [npm](#)을 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 [네스트](#)를 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 <code>package.json</code>	인스턴스 분리	모노레포에서 공유
기본 컴파일러	<code>tsc</code>	웹팩
컴파일러 설정	별도로 지정	모노레포 기본값은 다음과 같습니다. 프로젝트별로 재정의
다음과 같은 구성 파일	<code>.prettierrc</code> 등	
<code>.eslintrc.js</code> ,		

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

어떤 모드가 가장 적합한지 결정하는 데 도움이 되는 자세한 정보는 [작업 공간](#) 및 [라이브러리 섹션](#)을 참조하세요.

CLI 명령 구문

모든 네스트 명령은 동일한 형식을 따릅니다:

```
nest commandOrAlias 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-run*에는 이에 상응하는 약식 *-d*가 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



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 [타입 스크립트 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:

액 내-

액 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
```

액 생성 스키마는 각 애플리케이션 프로젝트를 액 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 사용`됩니다.

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 `tests` 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				
tsConfigPath		부울	deleteOutDir			부울

tsConfigPath	문자열	자산	배열
--------------	-----	----	----

설명

true이면 웹팩 컴파일러를 사용합니다. false이거나 존재하지 않으면 tsc를 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 ./dist이며 tsconfig.json에 구성된 대로).

컴파일 단계가 시작될 때마다 타입스크립트가 아닌 에셋을 자동으로 배포하도록 설정합니다(--watch 모드의 종분 컴파일에서 는 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 섹션 을 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 옵션이라는 두 가지 속성이 포함된 객체를 전달하면 됩니다.
typeCheck	부울	true이면 SWC 기반 프로젝트에 대해 유형 검사를 활성화합니다(빌더는 SWC입니다). 기본값은 false 입니다.

글로벌 생성 옵션

이러한 속성은 [동지 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: 등지 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 `false`로 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_modules` 번들링하도록 웹팩에 지시하려면 `webpack.config.js`에 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; `true`이면 지정된 에셋을 감시하는 감시 모드로 실행합니다

예를 들어:

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

경고 경고 최상위 컴파일러 옵션 프로퍼티에서 `watchAssets`를 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 모듈리식 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 라이브러리**라는 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:

```
[REDACTED]
```

```
nest build my-library
```

라이브러리 사용

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

먼저, 라이브러리 모듈을 사용하는 것은 다른 Nest 모듈을 사용하는 것과 동일하다는 점에 유의하세요. 모노레포 가 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 제공 한 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.

`MyLibraryModule`을 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. [여기](#)에 설명된 대로 `tsc`로 전환할 수도 있습니다.

CLI 명령 참조

등지 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
--dry-run	변경 사항을 보고하지만 파일 시스템을 변경하지는 않습니다.
--skip-git	--언어 [언어] --컬렉션 [컬렉션 이름]
--skip-install	--컬렉션 [컬렉션 이름]
--package-prefix	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

프로그래밍 언어(`TS` 또는 `JS`)를 지정합니다. 별칭

토

: `-l`

리

회로도 컬렉션을 지정합니다. 회로도가 포함된 설치된 `npm` 패키지의 패키지 이름을 사용합니다.

초

별칭: `-c`

옵션

설명

다음 TypeScript 컴파일러 플래그를 활성화하여 프로젝트를 시작하세요:

--엄격한

`strictNullChecks`, `noImplicitAny`, `strictBindCallApply`,
`forceConsistentCasingInFileNames`, `noFallthroughCasesInSwitch`.

동지 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
--dry-run	변경된 내용은 보고하지만 파일 시스템에 영향을 미치지 않습니다.
--driven-by	별칭: <code>-d</code>
--project [프로젝트]	요소를 추가해야 하는 프로젝트입니다. 별칭: <code>-p</code>
--flat	요소에 대한 폴더를 생성하지 않습니다.
--collection [컬렉션 이름]	회로도 컬렉션을 지정합니다. 설치된 npm의 패키지 이름 사용 도식이 포함된 패키지입니다. 별칭: <code>-c</code>
--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-paths`를 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	<code>tsconfig</code> 파일의 경로입
[경로]	니다. 별칭 <code>-p</code>
--config	<code>nest-cli</code> 구성 파일의 경로입니다. 별칭
[경로]	<code>-C</code>
	시계 모드에서 실행(실시간 새로 고침)합니다.
--시계	컴파일에 <code>tsc</code> 을 사용하는 경우 <code>rs</code> 를 입력하여 애플리케이션을 다시 시작할 수 있습니다(수동 재시작 옵션이 <code>true</code> 로 설정됨). 별칭 -
--빌더 [이 름]	<code>w</code> 컴파일에 사용할 빌더를 지정합니다(<code>tsc</code> , <code>swc</code> 또는 웹팩). 별칭 <code>-b</code>
--webpack	컴파일에 웹팩을 사용합니다(더 이상 사용되지 않음: 대신 <code>--builder 웹팩</code> 사용).
--	
웹팩 경로	웹팩 구성 경로입니다.
--tsc	컴파일에 <code>tsc</code> 을 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]	<code>tsconfig</code> 파일의 경로입니다. 별칭 <code>-p</code>
--config [경로]	<code>nest-cli</code> 구성 파일의 경로입니다. 별칭 <code>-c</code>
--시계	시계 모드에서 실행(실시간 다시 로드) 별칭 <code>-w</code>

옵션	설명
--빌더 [이름]	컴파일에 사용할 빌더를 지정합니다(tsc , swc 또는 웹팩). 별칭 -b
--보존워치 출력	화면을 지우는 대신 오래된 콘솔 출력력을 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 사용합 니다. 실행할 바이너리(기본 값: 노드). 별칭 -e
--exec [바이너리]	

등지 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

등지 정보

A complex musical score consisting of six staves. Each staff contains multiple note heads and stems, indicating a dense and intricate musical composition. The notes vary in size and position, creating a visual representation of a multi-layered musical piece.

_ _ _ _ / _ | _ / _ _ _ _ / _ _ _ _ /

[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 build`) 및 실행(`nest start`) 하위 명령을 실행하는 것이 좋습니다(`nest new`를 실행하는 대신 리포지토리를 복제하여 시작하려는 경우 [타입스크립트 스타터](#)를 참조하세요).

빌드

`nest build`은 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`을 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 [동지 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.json**에서 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 스타터](#)를 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 시작**을 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.

マイグレーション

변경할 필요는 없지만, [tsc-watch](#) 또는 [ts-node](#)와 같은 도구를 사용하는 대신 새 CLI 명령을 사용하도록 마이그레이션할 수 있습니다. 이 경우, 글로벌 및 로컬 모두에서 최신 버전의 [@nestjs/cli](#)를 설치하기만 하면 됩니다

:

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

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

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

소개

OpenAPI 사양은 언어에 구애받지 않는 정의 형식으로 RESTful API를 설명하는 데 사용됩니다. Nest는 데코레이터를 활용하여 이러한 사양을 생성할 수 있는 전용 [모듈을](#) 제공합니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
npm install --save @nestjs/swagger
```

부트스트랩

설치 프로세스가 완료되면 `main.ts` 파일을 열고 다음을 사용하여 Swagger를 초기화합니다.

`SwaggerModule` 클래스입니다:

```
@@파일명(메인)

'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/swagger'에서 { SwaggerModule, DocumentBuilder }를 임포트하고,
'./app.module'에서 { AppModule }을 임포트합니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule);

  const config = new DocumentBuilder()
    .setTitle('고양이 예시')
    .setDescription('고양이 API 설명')
    .setVersion('1.0')
    .addTag('cats')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

  await app.listen(3000);
}
```

정보 힌트 문서(`SwaggerModule#createDocument()` 메서드에서 반환)는 [OpenAPI Document](#)을 부트스트랩();

준수하는 직렬화 가능한 객체입니다. HTTP를 통해 호스팅하는 대신 JSON/YAML 파일로 저장하여 다양

한 방식으로 사용할 수도 있습니다.

문서 작성기는 OpenAPI 사양을 준수하는 기본 문서를 구조화하는 데 도움이 됩니다. 제목, 설명, 버전 등과 같은 속성을 설정할 수 있는 여러 메서드를 제공합니다. 전체 문서(모든 HTTP 경로가 정의된)를 생성하려면 `SwaggerModule` 클래스의 `createDocument()` 메서드를 사용합니다. 이 메서드는 애플리케이션 인스턴스와 Swagger 옵션이라는 두 개의 인수를 받습니다.

객체입니다. 또는 세 번째 인수를 제공할 수 있는데, 이 인수의 유형은 [Swagger도큐먼트옵션](#). 이에 대한 자세한 내용은 [문서 옵션 섹션을 참조하세요](#). 문서

서를 생성하고 나면 `setup()` 메서드를 호출할 수 있습니다. 수락합니다:

- . 스웨거 UI를 마운트하는 경로
- . 애플리케이션 인스턴스
- . 위에 인스턴스화된 문서 객체
- . 선택적 구성 매개변수(자세한 내용은 [여기](#)를 참조하세요)

이제 다음 명령을 실행하여 HTTP 서버를 시작할 수 있습니다:

```
$ npm 실행 시작
```

애플리케이션이 실행되는 동안 브라우저를 열고 <http://localhost:3000/api> 으로 이동합니다. Swagger UI 가 표시됩니다.



보시다시피 [SwaggerModule](#)은 모든 엔드포인트를 자동으로 반영합니다.

정보 힌트 Swagger JSON 파일을 생성하고 다운로드하려면 <http://localhost:3000/api-json>(<http://localhost:3000/api>)로 이동합니다(Swagger 문서가 제공된다고 가정).

경고 패스트파이와 헬멧을 사용할 때 [CSP](#)에 문제가 있을 수 있으며, 이 충돌을 해결하려면 아래 그림과 같이 CSP를 구성하세요:

```
app.register(helmet, {
  contentSecurityPolicy: {
    지시어를 추가합니다: {
      defaultSrc: [`'self'`],
      styleSrc: [`'self'`, `unsafe-inline`],
      imgSrc: [`'self'`, 'data:', 'validator.swagger.io'],
      scriptSrc: [`'self'`, `https: 'unsafe-inline'`],
    },
  },
});

// CSP를 전혀 사용하지 않을 경우 다음과 같이 사용할 수 있습니다:
app.register(helmet, {
  contentSecurityPolicy: false,
});
```

문서 옵션

문서를 만들 때 라이브러리의 동작을 미세 조정하기 위해 몇 가지 추가 옵션을 제공할 수 있습니다. 이러한 옵션은 다음과 같은 [SwaggerDocumentOptions](#) 유형이어야 합니다:

```
내보내기 인터페이스 SwaggerDocumentOptions {
    /**
     * 사양에 포함할 모듈 목록
     */
    include? 함수[];

    /**
     * 검사하고 사양에 포함시켜야 하는 추가 추가 모델
     */
    extraModels?: Function[];

    /**
     * true`이면 스웨거는 글로벌 접두사 설정된
     `setGlobalPrefix()` 메서드
     */
    무시 글로벌 접두사?: 부울;

    /**
     * true`인 경우, 스웨거는 다음에서 가져온 모듈의 경로도 로드합니다.
     포함` 모듈
     */
    deepScanRoutes?: 부울;

    /**
     * 커스텀 오퍼레이션아이디팩토리를 생성하는 데 사용되는
     `운영아이디`
     * 컨트롤러 키`와 `메소드 키`를 기반으로 합니다.
     * 기본값 () => 컨트롤러키_메소드키
     */
    operationIdFactory?: (컨트롤러키: 문자열, 메소드키: 문자열) => 문자열;
}
```

예를 들어 라이브러리에서 `UserController_createUser` 대신 `createUser`와 같은 작업 이름을 생성하도록 하려면 다음을 설정하면 됩니다:

```
const 옵션: SwaggerDocumentOptions ={
    operationIdFactory: (
        컨트롤러키: 문자열, 메서드키:
        문자열
    ) => 메서드키
};

const document = SwaggerModule.createDocument(app, config, options);
```

설정 옵션

SwaggerModule#setup 메서드의 네 번째 인수로 ExpressSwaggerCustomOptions(express를 사용하는 경우) 인터페이스를 충족하는 옵션 오브젝트를 전달하여 Swagger UI를 구성할 수 있습니다.

```
내보내기 인터페이스 ExpressSwaggerCustomOptions { 탐색기
  ?: 부울;
  swaggerOptions?: Record<string, any>;
  customCss?: 문자열;
  customCssUrl?: 문자열;
  customJs?: 문자열;
  customfavicon?: 문자열;
  swaggerUrl?: 문자열;
  customSiteTitle?: 문자열;
  validatorUrl?: 문자열; url?:
  문자열입니다;
  urls?: Record<'url' | 'name', string>[];
  patchDocumentOnRequest?: <TRequest = any, TResponse = any> (req:
  TRequest, res: TResponse, 문서: OpenAPIObject) => OpenAPIObject;
}
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

유형 및 매개변수

SwaggerModule은 라우트 핸들러에서 모든 `@Body()`, `@Query()`, `@Param()` 데코레이터를 검색하여 API 문서를 생성합니다. 또한 리플렉션을 활용하여 해당 모델 정의를 생성합니다. 다음 코드를 살펴보세요:

```
@Post()  
async create(@Body() createCatDto: CreateCatDto) {  
  this.catsService.create(createCatDto);  
}
```

정보 힌트 본문 정의를 명시적으로 설정하려면 `@ApiBody()` 데코레이터를 사용하십시오 (`nestjs/swagger` 패키지).

`CreateCatDto`를 기반으로 다음과 같은 모델 정의 Swagger UI가 생성됩니다:



보시다시피 클래스에 몇 가지 선언된 프로퍼티가 있지만 정의는 비어 있습니다. 클래스 프로퍼티를 SwaggerModule에 표시하려면 `@ApiProperty()` 데코레이터로 주석을 달거나 자동으로 처리하는 CLI 플러그인(플러그인 섹션에서 자세히 읽어보세요)을 사용해야 합니다:

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기  
  
클래스 CreateCatDto {  
  @ApiProperty()  
  이름: 문자열;  
  
  @ApiProperty()  
  age: 숫자;  
  
  @ApiProperty()  
  품종: 문자열;  
}
```

정보 힌트 각 프로퍼티에 수동으로 주석을 달는 대신 이를 자동으로 제공하는 Swagger 플러그인([플러그인](#) 섹션 참조)을 사용하는 것이 좋습니다.

브라우저를 열고 생성된 `CreateCatDto` 모델을 확인해 보겠습니다:



또한 `@ApiProperty()` 데코레이터를 사용하면 다양한 **스키마 객체** 속성을 설정할 수 있습니다:

```
@ApiProperty({  
    설명: '고양이의 나이',
```

```
최소: 1,  
기본값: 1,  
}  
나이: 숫자;
```

정보 힌트 `{} "@ApiProperty({ 필수: false })"`을 명시적으로 입력하는 대신

`@ApiPropertyOptional()` 단축 데코레이터를 사용할 수 있습니다.

속성의 유형을 명시적으로 설정하려면 `유형` 키를 사용합니다:

```
@ApiProperty({  
    type: 숫자,  
})  
나이: 숫자;
```

배열

프로퍼티가 배열인 경우 아래와 같이 배열 유형을 수동으로 표시해야 합니다:

```
@ApiProperty({ 유형: [문자열] }) names:  
문자열[];
```

정보 힌트 배열을 자동으로 감지하는 Swagger 플러그인([플러그인](#) 섹션 참조)을 사용하는 것이 좋습니다.

위와 같이 유형을 배열의 첫 번째 요소로 포함하거나(위 그림 참조), `isArray` 속성을 다음과 같이 설정합니다.
`true`.

순환 종속성

클래스 간에 순환 종속성이 있는 경우, 자연 함수를 사용하여 `SwaggerModule`을 제공하세요.

유형 정보와 함께:

```
@ApiProperty({ 유형: () => 노드 })  
노드: 노드;
```

정보 힌트 순환 종속성을 자동으로 감지하는 Swagger 플러그인([플러그인](#) 섹션 참조)을 사용하는 것이 좋습니다.

제네릭 및 인터페이스

TypeScript는 제네릭이나 인터페이스에 대한 메타데이터를 저장하지 않으므로 DTO에서 제네릭이나 인터페이스를 사용할 때 SwaggerModule이 런타임에 모델 정의를 제대로 생성하지 못할 수 있습니다. 예를 들어, 다음 코드는 Swagger 모듈에서 올바르게 검사되지 않습니다:

```
createBulk(@Body() usersDto: CreateUserDto[])
```

이 제한을 극복하기 위해 유형을 명시적으로 설정할 수 있습니다:

```
APIBody({ type: [CreateUserDto] })
createBulk(@Body() usersDto: CreateUserDto[])
```

열거형

열거 형을 식별하려면 값 배열을 사용하여 **@ApiProperty**에서 열거 형 속성을 수동으로 설정해야 합니다.

```
@ApiProperty({ 열거형: ['관리자', '진행자', '사용자'] }) 역할:
UserRole;
```

또는 다음과 같이 실제 TypeScript 열거형을 정의합니다:

```
export enum UserRole {
  Admin = 'Admin',
  Moderator = 'Moderator',
  User = 'User',
}
```

그런 다음 **@Query()** 매개 변수 데코레이터와 함께 열거 형을 직접 사용할 수 있습니다.

@ApiQuery() 데코레이터.

```
@ApiQuery({ 이름: 'role', 열거형: UserRole })
async filterByRole(@Query('role') role: UserRole = UserRole.User) {}
```



isArray을 **true**로 설정하면 열거형을 다중 선택으로 선택할 수 있습니다:



열거형 스키마

기본적으로 열거형 속성은 **매개변수**에 열거형에 대한 원시 정의를 추가합니다.

- 품종:

유형: '문자열'

열거형 :

- 페르시아어
- Tabby
- Siamese

위의 사양은 대부분의 경우 정상적으로 작동합니다. 그러나 사양을 입력으로 받아 클라이언트 측 코드를 생성하는 도구를 사용하는 경우 생성된 코드에 중복된 열거 형이 포함되어 있는 문제가 발생할 수 있습니다. 다음 코드 스니펫을 살펴보세요:

```
// 생성된 클라이언트 측 코드 내보내기 클래

스 CatDetail {
    품종: CatDetailEnum;
}

내보내기 클래스 CatInformation {
    breed: CatInformationEnum;
}

export enum CatDetailEnum {
    페르시아어 = '페르시아어',
    타
    비 = '타비',
    샌어 = '샌어',
}

내보내기 열거형 CatInformationEnum {
    페르시아어 = '페르시아어',
}
```

정보 힌트 위의 스니펫은 NSwag라는 도구를 사용하여 생성되었습니다.

이제 완전히 동일한 열거형 두 개가 생겼음을 알 수 있습니다. 이 문제를 해결하려면
데코레이터의 enum 속성과 함께 enumName을 입력합니다.

```
내보내기 클래스 CatDetail {
    @ApiProperty({ enum: CatBreed, enumName: 'CatBreed' })
    breed: CatBreed;
}
```

enumName 속성을 사용하면 @nestjs/swagger가 CatBreed를 자체 스키마로 전환할 수 있으며, 이를 통해

CatBreed 열거형을 재사용할 수 있게 됩니다. 사양은 다음과 같습니다:

CatDetail: 유형:

'객체' 속성:

...

- 품종:

스키마:

참조: '#/components/schemas/CatBreed'

CatBreed: 유형:

문자열 열거형입니다

니다:

- 페르시아어
- Tabby
- Siamese

정보 힌트 enum을 속성으로 취하는 데코레이터는 enumName도 취합니다.

원시 정의

일부 특정 시나리오(예: 깊게 중첩된 배열, 행렬)에서는 유형을 직접 설명해야 할 수도 있습니다.

```
@ApiProperty({
  type: 'array',
  items: {
    유형: '배열', 항목: {
      유형: '숫자',
    },
  },
})
좌표: 숫자[][];
```

마찬가지로 컨트롤러 클래스에서 입력/출력 콘텐츠를 수동으로 정의하려면 스키마를 사용하세요.

속성입니다:

```
@ApiBody({
  schema: {
    유형: '배열', 항목: {
      유형: '배열', 항목:
      {
        유형: '숫자',
      },
    },
  },
})
async create(@Body() coords: number[][][]){}
```

추가 모델

컨트롤러에서 직접 참조되지는 않지만 Swagger 모듈에서 검사해야 하는 추가 모델을 정의하려면

`@ApiExtraModels()` 데코레이터를 사용합니다:

```
@ApiExtraModels(ExtraModel)
내보내기 클래스 CreateCatDto {}
```

정보 힌트 특정 모델 클래스에 대해 `@ApiExtraModels()`를 한 번만 사용하면 됩니다.

또는 `추가 모델` 속성이 지정된 옵션 객체를

`SwaggerModule#createDocument()` 메서드를 다음과 같이 호출합니다:

```
const document = SwaggerModule.createDocument(app, options, {
  extraModels: [ExtraModel],
});
```

모델에 대한 참조(`$ref`)를 가져오려면 `getSchemaPath(ExtraModel)` 함수를 사용합니다:

```
'application/vnd.api+json': {
  스키마를 추가합니다: { $ref: getSchemaPath(ExtraModel) },
},
```

`oneOf`, `anyOf`, `allOf`

스키마를 결합하려면 `oneOf`, `anyOf` 또는 `allOf` 키워드를 사용할 수 있습니다([자세히 보기](#)).

```
@ApiProperty({
  oneOf: [
    { $ref: getSchemaPath(Cat) },
    { $ref: getSchemaPath(Dog) },
  ],
})
반려동물: 고양이 | 개;
```

다형성 배열(즉, 멤버가 여러 스키마에 걸쳐 있는 배열)을 정의하려면 원시 정의(위 참조)를 사용하여 유형을 수동으로 정의해야 합니다.

유형 애완동물 = 고양이 | 개;

```
@ApiModelProperty({  
    유형: '배열', 항  
    목: {  
        oneOf: [  
            {$ref: getSchemaPath(Cat)},
```

```
{ $ref: getSchemaPath(Dog) } ,  
],  
},  
}  
}  
애완 동물: 애완동물[];
```

정보 힌트 `getSchemaPath()` 함수는 `@nestjs/swagger`에서 가져온 것입니다.

Cat과 Dog는 모두 (클래스 수준에서) `@ApiExtraModels()` 데코레이터를 사용하여 추가 모델로 정의해야 합니다.

운영

OpenAPI 용어에서 경로는 API가 노출하는 `/users` 또는 `/reports/summary`와 같은 엔드포인트(리소스)이며, 작업은 이러한 경로를 조작하는 데 사용되는 HTTP 메서드(예: `GET`, `POST` 또는 `DELETE`)입니다.

태그

특정 태그에 컨트롤러를 연결하려면 `@ApiTags(...태그)` 데코레이터를 사용합니다.

```
@ApiTags('cats')
@Controller('cats')
내보내기 클래스 CatsController {}
```

헤더

요청의 일부로 예상되는 사용자 정의 헤더를 정의하려면 `@ApiHeader()`를 사용합니다.

```
@ApiHeader({
    이름: 'X-MyHeader', 설명: '사용자
    정의 헤더',
})
@Controller('cats')
내보내기 클래스 CatsController {}
```

응답

사용자 정의 HTTP 응답을 정의하려면 `@ApiResponse()` 데코레이터를 사용합니다.

```
@Post()
 ApiResponse({ status: 201, description: '레코드가 성공적으로 생성되었습니다.' })
 APIResponse({ status: 403, description: 'Forbidden.' })
 async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
}
```

Nest는 `@ApiResponse`을 상속하는 짧은 API 응답 데코레이터 세트를 제공합니다.

데코레이터:

- ◆ `@ApiOkResponse()`
- ◆ `@ApiCreatedResponse()`
- ◆ `@ApiAcceptedResponse()`
- ◆ `@ApiNoContentResponse()`

- ◆ `APIMovedPermanentlyResponse()`
- ◆ `@ApiFoundResponse()`
- ◆ `@ApiBadRequestResponse()`
- ◆ `@ApiUnauthorizedResponse()`
- ◆ `@ApiNotFoundResponse()`
- ◆ `@ApiForbiddenResponse()`
- ◆ `@ApiMethodNotAllowedResponse()`
- ◆ `@ApiNotAcceptableResponse()`
- ◆ `@ApiRequestTimeoutResponse()`
- ◆ `@ApiConflictResponse()` `@ApiPre-`
- ◆ `conditionFailedResponse()`
- ◆ `@ApiTooManyRequestsResponse()`
- ◆ `@ApiGoneResponse()`
- ◆ `@ApiPayloadTooLargeResponse()`
- ◆ `@ApiUnsupportedMediaTypeResponse()`
- ◆ `@ApiUnprocessableEntityResponse()`
- ◆ `@ApiInternalServerErrorResponse()`
- ◆ `@ApiNotImplementedResponse()`
- ◆ `@ApiBadGatewayResponse()`
- ◆ `@ApiServiceUnavailableResponse()`
- ◆ `@ApiGatewayTimeoutResponse()`
- ◆ `@ApiDefaultResponse()`

```
@Post()
@ApiCreatedResponse({ description: '레코드가 성공적으로 생성되었습니다.' })
@APIForbiddenResponse({ description: 'Forbidden.' })
async create(@Body() createCatDto: CreateCatDto) {
    this.catsService.create(createCatDto);
}
```

요청에 대한 반환 모델을 지정하려면 클래스를 생성하고 모든 프로퍼티에

`@ApiProperty()` 데코레이터.

```
export class Cat {  
    @ApiProperty()  
    id: number;  
  
    @ApiProperty()  
    이름: 문자열;  
  
    @ApiProperty()  
    age: 숫자;  
  
    @ApiProperty()  
    품종: 문자열;  
}
```

그런 다음 `Cat` 모델을 응답 데코레이터의 `유형` 속성과 함께 사용할 수 있습니다.

```
@ApiTags('cats')
@Controller('cats')
export class CatsController {
  @Post()
  @ApiCreatedResponse({
    설명: '레코드가 성공적으로 생성되었습니다.', 유형: Cat,
  })
  async create(@Body() createCatDto: CreateCatDto): Promise<Cat> {
    return this.catsService.create(createCatDto);
  }
}
```

브라우저를 열고 생성된 `고양이` 모델을 확인해 보겠습니다:



파일 업로드

다음과 함께 `@ApiBody` 데코레이터를 사용하여 특정 방법에 대한 파일 업로드를 활성화할 수 있습니다.

를 호출합니다. 다음은 파일 업로드 기법을 사용한 전체 예제입니다:

```
사용 인터셉터(파일인터셉터('파일'))
@ApiConsumes('multipart/form-data')
@ApiBody({
  설명: '고양이 목록', 유형:
  FileUploadDto,
})
uploadFile(@UploadedFile() 파일) {}
```

여기서 `FileUploadDto`는 다음과 같이 정의됩니다:

```
FileUploadDto 클래스 {
  @ApiProperty({ 유형: '문자열', 형식: '바이너리' }) 파일:
  any;
}
```

여러 파일 업로드를 처리하려면 다음과 같이 `FilesUploadDto`를 정의할 수 있습니다:

```
클래스 FilesUploadDto {  
    @ApiModelProperty({ 유형: '배열', 항목: { 유형: '문자열', 형식: '바이너리'  
} })
```

```
파일: any[];  
}
```

확장 기능

요청에 확장자를 추가하려면 `@ApiExtension()` 데코레이터를 사용합니다. 확장자 이름 앞에는 `x-`를 붙여야 합니다.

```
@ApiExtension('x-foo', { hello: 'world' })
```

고급 일반 API 응답

원시 정의를 제공하는 기능을 통해 Swagger UI에 대한 일반 스키마를 정의할 수 있습니다. 다음과 같은 DTO가 있다고 가정합니다:

```
export class PaginatedDto<TData> {  
    @ApiModelProperty()  
    합계: 숫자;  
  
    @ApiModelProperty()  
    제한: 숫자;  
  
    @ApiModelProperty() 오프셋  
    : 숫자;  
  
    결과를 반환합니다: TData[];  
}
```

나중에 원시 정의를 제공할 것이므로 결과 꾸미기는 생략하겠습니다. 이제 다른 DTO를 정의하고 이름을 다음과 같이 CatDto로 지정해 보겠습니다:

```
export class CatDto
{ @ApiProperty()
  name: string;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

이를 통해 다음과 같이 `PaginatedDto<CatDto>` 응답을 정의할 수 있습니다:

```

@apiOkResponse({
  schema: {
    allOf: [
      { $ref: getSchemaPath(PaginatedDto) },
      {
        속성을 추가합니다: {
          results: {
            유형: '배열',
            항목을 반환합니다: { $ref: getSchemaPath(CatDto) },
          },
        },
      ],
    ],
  },
})

비동기 findAll(): Promise<PaginatedDto<CatDto>> {}

```

이 예제에서는 응답에 `allOf` `PaginatedDto`가 있고 `결과` 속성이 `Array<CatDto>` 유형이 되도록 지정합니다.

- 주어진 모델에 대한 OpenAPI 사양 파일 내에서 OpenAPI 스키마 경로를 반환하는 `getSchemaPath()` 함수를 호출합니다.
- `allOf`는 다양한 상속 관련 사용 사례를 다루기 위해 OAS 3에서 제공하는 개념입니다.

마지막으로, `PaginatedDto`는 컨트롤러에서 직접 참조하지 않기 때문에 `SwaggerModule`은 아직 해당 모델 정의를 생성할 수 없습니다. 이 경우 [추가 모델로](#) 추가해야 합니다. 예를 들어 다음과 같이 컨트롤러 수준에서 `@ApiExtraModels()` 데코레이터를 사용할 수 있습니다:

```

@Controller('cats')
@ApiExtraModels(PaginatedDto)
export class CatsController {}

```

지금 Swagger를 실행하면 이 특정 엔드포인트에 대해 생성된 `swagger.json`에 다음과 같은 응답이 정의되어 있어야 합니다:

```
"응답": {  
    "200": {  
        "설명": "", "내용": {  
            "application/json": {  
                "스키마": {  
                    "allOf": [  
                        {  
                            "$ref": "#/components/schemas/PaginatedDto"  
                        },  
                        {  
                            "properties": {  
                                "results": {  

```

재사용할 수 있도록 하기 위해 다음과 같이 `PaginatedDto`에 대한 사용자 지정 데코레이터를 만들 수 있습니다:

```
export const ApiPaginatedResponse = <TModel extends Type<any>>(
  model: TModel,
) => {
  return applyDecorators(
    ApiExtraModels(model),
    ApiOkResponse({
      스키마: {
        allOf: [
          { $ref: getSchemaPath(PaginatedDto) },
          {
            속성을 추가합니다: {
              results: {
                유형: '배열',
                항목을 반환합니다: { $ref: getSchemaPath(모델) },
              },
            },
          },
        ],
      },
    }),
  );
};
```

정보 힌트 유형<any> 인터페이스와 applyDecorators 함수는

nestis/common 패키지

SwaggerModule이 모델에 대한 정의를 생성하도록 하려면 앞서 컨트롤러의 `PaginatedDto`에서 한 것처럼 추가 모델로 추가해야 합니다.

이제 엔드포인트에서 사용자 정의 `@ApiPaginatedResponse()` 데코레이터를 사용할 수 있습니다:

@ApiPaginatedResponse(CatDto)

```
비동기 findAll(): Promise<PaginatedDto<CatDto>> {}
```

클라이언트 생성 도구의 경우, 이 접근 방식은 클라이언트에 대해 페이지 매김된 응답<TModel>이 생성되는 방식에 모호함을 야기합니다. 다음 코드조각은 위의 GET/엔드포인트에 대한 클라이언트 생성기 결과의 예시입니다.

```
// 각도  
findAll(): Observable<{ total: 숫자, limit: 숫자, offset: 숫자, results: CatDto[] }>
```

보시다시피 여기의 반환 유형이 모호합니다. 이 문제를 해결하려면 제목을 추가하면 됩니다.

속성을 ApiPaginatedResponse의 스키마에 추가합니다:

```
export const ApiPaginatedResponse = <TModel extends Type<any>>(model: TModel) => {  
    return applyDecorators(  
        ApiOkResponse({  
            스키마: {  
                title: `PaginatedResponseOf${model.name}`  
                allOf: [  
                    // ...  
                ],  
            },  
        }),  
    );  
};
```

이제 클라이언트 생성기 도구의 결과가 나타납니다:

```
// 각도  
findAll(): Observable<PaginatedResponseOfCatDto>
```

보안

특정 작업에 어떤 보안 메커니즘을 사용해야 하는지 정의하려면 `@ApiSecurity()`를 사용하십시오.
데코레이터.

```
@ApiSecurity('basic')
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

`DocumentBuilder`:

```
const options = new DocumentBuilder().addSecurity('basic', {
    type: 'http',
    체계: '기본',
});
```

가장 많이 사용되는 인증 기술 중 일부는 기본으로 제공되므로(예: [기본](#) 및 [무기명](#)) 위와 같이 보안 메커니즘을 수동으로 정의할 필요가 없습니다.

기본 인증

기본 인증을 사용하려면 `@ApiBasicAuth()`를 사용합니다.

```
@ApiBasicAuth()
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

`DocumentBuilder`:

```
const options = new DocumentBuilder().addBasicAuth();
```

무기명 인증

무기명 인증을 활성화하려면 `@ApiBearerAuth()`를 사용합니다.

```
APIBearerAuth()  
@Controller('cats')  
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

DocumentBuilder:

```
const options = new DocumentBuilder().addBearerAuth();
```

OAuth2 인증

OAuth2를 활성화하려면 `@ApiOAuth2()`를 사용합니다.

```
@ApiOAuth2(['pets:write'])
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

DocumentBuilder:

```
const options = new DocumentBuilder().addOAuth2();
```

쿠키 인증

쿠키 인증을 활성화하려면 `@ApiCookieAuth()`를 사용합니다.

```
API쿠키인증()
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

DocumentBuilder:

```
const options = new DocumentBuilder().addCookieAuth('optional-session-
id');
```

매핑된 유형

CRUD(생성/읽기/업데이트/삭제)와 같은 기능을 구축할 때 기본 엔티티 유형에서 변형을 만드는 것이 유용할 때가 많습니다. Nest는 유형 변환을 수행하는 여러 유ти리티 함수를 제공하여 이 작업을 더 편리하게 만듭니다.

부분

입력 유효성 검사 유형(DTO라고도 함)을 구축할 때 동일한 유형에 대해 만들기 및 업데이트 변형을 구축하는 것이 유용한 경우가 많습니다. 예를 들어, 만들기 변형은 모든 필드를 필수로 설정하고 업데이트 변형은 모든 필드를 선택 사항으로 설정할 수 있습니다.

Nest는 이 작업을 더 쉽게 수행하고 상용구를 최소화하기 위해 `PartialType()` 유ти리티 함수를 제공합니다.

`PartialType()` 함수는 입력 유형의 모든 속성이 선택 사항으로 설정된 유형(클래스)을 반환합니다. 예를 들어 다음과 같은 `create` 유형이 있다고 가정해 보겠습니다:

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기

클래스 CreateCatDto {
  @ApiProperty()
  이름: 문자열;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

기본적으로 이러한 필드는 모두 필수입니다. 필드는 동일하지만 각 필드가 선택 사항인 유형을 만들려면 클래스 참조(`CreateCatDto`)를 인수로 전달하는 `PartialType()`을 사용합니다:

```
export class UpdateCatDto extends PartialType(CreateCatDto) {} {}
```

정보 힌트 `PartialType()` 함수는 `@nestjs/swagger` 패키지에서 가져온 것입니다.

선택

PickType() 함수는 입력 유형에서 속성 집합을 선택하여 새 유형(클래스)을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기  
클래스 CreateCatDto {  
    @ApiProperty()  
}
```

```
이름: 문자열;  
  
@ApiProperty()  
age: 숫자;  
  
@ApiProperty()  
품종: 문자열;  
}
```

이 클래스에서 `PickType()` 유틸리티 함수를 사용하여 프로퍼티 집합을 선택할 수 있습니다:

```
export class UpdateCatAgeDto extends PickType(CreateCatDto, [ 'age' ] as  
const) {} {}
```

정보 힌트 `PickType()` 함수는 [@nestjs/swagger](#) 패키지에서 가져온 것입니다.

생략

`OmitType()` 함수는 입력 유형에서 모든 속성을 선택한 다음 특정 키 집합을 제거하여 유형을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기  
  
클래스 CreateCatDto {  
  @ApiProperty()  
  이름: 문자열;  
  
  @ApiProperty()  
  age: 숫자;  
  
  @ApiProperty()  
  품종: 문자열;  
}
```

아래와 같이 `이름`을 제외한 모든 프로퍼티를 가진 파생 유형을 생성할 수 있습니다. 이 구조체에서 `OmitType`의 두 번째 인수는 프로퍼티 이름의 배열입니다.

```
export class UpdateCatDto extends OmitType(CreateCatDto, [ 'name' ] as  
const) {} {}
```

정보 힌트 `OmitType()` 함수는 [@nestjs/swagger](#) 패키지에서 가져온 것입니다.

교차로

`IntersectionType()` 함수는 두 유형을 하나의 새로운 유형(클래스)으로 결합합니다. 예를 들어 다음과 같은 두 가지 유형으로 시작한다고 가정해 보겠습니다:

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기

클래스 CreateCatDto {
  @ApiProperty()
  이름: 문자열;

  @ApiProperty()
  품종: 문자열;
}

내보내기 클래스 AdditionalCatInfo {
  @ApiProperty()
  색상: 문자열;
}
```

두 유형의 모든 속성을 결합한 새로운 유형을 생성할 수 있습니다.

```
내보내기 클래스 UpdateCatDto extends IntersectionType(
  CreateCatDto,
  AdditionalCatInfo,
) {}
```

정보 힌트 `IntersectionType()` 함수는 `@nestjs/swagger` 패키지에서 가져온 것입니다.

구성

유형 매핑 유ти리티 함수는 컴포지션 가능합니다. 예를 들어 다음은 `이름을` 제외한 `CreateCatDto` 유형의 모든 속성을 가진 유형(클래스)을 생성하며, 해당 속성은 선택 사항으로 설정됩니다:

```
내보내기 클래스 UpdateCatDto extends PartialType(
  OmitType(CreateCatDto, [ 'name' ] as const),
) {}
```

데코레이터

사용 가능한 모든 OpenAPI 데코레이터에는 핵심 데코레이터와 구분하기 위해 API 접두사가 붙습니다. 아래는 내보낸 데코레이터의 전체 목록과 데코레이터가 적용될 수 있는 레벨을 지정한 것입니다.

@ApiBasicAuth()	메서드 / 컨트롤러
@ApiBearerAuth()	메서드 / 컨트롤러
@ApiBody()	메서드
@ApiConsumes()	메서드 / 컨트롤러
@ApiCookieAuth()	메서드/컨트롤러
@ApiExcludeController()	컨트롤러
@ApiExcludeEndpoint()	메서드
@ApiExtension()	메서드
@ApiExtraModels()	메서드 / 컨트롤러
@ApiHeader()	메서드 / 컨트롤러
@ApiHideProperty()	Model
@ApiOAuth2()	메서드 / 컨트롤러
@ApiOperation()	메서드
@ApiParam()	메서드
@ApiProduces()	메서드 / 컨트롤러
@ApiProperty()	모델
@ApiPropertyOptional()	모델
@ApiQuery()	메서드

`@ApiResponse()` 메서드 / 컨트롤러

`@ApiSecurity()` 메서드 / 컨트롤러

`@ApiTags()` 메서드 / 컨트롤러

CLI 플러그인

TypeScript의 메타데이터 반영 시스템에는 몇 가지 제약이 있어 클래스가 어떤 프로퍼티로 구성되어 있는지 확인하거나 특정 프로퍼티가 선택 사항인지 필수 사항인지 인식할 수 없습니다. 하지만 이러한 제약 조건 중 일부는 컴파일 시점에 해결할 수 있습니다. Nest는 필요한 상용구 코드의 양을 줄이기 위해 TypeScript 컴파일 프로세스를 개선하는 플러그인을 제공합니다.

정보 힌트 이 플러그인은 옵트인입니다. 원하는 경우 모든 데코레이터를 수동으로 선언하거나 필요한 곳에 특정 데코레이터만 선언할 수 있습니다.

개요

스웨거 플러그인이 자동으로 실행됩니다:

- annotate all DTO properties with `@ApiProperty` unless `@ApiHideProperty` is used
- 물음표에 따라 필요한 속성을 설정합니다(예: 이름?: 문자열이 설정됨).
필수: 거짓)
- 유형에 따라 유형 또는 열거형 속성을 설정합니다(배열도 지원)• 지정된 기본값에 따라 기본 속성을 설정합니다.
- 클래스 유효성 검사기 데코레이터를 기반으로 몇 가지 유효성 검사 규칙을 설정합니다(클래스 유효성 검사기 심이 true로 설정된 경우).
- 적절한 상태와 유형(응답 모델)으로 모든 엔드포인트에 응답 데코레이터를 추가합니다.
- 코멘트를 기반으로 프로퍼티 및 엔드포인트에 대한 설명을 생성합니다(introspectComments 참으로 설정)
- 코멘트를 기반으로 프로퍼티에 대한 예제 값을 생성합니다(introspectComments가 true로 설정된 경우).

파일 이름에는 다음 접미사 중 하나가 포함되어야 합니다: `['.dto.ts', '.entity.ts']`

(예: `create-user.dto.ts`)를 생성하여 플러그인에서 분석할 수 있도록 합니다.

다른 접미사를 사용하는 경우 플러그인의 동작을 조정하려면

`dtoFileNameSuffix` 옵션(아래 참조).

이전에는 Swagger UI로 인터랙티브한 경험을 제공하려면 모델/컴포넌트를 사양에 어떻게 선언해야 하는지 패키지에 알리기 위해 많은 코드를 복제해야 했습니다. 예를 들어 다음과 같이 간단한 `CreateUserDto` 클래스를 정

의할 수 있습니다:

```
export 클래스 CreateUserDto {  
    @ApiProperty()  
    이메일: 문자열;  
  
    @ApiProperty() 비밀번호  
    : 문자열;  
  
    @ApiProperty({ 열거형: RoleEnum, 기본값: [], isArray: true }) 역할:  
    RoleEnum[] = [];  
  
    @ApiProperty({ 필수: false, 기본값: true }) isEnabled?:  
    boolean = true;  
}
```

중간 규모의 프로젝트에서는 큰 문제가 되지 않지만, 클래스의 수가 많아지면 장황해지고 유지 관리가 어려워집니다.

Swagger 플러그인을 활성화하면 위의 클래스 정의를 간단하게 선언할 수 있습니다:

```
export 클래스 CreateUserData {  
    이메일: 문자열;  
    비밀번호: 문자열; 역할:  
    RoleEnum[] = [];  
    isEnabled?: boolean = true;  
}
```

정보 참고 Swagger 플러그인은 TypeScript 유형 및 클래스 유효성 검사기 데코레이터에서 `@ApiProperty()` 어노테이션을 파생합니다. 이는 생성된 Swagger UI 문서에 대한 API를 명확하게 설명하는 데 도움이 됩니다. 그러나 런타임에 유효성 검사는 여전히 클래스 유효성 검사기 데코레이터에 의해 처리됩니다. 따라서 `IsEmail()`, `IsNumber()` 등과 같은 유효성 검사기를 계속 사용해야 합니다.

따라서 문서 생성을 위해 자동 어노테이션에 의존하면서도 런타임 유효성 검사를 원하는 경우 클래스 유효성 검사기 데코레이터가 여전히 필요합니다.

정보 힌트 DTO에서 **매핑된 유형 유틸리티**(예: `PartialType`)를 사용할 때는 다음에서 가져옵니다.

플러그인이 스키마를 가져올 수 있도록 `@nestjs/mapped-types` 대신 `@nestjs/swagger`를 입력하세요.

플러그인은 추상 구문 트리를 기반으로 적절한 데코레이터를 즉시 추가합니다. 따라서 코드 곳곳에 흘어져 있는 `@ApiProperty` 데코레이터로 고생할 필요가 없습니다.

정보 힌트 플러그인은 누락된 스웨거 프로퍼티를 자동으로 생성하지만, 재정의해야 하는 경우 `@ApiProperty()`를 통해 명시적으로 설정하기만 하면 됩니다.

댓글 성찰

댓글 인트로스펙션 기능을 활성화하면 CLI 플러그인이 댓글을 기반으로 속성에 대한 설명과 예제 값을 생성합니다.

예를 들어 역할 속성을 예로 들어 보겠습니다:

```
/**  
 * 사용자 역할 목록  
 * @example ['admin']  
 */  
@ApiModelProperty({  
    설명: `사용자의 역할 목록`, 예시: ['admin'],  
})  
역할: RoleEnum[] = [];
```

설명과 예제 값을 모두 복제해야 합니다. `introspectComments` 를 활성화하면 CLI 플러그인이 이러한 코멘트를 추출하여 속성에 대한 설명(정의된 경우 예제)을 자동으로 제공할 수 있습니다. 이제 위의 프로퍼티는 다음과 같이 간단하게 선언할 수 있습니다:

```
/**  
 * 사용자 역할 목록  
 * @example ['admin']  
 */  
역할: RoleEnum[] = [];
```

플러그인에서 각각 `ApiProperty` 및 `ApiOperation` 데코레이터의 값을 설정하는 방법을 사용자 정의하는 데 사용할 수 있는 `dtoKeyOfComment` 및 `controllerKeyOfComment` 플러그인 옵션이 있습니다. 다음 예시를 살펴보세요:

```
일부 컨트롤러 내보내기 클래스 {  
    /**  
     * 리소스 생성  
     */  
    @Post()  
    create() {}  
}
```

기본적으로 이 옵션은 "설명"으로 설정되어 있습니다. 즉, 플러그인은 "일부 리소스 만들기"를 설명 키에 할당하여 `ApiOperation` 연산자의 설명 키로 사용합니다. 이렇게 설정합니다:

```
@ApiOperation({ description: "일부 리소스 생성" })
```

정보 힌트 모델의 경우 동일한 로직이 적용되지만 대신 `ApiProperty` 데코레이터에 적용됩니다.

CLI 플러그인 사용

플러그인을 활성화하려면 `nest-cli.json`(Nest CLI를 사용하는 경우)을 열고 다음 플러그인을 추가합니다. 구성합니다:

```
{  
  "collection": "@nestjs/schematics",  
  "sourceRoot": "src", "컴파일러옵션": {  
    "플러그인": ["@nestjs/swagger"]  
  }  
}
```

옵션 속성을 사용하여 플러그인의 동작을 사용자 정의할 수 있습니다.

```
"플러그인": [
  {
    "name": "@nestjs/swagger", "옵션": {
      "classValidatorShim": false,
      "introspectComments": true
    }
  }
]
```

옵션 속성은 다음 인터페이스를 충족해야 합니다:

```
export interface PluginOptions {
  dtoFileNameSuffix?: string[];
  controllerFileNameSuffix?: string[];
  classValidatorShim?: boolean;
  dtoKeyOfComment?: string;
  controllerKeyOfComment?: string;
  introspectComments?: boolean;
}
```

옵션	기본값	설명
dtoFileName 접미사	['.dto.ts', '.entity.ts']	DTO(데이터 전송 개체) 파일 접미사
컨트롤러 파일 이름 접미사 .controller.ts 컨트롤러 파일 접미사		
클래스 유효성 검사가 심	참	true로 설정하면 모듈은 클래스 유효성 검사 기 유효성 검사 데코레이터를 재사용합니다(예)
dtoKeyOfComment	'설명'	Max(10)은 스키마 정의에 최대: 10을 스키마 정의에 추가합니다)
컨트롤러 키 코멘트	'설명'	ApiProperty. 컨트롤러 키를 커짐으로 설정하는 속성 키입니다.
introspectComments	false	ApiOperation. true로 설정하면 플러그인이 댓글을 기반으로 속성에 대한 설명과 예제 값을 생성합니다.

플러그인 옵션이 업데이트될 때마다 `/dist` 폴더를 삭제하고 애플리케이션을 다시 빌드해야 합니다. CLI를 사용하지 않고 사용자 정의 웹팩을 구성한 경우 이 플러그인을 `ts-loader`와 함께 사용할 수 있습니다:

```
getCustomTransformers: (program: any) => ({{  
  before: [require('@nestjs/swagger/plugin').before({}, program)]  
}},
```

SWC 빌더

표준 설정(비모노레포)의 경우 SWC 빌더에서 CLI 플러그인을 사용하려면 [여기에](#) 설명된 대로 유형 검사를 사용하도록 설정해야 합니다.

```
nest start -b swc --type-check
```

모노레포 설정의 경우 [여기](#) 지침을 따르세요.

```
$ npx ts-node src/generate-metadata.ts  
# 또는 npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

이제 아래와 같이 직렬화된 메타데이터 파일을 [SwaggerModule#loadPluginMetadata](#) 메서드를 통해 로드해야 합니다:

```
'./metadata'에서 메타데이터 가져오기; // <-- "플러그인 메타데이터 생성기"에 의해 자동 생성된  
파일입니다.
```

```
await SwaggerModule.loadPluginMetadata(metadata); // <-- 여기  
const document = SwaggerModule.createDocument(app, config);
```

ts-jest와 통합(e2e 테스트)

[ts-jest](#)는 e2e 테스트를 실행하기 위해 소스 코드 파일을 메모리에서 즉석에서 컴파일합니다. 즉, Nest CLI 컴파일러를 사용하지 않으며 플러그인을 적용하거나 AST 변환을 수행하지 않습니다.

플러그인을 활성화하려면 e2e 테스트 디렉토리에 다음 파일을 생성합니다:

```
const transformer = require('@nestjs/swagger/plugin');

module.exports.name = 'nestjs-swagger-transformer';
// 아래 구성을 변경할 때마다 버전 번호를 변경해야 합니다. 그렇지 않으면 jest가 변경
사항을 감지하지 못합니다 module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
      // @nestjs/swagger/플러그인 옵션(비워둘 수 있음)
```

```

    },
    cs.program, // 이전 버전의 Jest의 경우 "cs.tsCompiler.program" (<=
v27)
);
}

```

이 설정이 완료되면 **jest** 구성 파일에서 AST 트랜스포머를 가져옵니다. 기본적으로(스타터 애플리케이션에서) **e2e** 테스트 구성 파일은 **테스트** 폴더 아래에 있으며 이름은 **jest-e2e.json**입니다.

```

{
  ... // 기타 구성 "전역": {
    "ts-jest": {
      "astTransformers": {
        "전에": ["<위에서 생성한 파일 경로>"]
      }
    }
  }
}

```

jest@^29를 사용하는 경우 이전 접근 방식이 더 이상 사용되지 않으므로 아래 스니펫을 사용하세요.

```

{
  ... // 기타 구성 "변환": [
    "^.+\\.\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "전에": ["<위에서 생성한 파일 경로>"]
        }
      }
    ]
  }
}

```

농담 문제 해결(e2e 테스트)

Jest가 설정 변경 사항을 적용하지 않는 것 같다면 이미 빌드 결과를 캐시한 것일 수 있습니다. 새 구성을 적용하려면 Jest의 캐시 딕렉터리를 지워야 합니다.

캐시 딕렉터리를 지우려면 NestJS 프로젝트 폴더에서 다음 명령을 실행합니다:

```
$ npx jest --clearCache
```

자동 캐시 지우기가 실패하는 경우에도 다음 명령을 사용하여 캐시 폴더를 수동으로 제거할 수 있습니다:

```
# 농담 캐시 디렉터리 찾기(보통 /tmp/jest_rs)
# NestJS 프로젝트 루트에서 다음 명령을 실행합니다.
$ npx jest --showConfig | grep cache
# ex 결과:
#   "캐시": true,
#   "캐시디렉토리": "/tmp/jest_rs"

# Jest 캐시 디렉터리 제거 또는 비우기
$ rm -rf <cacheDirectory값> #
```

예제:

```
# rm -rf /tmp/jest_rs
```

기타 기능

이 페이지에는 유용하게 사용할 수 있는 다른 모든 기능이 나열되어 있습니다. 글

로벌 접두사

`setGlobalPrefix()`를 통해 설정된 경로의 글로벌 접두사를 무시하려면 `ignoreGlobalPrefix`를 사용합니다:

```
const document = SwaggerModule.createDocument(app, options, {  
    ignoreGlobalPrefix: true,  
});
```

전역 매개 변수

문서 작성기를 사용하여 모든 경로에 매개변수 정의를 추가할 수 있습니다:

```
const options = new DocumentBuilder().addGlobalParameters({  
    name: 'tenantId',  
    in: '헤더',  
});
```

다양한 사양

SwaggerModule은 여러 사양을 지원하는 방법을 제공합니다. 즉, 서로 다른 엔드포인트에서 서로 다른 UI로 서로 다른 문서를 제공할 수 있습니다.

여러 사양을 지원하려면 애플리케이션을 모듈 방식으로 작성해야 합니다. `createDocument()` 메서드에는 `include`라는 속성을 가진 객체인 세 번째 인수 `extraOptions`가 사용됩니다. `include` 속성은 모듈의 배열인 값을 받습니다.

아래와 같이 여러 사양 지원을 설정할 수 있습니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다;  
'@nestjs/swagger'에서 { SwaggerModule, DocumentBuilder }를 임포트하고,  
'./app.module'에서 { AppModule }을 임포트합니다;  
'./cats/cats.module'에서 { CatsModule }을 임포트하고,  
'./dogs/dogs.module'에서 { DogsModule }을 임포트합니다;
```

비동기 함수 부트스트랩()

```
const app = await NestFactory.create(AppModule);  
  
/**  
 * createDocument(애플리케이션, 구성옵션, 추가옵션);  
 *  
 * createDocument 메서드는 선택적 세 번째 인자 "extraOptions"를 받습니다.  
 * 를 전달할 수 있는 "포함" 속성을 가진 객체입니다.
```

배열

- * 해당 스웨거 사양에 포함하려는 모듈의 개수
- * 예시 예를 들어, 캣츠모듈과 독스모듈에는 다음과 같은 두 개의 별도 스웨거 사양

이 있습니다.

* 는 두 개의 서로 다른 엔드포인트가 있는 두 개의 서로 다른 SwaggerUI에 노출 됩니다.

*/

```
const options = new DocumentBuilder()
    .setTitle('고양이 예시')
    .setDescription('고양이 API 설명')
    .setVersion('1.0')
    .addTag('cats')
    .build();

const catDocument = SwaggerModule.createDocument(app, options, {
    include: [CatsModule],
});
SwaggerModule.setup('api/cats', app, catDocument);

const secondOptions = new DocumentBuilder()
    .setTitle('개 예시')
    .setDescription('개 API 설명')
    .setVersion('1.0')
    .addTag('dogs')
    .build();

const dogDocument = SwaggerModule.createDocument(app, secondOptions, {
    include: [DogsModule],
});
SwaggerModule.setup('api/dogs', app, dogDocument);

await app.listen(3000);
}

부트스트랩();
```

이제 다음 명령으로 서버를 시작할 수 있습니다:

```
$ npm 실행 시작
```

<http://localhost:3000/api/cats> 으로 이동하여 고양이를 위한 Swagger UI를 확인하세요:



그러면 <http://localhost:3000/api/dogs> 에 반려견용 Swagger UI가 노출됩니다:



マイグレーション ガイド

현재 [@nestjs/swagger@3.*](#)를 사용 중이라면 버전 4.0의 다음과 같은 중단/API 변경 사항을 참고하세요. 주요

변경 사항

다음 데코레이터가 변경/명칭이 변경되었습니다:

- 이제 `@ApiModelProperty`는 [@ApiProperty](#)입니다.
- `@ApiModelPropertyOptional`은 이제 [@ApiPropertyOptional](#)
- `@ApiResponseModelProperty`는 이제 [@ApiImplicitQuery](#)는 이제 [@ApiImplicitQuery](#)로 변경되었습니다.
- `@ApilImplicitParam`은 이제 [@ApiParam](#)•
`@ApilImplicitBody`는 이제 [@ApiBody](#)입니다.
- `@ApilImplicitHeader`는 이제 [@ApiHeader](#)입니다.
- 이제 `@ApiOperation({{ '{' }} title: 'test' {{ '}' }})`은 `@ApiOperation({{ '{' }} summary: 'test' {{ '}' }})`
- 이제 `@ApiUseTags`는 [@ApiTags](#)입니다.

`DocumentBuilder` 브레이킹 변경 사항(메서드 서명 업데이트):

- 추가 태그
- `addBearerAuth`
- `addOAuth2`
- `setContactEmail`은 이제 [setContact](#)입니다.
- `setHost`가 제거되었습니다.
- `setSchemes`가 제거되었습니다(대신 `addServer('http://')`와 같은 추가 서버를 사용하세요).

(). 새로운 메서드

다음과 같은 방법이 추가되었습니다:

- ◆ 추가서버 추가Api
- ◆ 키 추가기본인증
- ◆ 추가보안
- ◆ 보안 요구 사항 추가

읽기-값-출력-루프(REPL)

REPL은 단일 사용자 입력을 받아 실행하고 결과를 사용자에게 반환하는 간단한 대화형 환경입니다. REPL 기능을 사용하면 터미널에서 직접 공급자(및 컨트롤러)의 의존성 그래프와 호출 메서드를 검사할 수 있습니다.

사용법

REPL 모드에서 NestJS 애플리케이션을 실행하려면 기존 `main.ts` 와 함께 새 `repl.ts` 파일을 생성합니다. 파일)을 열고 그 안에 다음 코드를 추가합니다:

@@파일명(답글)

```
'@nestjs/core'에서 { repl } 임포트;  
'./app.module'에서 { AppModule } 임포트;
```

비동기 함수 `bootstrap()` { await

```
    repl(AppModule);  
}
```

부트스트랩(); @@스

위치

```
'@nestjs/core'에서 { repl } 임포트;
```

```
'./app.module'에서 { AppModule } 임포트;
```

비동기 함수 `bootstrap()` { await

```
    repl(AppModule);  
}
```

부트스트랩();

이제 터미널에서 다음 명령어로 REPL을 시작합니다:

```
$ npm 실행 시작 -- --entryFile repl
```

정보 힌트 `repl`은 [Node.js REPL 서버](#) 객체를 반환합니다.

실행이 완료되면 콘솔에 다음 메시지가 표시됩니다:

```
LOG [NestFactory] Nest 애플리케이션을 시작하는 중...
```

```
LOG [InstanceLoader] 앱모듈 종속성 초기화됨 LOG REPL 초기화됨
```

이제 종속성 그래프와 상호작용을 시작할 수 있습니다. 예를 들어

AppService(여기서는 스타터 프로젝트를 예로 사용합니다)를 열고 `getHello()` 메서드를 호출합니다:

```
> get(AppService).getHello()  
'Hello World!'
```

예를 들어 터미널 내에서 자바스크립트 코드를 실행할 수 있습니다.

AppController를 로컬 변수로 설정하고 await을 사용하여 비동기 메서드를 호출합니다:

```
> appController = get(AppController)  
AppController { appService: AppService {} }  
> await appController.getHello()  
'Hello World!'
```

지정된 공급자 또는 컨트롤러에서 사용 가능한 모든 공개 메서드를 표시하려면 다음과 같이 methods() 함수를 사용합니다:

```
> 메소드(앱 컨트롤러) 메소드:  
□ getHello
```

등록된 모든 모듈을 컨트롤러 및 프로바이더와 함께 목록으로 인쇄하려면 debug()를 사용합니다.

```
> 디버그()  
  
앱모듈:  
- 컨트롤러:  
□ 앱 컨트롤러  
- 제공업체:  
□ 앱 서비스
```

빠른 데모:



미리 정의된 기존 네이티브 메서드에 대한 자세한 내용은 아래 섹션에서 확인할 수 있습니다. 네이티브 함

수

기본 제공 NestJS REPL에는 REPL을 시작할 때 전역적으로 사용할 수 있는 몇 가지 기본 함수가 포함되어 있습니다.

`help()`를 호출하여 나열할 수 있습니다.

함수의 시그니처(예: 예상 매개변수 및 반환 유형)가 무엇인지 기억나지 않는 경우 `<function_name>.help`를 호출할 수 있습니다. 예를 들어

```
> .help
인젝터블 또는 컨트롤러의 인스턴스를 검색하고, 그렇지 않으면 예외를 던집니다.
인터페이스: $(토큰: InjectionToken) => any
```

정보 힌트 이러한 함수 인터페이스는 [TypeScript 함수 유형 표현식 구문으로](#) 작성됩니다.

기능	설명	서명
debug	등록된 모든 모듈을 컨트롤러 및 공급자와 함께 목록으로 인쇄합니다.	debug(moduleCls?: ClassRef \ 문자열) => void
get 또	인젝터블 또는 컨트롤러의 인스턴스를 검색하고, 그렇지 않으면 예외를 던집니다.	get(token: InjectionToken) => any
는 \$ 메	지정된 공급자 또는 컨트롤러에서 사용 가능한 모든 공개 메소드를 표시합니다.	메서드(token: ClassRef \ 문자열) => void
서드	인젝터블 또는 컨트롤러의 일시적이거나 요청 범위가 지정된 인스턴스를 해결하고, 그렇지 않으면 예외를 던집니다.	resolve(token: InjectionToken, contextId: any) => Promise<any>
해결	예를 들어 모듈 트리를 탐색하여 선택한 모듈에서 특정 인스턴스를 가져올 수 있습니다.	select(token: DynamicModule \ ClassRef) => INestApplicationContext
선택		

시계 모드

개발 중에는 모든 코드 변경 사항을 자동으로 반영하기 위해 감시 모드에서 REPL을 실행하는 것이 유용합니다:

```
$ npm 실행 시작 -- -watch --entryFile repl
```

여기에는 한 가지 결함이 있는데, REPL의 명령 기록은 재로드할 때마다 삭제되므로 번거로울 수 있습니다. 다행히도 아주 간단한 해결책이 있습니다. [부트스트랩](#) 함수를 다음과 같이 수정하면 됩니다:

```
비동기 함수 부트스트랩() {
  const replServer = await repl(AppModule);
  replServer.setupHistory(".nestjs_repl_history", (err) => {
    if (err) {
      console.error(err);
    }
  });
}
```

이제 실행/재로드 사이에 기록이 보존됩니다.

CRUD 생성기

프로젝트의 수명 기간 동안 새로운 기능을 구축할 때 애플리케이션에 새로운 리소스를 추가해야 하는 경우가 많습니다. 이러한 리소스에는 일반적으로 새 리소스를 정의할 때마다 반복해야 하는 여러 번의 반복 작업이 필요합니다.

소개

사용자 및 제품 엔터티라는 두 개의 엔터티에 대한 CRUD 앤드포인트를 노출해야 하는 실제 시나리오를 가정해 보겠습니다. 모범 사례에 따라 각 엔터티에 대해 다음과 같이 몇 가지 작업을 수행해야 합니다:

- 모듈을 생성하여 코드를 체계적으로 정리하고 명확한 경계를 설정(관련 컴포넌트 그룹화)하세요.
- CRUD 경로(또는 GraphQL 애플리케이션의 경우 쿼리/변형)를 정의하는 컨트롤러(`nest g co`)를 생성합니다.
- 비즈니스 로직을 구현하고 격리하기 위한 서비스 생성(네스트 [지에스](#))
- 리소스 데이터 형태를 나타내는 엔터티 클래스/인터페이스 생성
- 데이터 전송 개체(또는 GraphQL 애플리케이션의 경우 입력)를 생성하여 네트워크를 통해 데이터를 전송하는 방법을 정의합니다.

정말 많은 단계입니다!

Nest CLI는 이러한 반복적인 프로세스의 속도를 높이기 위해 모든 상용구 코드를 자동으로 생성하는 제너레이터(회로도)를 제공하여 이 모든 작업을 피하고 개발자 환경을 훨씬 더 단순하게 만듭니다.

정보 참고 이 스키마는 HTTP 컨트롤러, 마이크로서비스 컨트롤러, GraphQL 리졸버(코드 우선 및 스키마 우선 모두) 및 웹소켓 게이트웨이 생성을 지원합니다.

새 리소스 생성

새 리소스를 만들려면 프로젝트의 루트 디렉터리에서 다음 명령을 실행하면 됩니다:

```
nest g resource
```

`nest g resource` 명령은 모든 NestJS 빌딩 블록(모듈, 서비스, 컨트롤러 클래스)뿐만 아니라 엔터티 클래스,

DTO 클래스, 테스트(`.spec`) 파일도 생성합니다.

아래에서 생성된 컨트롤러 파일(REST API용)을 확인할 수 있습니다:

```
컨트롤러('users')

사용자 컨트롤러 클래스 내보내기 {

생성자(비공개 읽기 전용 usersService: UsersService) {}

@Post()
create(@Body() createUserDto: CreateUserDto) {
```

```
이.usersService.create(createUserDto)를 반환합니다;
}

@GetMapping()
findAll() {
    this.usersService.findAll()을 반환합니다;
}

@GetMapping(':id')
findOne(@Param('id') id: 문자열) { return
    this.usersService.findOne(+id);
}

@Patch(':id')
update(@Param('id') id: 문자열, @Body() updateUserDto: UpdateUserDto) {
    return this.usersService.update(+id, updateUserDto);
}

삭제(':id') remove(@Param('id')
id: 문자열) {
    this.usersService.remove(+id)를 반환합니다;
}
}
```

또한 손가락 하나 까딱하지 않고도 모든 CRUD 엔드포인트(REST API의 경우 경로, GraphQL의 경우 쿼리 및 변이, 마이크로서비스 및 웹소켓 게이트웨이의 경우 메시지 구독)에 대한 자리 표시자를 자동으로 생성합니다.

경고 생성된 서비스 클래스는 특정 ORM(또는 데이터 소스)에 연결되지 않습니다. 따라서 생성기는 모든 프로젝트의 요구 사항을 충족할 수 있을 만큼 충분히 일반적입니다. 기본적으로 모든 메서드에는 프로젝트에 특정한 데이터 소스로 채울 수 있는 자리 표시자가 포함됩니다.

마찬가지로 GraphQL 애플리케이션용 리졸버를 생성하려면 전송 계층으로 [GraphQL\(코드 우선\)](#)(또는 [GraphQL\(스키마 우선\)](#))을 선택하기만 하면 됩니다.

이 경우 NestJS는 REST API 컨트롤러 대신 리졸버 클래스를 생성합니다:

nest g 리소스 사용자

- > ? 어떤 전송 계층을 사용하시나요? GraphQL(코드 우선)
- > ? CRUD 진입점을 생성하시겠습니까? 예
- > CREATE src/users/users.module.ts (224바이트)
- > src/users/users.resolver.spec.ts 생성 (525 바이트)
- > src/users/users.resolver.ts 생성(1109바이트)
- > CREATE src/users/users.service.spec.ts (453바이트)
- > CREATE src/users/users.service.ts (625바이트)
- > CREATE src/users/dto/create-user.input.ts (195 바이트)
- > src/users/dto/update-user.input.ts 생성 (281바이트)
- > CREATE src/users/entities/user.entity.ts (187 바이트)
- > src/app.module.ts 업데이트 (312바이트)

정보 힌트 테스트 파일을 생성하지 않으려면 다음과 같이 `--no-spec` 플래그를 전달할 수 있습니다: `nest`

```
g resource users --no-spec
```

아래에서 모든 상용구 변이와 쿼리가 생성되었을 뿐만 아니라 모든 것이 서로 연결되어 있음을 알 수 있습니다. 우리는 사용자 서비스, 사용자 엔티티, DTO를 활용하고 있습니다.

```
'@nestjs/graphql'에서 { Resolver, Query, Mutation, Args, Int }를 가져오고,
'./users.service'에서 { UsersService }를 가져옵니다;
'./entities/user.entity'에서 { 사용자 }를 가져옵니다;
'./dto/create-user.input'에서 { CreateUserInput } 가져오기;
'./dto/update-user.input'에서 { UpdateUserInput } 가져오기;

@Resolver(() => 사용자)
사용자 해결자 클래스 내보내기 {

생성자(비공개 읽기 전용 usersService: UsersService) {}

@Mutation(() => 사용자)
createUser(@Args('createUserInput') createUserInput: CreateUserInput) {
    return this.usersService.create(createUserInput);
}

@Query(() => [User], { name: 'users' })
findAll() {
    this.usersService.findAll()을 반환합니다;
}

쿼리(() => 사용자, { 이름: '사용자' })
findOne(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.findOne(id);
}

@Mutation(() => 사용자)
updateUser(@Args('updateUserInput') updateUserInput: UpdateUserInput) {
    return this.usersService.update(updateUserInput.id, updateUserInput);
}

@Mutation(() => 사용자)
removeUser(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.remove(id);
}
```

SWC

SWC(Speedy Web Compiler)는 컴파일과 번들링에 모두 사용할 수 있는 확장 가능한 Rust 기반 플랫폼입니다.

Nest CLI와 함께 SWC를 사용하면 개발 프로세스의 속도를 크게 높일 수 있는 간단하고 훌륭한 방법입니다.

정보 힌트 SWC는 기본 TypeScript 컴파일러보다 약 20배 빠릅니다.

설치

시작하려면 먼저 몇 가지 패키지를 설치하세요:

```
$ npm i --save-dev @swc/cli @swc/core
```

시작하기

설치 프로세스가 완료되면 다음과 같이 Nest CLI와 함께 swc 빌더를 사용할 수 있습니다:

```
nest start -b swc
# OR nest start --builder swc
```

정보 힌트 리포지토리가 모노레포인 경우 [이 섹션을 확인하세요](#).

b 플래그를 전달하는 대신 다음과 같이 `nest-cli.json` 파일에서 `compilerOptions.builder` 속성을 "swc"로 설정할 수도 있습니다:

```
{
  "컴파일러옵션": {
    "빌더": "swc"
  }
}
```

빌더의 동작을 사용자 정의하려면 두 가지 속성이 포함된 객체, 유형("swc")과 옵션은 다음과 같습니다:

```
"컴파일러옵션": { "빌더": {  
    "유형": "SWC", "옵션  
    ": {  
        "swcrcPath": "인프라/.swcrc",  
    }  
},  
}
```

감시 모드에서 애플리케이션을 실행하려면 다음 명령을 사용합니다:

```
nest start -b swc -w  
# 또는 nest start --builder swc --watch
```

유형 검사

SWC는 기본 타입스크립트 컴파일러와 달리 자체적으로 타입 검사를 수행하지 않으므로 이를 켜려면 `--`

`type-check` 플래그를 사용해야 합니다:

```
nest start -b swc --type-check
```

이 명령은 Nest CLI가 유형 검사를 비동기적으로 수행하는 SWC와 함께 `noEmit` 모드에서 `tsc`를 실행하도록 지시합니다. 다시 말하지만, `--type-check` 플래그를 전달하는 대신 다음과 같이 `nest-cli.json` 파일에서 `compilerOptions.typeCheck` 속성을 `true`로 설정할 수도 있습니다:

```
{  
  "컴파일러옵션": { "빌더":  
    "swc", "typeCheck": true  
  }  
}
```

CLI 플러그인(SWC)

`type-check` 플래그를 사용하면 NestJS CLI 플러그인을 자동으로 실행하고 직렬화된 메타데이터 파일을 생성하여 런타임에 애플리케이션에서 로드할 수 있습니다.

SWC 구성

SWC 빌더는 NestJS 애플리케이션의 요구 사항에 맞게 사전 구성되어 있습니다. 그러나 루트 디렉터리에 `.swcrc` 파일을 생성하고 원하는 대로 옵션을 조정하여 구성을 사용자 지정할 수 있습니다.

```
{  
  "$schema": "https://json.schemastore.org/swcrc",  
  "sourceMaps": true,  
  "jsc": {  
    "파서": {  
      "구문": "typescript",  
      "decorators": true,  
      "dynamicImport": true  
    },  
    "baseUrl": "./"
```

```
},
  "minify": false
}
```

모노레포

리포지토리가 모노레포인 경우 `swc` 빌더를 사용하는 대신 `웹팩`을 사용하도록 구성해야 합니다.

`swc-loader`.

먼저 필요한 패키지를 설치해 보겠습니다:

```
$ npm i --save-dev swc-loader
```

설치가 완료되면 애플리케이션의 루트 디렉터리에 다음 내용으로 `webpack.config.js` 파일을 생성합니다:

```
const swcDefaultConfig = require('@nestjs/cli/lib/compiler/defaults/swc-defaults').swcDefaultsFactory().swcOptions;

module.exports = {
  module: {
    규칙: [
      {
        test: /\.ts$/,
        제외합니다: /node_modules/, 사
        용: {
          로더: 'swc-loader', 옵션:
            swcDefaultConfig,
        },
      ],
    ],
  },
};
```

모노레포 및 CLI 플러그인

이제 CLI 플러그인을 사용하는 경우 `swc-loader`가 자동으로 로드하지 않습니다. 대신 수동으로 로드할 별도의 파일을 만들어야 합니다. 이렇게 하려면 `main.ts` 파일 근처에 `생성 메타데이터.ts` 파일을 다음 내용으로 선언합니다:

```
'@nestjs/cli/lib/compiler/plugins'에서 {
PluginMetadataGenerator }를 임포트합니다;
'@nestjs/swagger/dist/plugin'에서 { ReadonlyVisitor }를 가져옵니다;

const generator = new PluginMetadataGenerator();
generator.generate({
```

```
방문자: [new ReadonlyVisitor({ introspectComments: true, pathToSource:  
_dirname }),  
outputDir: _dirname,  
watch: true,  
tsconfigPath: 'apps/<이름>/tsconfig.app.json',  
});
```

정보 힌트 이 예제에서는 [@nestjs/swagger](#) 플러그인을 사용했지만 원하는 플러그인을 사용할 수 있습니다.

`generate()` 메서드는 다음 옵션을 허용합니다:

`watch` 프로젝트의 변경 사항을 주시할지 여부입니다.

`tsconfigPath` `tsconfig.json` 파일의 경로입니다. 현재 작업 디렉터리를 기준으로 합니다.
(`process.cwd()`).

`outputDir` 메타데이터 파일이 저장될 디렉터리 경로입니다. 메타데이터를

생성하는 데 사용할 방문자 배열입니다. 파일 이름 메타데이터 파일의 이름입니다. 기

본값은 `metadata.ts`입니다. `printDiagnostics` 콘솔에 진단을 인쇄할지 여부입니다

. 기본값은 `true`입니다.

마지막으로 다음 명령을 사용하여 별도의 터미널 창에서 [생성-메타데이터](#) 스크립트를 실행할 수 있습니다:

```
$ npx ts-node src/generate-metadata.ts  
# 또는 npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

일반적인 함정

애플리케이션에서 TypeORM/MikroORM 또는 다른 ORM을 사용하는 경우 순환 가져오기 문제가 발생할 수 있습니다
. SWC는 순환 가져오기를 잘 처리하지 못하므로 다음 해결 방법을 사용해야 합니다:

엔티티()

사용자 클래스 내보내기 {

 @OneToOne(() => 프로필, (프로필) => 프로필.사용자)

 프로필: Relation<Profile>; // <--- 여기서는 "Profile" 대신 "Relation<>" 유형

을 참조하십시오.

} 정보 힌트 관계 유형은 typeorm 패키지에서 내보냅니다.

이렇게 하면 프로퍼티 메타데이터의 트랜스파일 코드에 프로퍼티 유형이 저장되지 않으므로 순환 종속성 문제를 방지할 수 있습니다.

ORM에서 유사한 해결 방법을 제공하지 않는 경우 래퍼 유형을 직접 정의할 수 있습니다:

```
/**
 * ESM 모듈 순환 종속성 문제를 우회하는 데 사용되는 래퍼 유형
 * 속성 유형을 저장하는 리플렉션 메타데이터로 인해 발생합니다.
 */
export type WrapperType<T> = T; // WrapperType === Relation
```

프로젝트의 모든 [순환](#) 종속성 주입에 대해서도 위에서 설명한 사용자 정의 래퍼 유형을 사용해야 합니다:

```
@Injectable()
내보내기 클래스 UserService { 생
    성자(
        @Inject(forwardRef(() => ProfileService))
        비공개 읽기 전용 profileService: 래퍼 유형<프로필 서비스>,
    ) {};
}
```

제스트 + SWC

Jest에서 SWC를 사용하려면 다음 패키지를 설치해야 합니다:

```
$ npm i --save-dev jest @swc/core @swc/jest
```

설치가 완료되면 구성에 따라 `package.json/jest.config.js` 파일을 다음 내용으로 업데이트합니다:

```
{
  "농담": {
    "transform": {
      "^.+\\.(t|j)s?$/": ["@swc/jest"]
    }
  }
}
```

또한 `.swcrc` 파일에 다음 [트랜스폼](#) 속성을 추가해야 합니다:

레거시 데코레이터, 데코레이터 메타데이터:

```
{  
  "$schema": "https://json.schemastore.org/swcrc",  
  "sourceMaps": true,  
  "jsc": {
```

```
"파서": {  
    "구문": "typescript",  
    "decorators": true,  
    "dynamicImport": true  
},  
"transform": {  
    "legacyDecorator": true,  
    "decoratorMetadata": true  
},  
"baseUrl": "./"  
},  
"minify": false  
}
```

프로젝트에서 NestJS CLI 플러그인을 사용하는 경우 [플러그인 메타데이터 생성기](#)를 수동으로 실행해야 합니다. 자세한 내용을 알아보려면 [이 섹션으로](#) 이동하세요.

Vitest

Vitest는 Vite와 함께 작동하도록 설계된 빠르고 가벼운 테스트 러너입니다. NestJS 프로젝트와 통합할 수 있는 현대적이고 빠르며 사용하기 쉬운 테스트 솔루션을 제공합니다.

설치

시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save-dev vitest unplugin-swc @swc/core @vitest/coverage-c8
```

구성

애플리케이션의 루트 디렉터리에 다음 내용으로 `vitest.config.ts` 파일을 생성합니다:

```
'unplugin-swc'에서 SWC를 가져옵니다;  
'vitest/config'에서 { defineConfig }를 가져옵니다;  
  
export default defineConfig({  
  test: {  
    globals: true,  
    root: './',  
  },  
  플러그인: [  
    // SWC로 테스트 파일을 빌드하는 데 필요합니다 swc.vite({  
      // 모듈 유형을 명시적으로 설정하여 `.swcrc` 구성 파일에서 이 값을 상속하지 않도록 합니다.  
      모듈을 추가합니다: { 유형: 'es6' },  
    }),
```

```
],  
});
```

이 구성 파일은 Vitest 환경, 루트 디렉토리 및 SWC 플러그인을 설정합니다. 또한 테스트 경로 정규식을 지정하는 추가 포함 필드를 사용하여 e2e 테스트를 위한 별도의 구성 파일을 만들어야 합니다:

```
'unplugin-swc'에서 SWC를 가져옵니다;  
'vitest/config'에서 { defineConfig }를 가져옵니다;  
  
export default defineConfig({  
  test: {  
    포함: ['**/*.e2e-spec.ts'], globals:  
      true,  
      root: './',  
    },  
    플러그인: [swc.vite()],  
  });
```

또한 테스트에서 TypeScript 경로를 지원하도록 별칭 옵션을 설정할 수 있습니다:

```
'unplugin-swc'에서 SWC를 가져옵니다;  
'vitest/config'에서 { defineConfig }를 가져옵니다;  
  
export default defineConfig({  
  test: {  
    포함: ['**/*.e2e-spec.ts'], globals:  
      true,  
      별칭: {  
        '@src': './src',  
        '@test': './test',  
      },  
      root: './',  
    },  
    해결합니다:  
    { alias:  
      {  
        '@src': './src',  
        '@test': './test',  
      },  
    },  
    플러그인: [swc.vite()],  
  });
```

E2E 테스트에서 가져오기 업데이트

`import *`를 사용하는 모든 E2E 테스트 가져오기를 '`supertest`'의 요청으로 가져오기 요청으로 변경합니다. 이는 Vite와 번들로 제공되는 경우 Vitest가 기본 가져오기를 기대하기 때문에 필요합니다.

를 사용하세요. 네임스페이스 가져오기를 사용하면 이 특정 설정에서 문제가 발생할 수

있습니다. 마지막으로 package.json 파일의 테스트 스크립트를 다음과 같이 업데이트

합니다:

```
{  
  "스크립트": {  
    "테스트": "바이테스트 실행",  
    "test:watch": "vitest",  
    "test:cov": "VITEST RUN --COVERAGE",  
    "test:debug": "vitest --inspect-brk --inspect --logHeapUsage --  
      threads=false",  
    "test:e2e": "vitest run --config ./vitest.config.e2e.ts"  
  }  
}
```

이 스크립트는 테스트 실행, 변경 사항 감시, 코드 커버리지 보고서 생성 및 디버깅을 위해 Vitest를 구성합니다.

test:e2e 스크립트는 특히 사용자 지정 구성 파일로 E2E 테스트를 실행하기 위한 스크립트입니다.

이 설정을 통해 이제 테스트 실행 속도 향상 및 최신 테스트 환경 등 NestJS 프로젝트에서 Vitest를 사용하는 이점
을 누릴 수 있습니다.

정보 힌트 이 [리포지토리에서](#) 작동하는 예제를 확인할 수 있습니다.

여권(인증)

Passport는 커뮤니티에서 잘 알려져 있고 많은 프로덕션 애플리케이션에서 성공적으로 사용되는 가장 인기 있는 node.js 인증 라이브러리입니다. 이 라이브러리를 Nest 애플리케이션과 통합하는 방법은 [@nestjs/passport](#) 모듈을 사용하면 간단합니다. 높은 수준에서 Passport는 다음과 같은 일련의 단계를 실행합니다:

- 사용자 '자격 증명'(예: 사용자 이름/비밀번호, JSON 웹 토큰([JWT](#)) 또는 ID 공급자의 ID 토큰)을 확인하여 사용자를 인증합니다.
- 인증 상태 관리([JWT](#)와 같은 휴대용 토큰을 발급하거나 [Express 세션을](#) 만들어서)
- 인증된 사용자에 대한 정보를 [요청](#) 개체에 첨부하여 라우트 핸들러에서 추가로 사용할 수 있습니다.

Passport에는 다양한 인증 메커니즘을 구현하는 풍부한 [전략](#) 생태계가 있습니다. 개념은 간단하지만, 선택할 수 있는 Passport 전략의 집합은 매우 크고 다양합니다. Passport는 이러한 다양한 단계를 표준 패턴으로 추상화하며, [@nestjs/passport](#) 모듈은 이 패턴을 익숙한 Nest 구조로 래핑하고 표준화합니다.

이 장에서는 이러한 강력하고 유연한 모듈을 사용하여 RESTful API 서버를 위한 완전한 엔드투엔드 인증 솔루션을 구현해 보겠습니다. 여기에 설명된 개념을 사용하여 인증 체계를 사용자 지정하기 위해 모든 Passport 전략을 구현할 수 있습니다. 이 장의 단계에 따라 이 완전한 예제를 구축할 수 있습니다.

인증 요구 사항

요구 사항을 구체화해 봅시다. 이 사용 사례에서 클라이언트는 사용자 이름과 비밀번호로 인증하는 것으로 시작합니다. 인증이 완료되면 서버는 인증을 증명하기 위해 후속 요청 시 인증 [헤더에 베어러 토큰으로](#) 전송할 수 있는 JWT를 발급합니다. 또한 유효한 JWT가 포함된 요청만 액세스할 수 있는 보호된 경로를 생성합니다.

첫 번째 요구 사항인 사용자 인증부터 시작하겠습니다. 그런 다음 JWT를 발행하여 이를 확장합니다. 마지막으로 요청에 대해 유효한 JWT를 검사하는 보호된 경로를 생성합니다.

먼저 필요한 패키지를 설치해야 합니다. Passport는 사용자 이름/비밀번호 인증 메커니즘을 구현하는 [passport-local](#)이라는 전략을 제공하므로 이 사용 사례의 요구 사항에 적합합니다.

```
$ npm install --save @nestjs/passport passport-local
$ npm install --save-dev @types/passport-local
```

경고 주의 어떤 패스포트 전략을 선택하든 항상 [@nestjs/passport](#) 및 [패스포트](#) 패키지가 필요합니다. 그런 다음 구축하려는 특정 인증 전략을 구현하는 전략별 패키지(예: [passport-jwt](#) 또는 [passport-local](#))를 설치해야 합니다. 또한 위에 표시된 것처럼 [@types/passport-local](#)을 사용하여 모든 Passport 전략에 대한 유형 정의를 설치하면 TypeScript 코드를 작성하는 동안 도움을 받을 수 있습니다.

패스포트 전략 구현

이제 인증 기능을 구현할 준비가 되었습니다. 먼저 Passport 전략에 사용되는 프로세스에 대한 개요부터 살펴보겠습니다. Passport는 그 자체로 하나의 미니 프레임워크라고 생각하면 도움이 됩니다. 이 프레임워크의 장점은 인증 프로세스를 구현하는 전략에 따라 사용자 지정할 수 있는 몇 가지 기본 단계로 추상화한다는 것입니다. 사용자 지정 매개변수(일반 JSON 객체)와 콜백 함수 형태의 사용자 지정 코드를 제공하여 구성하고, Passport가 적절한 시점에 이를 호출하기 때문에 프레임워크와 비슷합니다. 이 프레임워크를 Nest 스타일 패키지로 감싸는 `@nestjs/passport` 모듈은 Nest 애플리케이션에 쉽게 통합할 수 있도록 해줍니다. 아래에서는 `@nestjs/passport`를 사용하겠지만, 먼저 바닐라 Passport가 어떻게 작동하는지 살펴봅시다.

바닐라 패스포트에서는 두 가지를 제공하여 전략을 구성합니다:

- . 해당 전략에 특정한 옵션 집합입니다. 예를 들어 JWT 전략에서는 토큰에 서명하기 위한 비밀을 제공할 수 있습니다.
- . "확인 콜백"은 사용자 스토어(사용자 계정을 관리하는 곳)와 상호 작용하는 방법을 Passport에 알려주는 곳입니다. 여기에서 사용자가 존재하는지(및/또는 새 사용자를 생성하는지), 자격 증명이 유효한지 확인합니다. Passport 라이브러리는 이 콜백이 유효성 검사에 성공하면 전체 사용자를 반환하고, 실패하면 `null`을 반환할 것으로 예상합니다(실패는 사용자를 찾을 수 없거나 패스포트-로컬의 경우 비밀번호가 일치하지 않는 경우로 정의됨).

`nestjs/passport`를 사용하면 `PassportStrategy` 클래스를 확장하여 패스포트 전략을 구성할 수 있습니다. 하위 클래스에서 `super()` 메서드를 호출하여 전략 옵션(위 항목 1)을 전달하고, 선택적으로 옵션 객체를 전달합니다. 하위 클래스에서 `validate()` 메서드를 구현하여 확인 콜백(위 항목 2)을 제공합니다.

먼저 `AuthModule`을 생성하고 그 안에 `AuthService`를 생성하겠습니다:

```
nest g 모듈 인증  
nest g 서비스 인증
```

`AuthService`를 구현하면서 사용자 작업을 `UserService`에 캡슐화하는 것이 유용하다는 것을 알게 될 것이므로 이제 해당 모듈과 서비스를 생성해 보겠습니다:

```
nest g 모듈 사용자  
nest g 서비스 사용자
```

생성된 파일의 기본 내용을 아래와 같이 바꿉니다. 샘플 앱의 경우, 사용자 서비스는 단순히 하드코딩된 인메모리

사용자 목록과 사용자 이름으로 사용자를 검색하는 찾기 메서드를 유지 관리합니다. 실제 앱에서는 선택한 라이브러리(예: TypeORM, Sequelize, 몽구스 등)를 사용하여 사용자 모델과 지속성 레이어를 빌드할 수 있습니다.

@@파일명(사용자/사용자.서비스)

'@nestjs/common'에서 { Injectable }을 가져옵니다;

// 사용자 엔티티 내보내기 유형 User = any를 나타내는 실제 클래스/인터페이스여야 합니다;

```
@Injectable()
export 클래스 UserService {
    private 읽기 전용 사용자 = [
        {
            userId: 1, 사용자명:
            'john',
            비밀번호: 'changeme',
        },
        {
            userId: 2, 사용자 이름: 'maria', 비밀번호
            : 'guess',
        },
    ];
}

async findOne(username: 문자열): Promise<사용자 | 정의되지 않음> {
    return this.users.find(사용자 => 사용자.사용자이름 === 사용자이름
    );
}
}
@@switch
'@nestjs/common'에서 { Injectable }을 가져옵니다;

@Injectable()
내보내기 클래스 UserService {
    constructor() {
        this.users = [
            {
                userId: 1, 사용자명
                : 'john',
                비밀번호: 'changeme',
            },
            {
                userId: 2, 사용자
                이름: 'maria', 비밀
                번호: 'guess',
            },
        ],
    };
}

async findOne(username) {
    반환 이.사용자.찾기(사용자 => 사용자.사용자 이름 === 사용자 이름);
}
}
```

UsersModule에서 필요한 유일한 변경 사항은 UsersService를 모듈 데코레이터를 추가하여 이 모듈 외부에서 볼 수 있도록 합니다(곧 AuthService에서 사용하게 될 것입니다).

@@파일명(사용자/사용자.모듈)

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./users.service'에서 { UsersService }를 가져옵니다;  
다;
```

```
모듈({  
  제공자: [UserService], 내보  
  내기: [UserService],  
})
```

사용자 모듈 클래스 {} @@스위치 내보

내기

'@nestjs/common'에서 { Module }을 가져오고,
'./users.service'에서 { UserService }를 가져옵니다;
다;

```
모듈({  
  제공자: [UserService], 내보  
  내기: [UserService],  
})
```

사용자 모듈 클래스 {} 내보내기

인증 서비스는 사용자를 검색하고 비밀번호를 확인하는 작업을 수행합니다. 이를 위해 `validateUser()` 메서드를 생성합니다. 아래 코드에서는 편리한 ES6 스프레드 연산자를 사용하여 사용자 객체에서 비밀번호 속성을 제거한 후 반환합니다. 잠시 후 Passport 로컬 전략에서 `validateUser()` 메서드를 호출하겠습니다.

```
@@파일명(auth/auth.service)
'@nestjs/common'에서 { Injectable }을 임포트합니다;
'./users/users.service'에서 { UserService }를 가져옵니다;

@Injectable()
내보내기 클래스 AuthService {
  constructor(private userService: UserService) {}

  async validateUser(사용자 이름: 문자열, 패스: 문자열): Promise<any> {
    const user = await this.userService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user; 반
      환 결과;
    }
    널을 반환합니다;
  }
}
@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 임포트하고,
'./users/users.service'에서 { UserService }를 임포트합니다;

주입 가능()
@Dependencies(UserService)
내보내기 클래스 AuthService {
  constructor(userService) {
    this.userService = userService;
  }

  async validateUser(username, pass) {
    const user = await this.userService.findOne(username);
```

```

if (user && user.password === pass) {
  const { password, ...result } = user; 반
  환 결과;
}

널을 반환합니다;
}
}

```

경고 경고 물론 실제 애플리케이션에서는 비밀번호를 일반 텍스트로 저장하지 않습니다. 대신 솔트 처리된 단방향 해시 알고리즘이 포함된 [bcrypt](#)와 같은 라이브러리를 사용할 것입니다. 이 접근 방식을 사용하면 해시된 비밀번호만 저장한 다음 저장된 비밀번호를 들어오는 비밀번호의 해시된 버전과 비교하므로 사용자 비밀번호를 일반 텍스트로 저장하거나 노출하지 않습니다. 샘플 앱을 단순하게 유지하기 위해 이러한 절대적인 의무를 위반하고 일반 텍스트를 사용했습니다. 실제 앱에서는 이렇게 하지 마세요!

이제 [AuthModule](#)을 업데이트하여 [UsersModule](#)을 가져옵니다.

@@파일명(auth/auth.module)

```

'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'./users/users.module'에서 { UsersModule }을 가져옵니다;

```

모듈({

```

  임포트: [UsersModule], 공급자
  : [AuthService],
})

```

내보내기 클래스 AuthModule {}

@@switch

```

'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'./users/users.module'에서 { UsersModule }을 가져옵니다;

```

모듈({

```

  임포트: [UsersModule], 공급자
  : [AuthService],
})

```

내보내기 클래스 AuthModule {}

패스포트 로컬 구현

이제 Passport 로컬 인증 전략을 구현할 수 있습니다. 다음과 같은 파일을 만듭니다.

local.strategy.ts를 열고 다음 코드를 추가합니다:

```
@@파일명(auth/local.strategy)
'passport-local'에서 { Strategy }를 가져옵니다;
'@nestjs/passport'에서 { PassportStrategy }를 가져옵니다;
'@nestjs/common'에서 { Injectable, UnauthorizedException }을 임포트하고,
'./auth.service'에서 { AuthService }를 임포트합니다;
```

```

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super();
  }

  비동기 유효성 검사(사용자 이름: 문자열, 비밀번호: 문자열): Promise<any> {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      새로운 UnauthorizedException()을 던집니다;
    }
    사용자를 반환합니다;
  }
}

@@switch
'passport-local'에서 { Strategy }를 가져옵니다;
'@nestjs/passport'에서 { PassportStrategy }를 가져옵니다;
'@nestjs/common'에서 { Injectable, UnauthorizedException, Dependencies }를 가져옵니다;
'./auth.service'에서 { AuthService }를 가져옵니다;

주입 가능() @종속성(AuthService)
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(authService) {
    super();
    this.authService = authService;
  }

  async validate(username, password) {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      새로운 UnauthorizedException()을 던집니다;
    }
    사용자를 반환합니다;
  }
}

```

모든 패스포트 전략에 대해 앞서 설명한 레시피를 따랐습니다. 패스포트-로컬 사용 사례에서는 구성 옵션이 없으므로 생성자는 옵션 객체 없이 단순히 `super()`를 호출합니다.

정보 힌트 `super()` 호출에 옵션 객체를 전달하여 패스포트 전략의 동작을 사용자 정의할 수 있습니다. 이 예제에서 패스포트-로컬 전략은 기본적으로 요청 본문에서 `사용자 이름과 비밀번호라는 속성`을 기대합니다. 옵션 객체를 전달하여 다른 속성 이름을 지정할 수 있습니다(예: `super({{ '{' }} usernameField: 'email' {{ '}' }})`). 자세한 내용은 [Passport 문서](#)를 참조하세요.

또한 `validate()` 메서드도 구현했습니다. 각 전략에 대해 Passport는 적절한 전략별 매개변수 집합을 사용하여 검증 함수([@nestjs/passport](#)에서 `validate()` 메서드로 구현됨)를 호출합니다. 로컬 전략의 경우, Passport는 다음과 같은 서명을 가진 `validate()` 메서드를 기대합니다: `validate(username: 문자열, password:문자열) : any`.

대부분의 유효성 검사 작업은 `AuthService`에서 수행되므로(`UserService`의 도움으로) 이 메서드는 매우 간단합니다. 모든 Passport 전략의 유효성 검사() 메서드는 자격 증명이 표시되는 세부 사항만 다를 뿐 비슷한 패턴을 따릅니다. 사용자를 찾고 자격 증명이 유효하면 사용자를 반환하여 Passport가 작업(예: 요청 객체에서 사용자 속성 만들기)을 완료하고 요청 처리 파이프라인을 계속할 수 있습니다. 사용자를 찾을 수 없으면 예외를 발생시키고 [예외 계층에서 처리하도록 합니다](#).

일반적으로 각 전략의 유효성 검사() 메서드에서 유일하게 중요한 차이점은 사용자가 존재하고 유효한지 확인하는 방법입니다. 예를 들어, JWT 전략에서는 요구 사항에 따라 디코딩된 토큰에 포함된 `userId`가 사용자 데이터 베이스의 레코드와 일치하는지 또는 해지된 토큰 목록과 일치하는지 평가할 수 있습니다. 따라서 전략별 유효성 검사를 하위 분류하고 구현하는 이러한 패턴은 일관성 있고 우아하며 확장 가능합니다.

방금 정의한 패스포트 기능을 사용하도록 `AuthModule`을 구성해야 합니다. 업데이트

`auth.module.ts`를 다음과 같이 수정합니다:

@@파일명(auth/auth.module)

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다  
;  
'.../users/users.module'에서 { UsersModule }을 임포트하고,  
'@nestjs/passport'에서 { PassportModule }을 임포트하고,  
'./local.strategy'에서 { LocalStrategy }를 임포트합니다;
```

모듈({

```
    임포트합니다: [UsersModule, PassportModule], 공  
    급자: [AuthService, LocalStrategy],  
})
```

내보내기 클래스 AuthModule {}

@@switch

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다  
;  
'.../users/users.module'에서 { UsersModule }을 임포트하고,  
'@nestjs/passport'에서 { PassportModule }을 임포트하고,  
'./local.strategy'에서 { LocalStrategy }를 임포트합니다;
```

모듈({

```
    임포트합니다: [사용자 모듈, 패스포트 모듈], 공급자:  
    [AuthService, LocalStrategy],  
})
```

내보내기 클래스 AuthModule {}

내장형 여권 가드

가드 챕터에서는 가드의 주요 기능인 요청이 라우트 핸들러에 의해 처리될지 여부를 결정하는 기능에 대해 설명합니다. 이는 여전히 유효하며 곧 이 표준 기능을 사용하게 될 것입니다.

그러나 `@nestjs/passport` 모듈을 사용하는 맥락에서 처음에는 혼란스러울 수 있는 약간의 새로운 주름도 소개할 것이므로 지금부터 이에 대해 논의해 보겠습니다. 인증 관점에서 앱이 두 가지 상태로 존재할 수 있다고 가정해 보겠습니다:

사용자/클라이언트가 로그인되지 않음(인증되지 않음)

사용자/클라이언트가 로그인(인증됨)되었습니다.

첫 번째 경우(사용자가 로그인하지 않은 경우)에는 두 가지 기능을 수행해야 합니다:

- 인증되지 않은 사용자가 액세스할 수 있는 경로를 제한합니다(즉, 제한된 경로에 대한 액세스를 거부합니다). 이 기능을 처리하기 위해 보호된 경로에 가드를 배치하여 익숙한 기능인 가드를 사용할 것입니다. 예상 할 수 있듯이 이 가드에 유효한 JWT가 있는지 확인할 것이므로 나중에 JWT를 성공적으로 발급한 후 이 가드에 대해 작업할 것입니다.
- 이전에 인증되지 않은 사용자가 로그인을 시도할 때 인증 단계 자체를 시작합니다. 이 단계는 유효한 사용 자에게 JWT를 발급하는 단계입니다. 잠시 생각해보면 인증을 시작하려면 사용자 이름/비밀번호 자격 증명을 **POST해야** 한다는 것을 알 수 있으므로 **POST /auth/login** 경로를 사용하여 처리할 수 있습니다. 이 경우 해당 경로에서 정확히 어떻게 패스포트-로 컬 전략을 호출할 수 있을지에 대한 의문이 생깁니다.

답은 간단합니다. 약간 다른 유형의 가드를 사용하면 됩니다. 이 작업을 수행하는 내장된 가드를 **@nestjs/passport** 모듈에서 제공합니다. 이 가드는 패스포트 전략을 호출하고 위에서 설명한 단계(자격 증명 검 색, 확인 함수 실행, 사용자 속성 생성 등)를 시작합니다.

위에 열거한 두 번째 경우(로그인한 사용자)는 로그인한 사용자가 보호된 경로에 액세스할 수 있도록 이미 설명한 표준 유형의 가드에 의존하기만 하면 됩니다.

로그인 경로

이제 전략이 수립되었으므로 **베어본/인증/로그인** 경로를 구현하고 기본 제공 Guard를 적용하여 여권-로컬 플로우를 시작할 수 있습니다.

app.controller.ts 파일을 열고 내용을 다음과 같이 바꿉니다:

@@파일명(앱.컨트롤러)

'@nestjs/common'에서 { Controller, Request, Post, UseGuards }를 가져오고,
'@nestjs/passport'에서 { AuthGuard }를 가져옵니다;

컨트롤러()

```
export class AppController {  
  @UseGuards(AuthGuard('local'))  
  Post('auth/login')  
  async login(@Request() req) {  
    return req.user;  
  }  
}
```

@@switch

'@nestjs/common'에서 { Controller, Bind, Request, Post, UseGuards }를 임포트합니다;

'@nestjs/passport'에서 { AuthGuard }를 임포트합니다;

컨트롤러()

```
export class AppController {  
  @UseGuards(AuthGuard('local'))
```

```
Post('auth/login')
@Bind(Request())
async login(req) {
  req.user를 반환합니다;
}
}
```

사용가드(`AuthGuard('local')`)를 사용하면 패스포트-로컬 전략을 확장할 때 `@nestjs/passport`가 자동으로 프로비저닝한 `AuthGuard`를 사용하고 있습니다. 자세히 살펴보겠습니다. 패스포트 로컬 전략의 기본 이름은 '`local`'입니다. 이 이름을 `@UseGuards()` 데코레이터에서 참조하여 패스포트-로컬 패키지가 제공하는 코드와 연결합니다. 이는 앱에 여러 개의 패스포트 전략이 있는 경우 호출할 전략을 명확히 하기 위해 사용됩니다(각 전략은 전략별 `AuthGuard`를 제공할 수 있음). 지금까지는 이러한 전략이 하나뿐이지만 곧 두 번째 전략을 추가할 예정이므로 명확하게 구분하기 위해 필요합니다.

경로를 테스트하기 위해 지금은 `/auth/login` 경로에서 단순히 사용자를 반환하도록 하겠습니다. 이를 통해 또 다른 Passport 기능을 시연할 수 있습니다: Passport는 `validate()` 메서드에서 반환한 값을 기반으로 사용자 객체를 자동으로 생성하고 이를 요청 객체에 `req.user`로 할당합니다. 나중에는 이 대신 JWT를 생성하고 반환하는 코드로 대체하겠습니다.

API 경로이므로 일반적으로 사용 가능한 `cURL` 라이브러리를 사용하여 테스트해 보겠습니다. `UserService`에 하드코딩된 모든 사용자 객체로 테스트할 수 있습니다.

```
인증/로그인에 $ # POST 보내기
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
$ # result -> {"userId":1,"username":"john"}
```

이 방법은 작동하지만 전략 이름을 `AuthGuard()`에 직접 전달하면 코드베이스에 마법의 문자열이 생깁니다. 대신 아래와 같이 자체 클래스를 생성하는 것이 좋습니다:

```
@@파일명(auth/local-auth.guard)
'@nestjs/common'에서 { Injectable }을 임포트하고,
'@nestjs/passport'에서 { AuthGuard }를 임포트합니다
;

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}
```

이제 `/auth/login` 경로 핸들러를 업데이트하고 대신 `LocalAuthGuard`을 사용할 수 있습니다:

```
사용가드(LocalAuthGuard)
```

```
@Post('auth/login')
async login(@Request() req) {
  return req.user;
}
```

JWT 기능

이제 인증 시스템의 JWT 부분으로 넘어갈 준비가 되었습니다. 요구 사항을 검토하고 구체화해 보겠습니다:

- 사용자가 사용자 이름/비밀번호로 인증할 수 있도록 허용하여 보호된 API 엔드포인트에 대한 후속 호출에서 사용할 수 있도록 JWT를 반환합니다. 이 요구 사항을 충족하기 위한 작업은 순조롭게 진행 중입니다. 이를 완료하려면 JWT를 발급하는 코드를 작성해야 합니다.
- 무기명 토큰으로 유효한 JWT의 존재를 기반으로 보호되는 API 경로 만들기 JWT 요구 사항을 지원하

```
npm install --save @nestjs/jwt passport-jwt
$ npm install --save-dev @타입스/패스포트-jwt
```

`nestjs/jwt` 패키지(자세한 내용은 [여기를](#) 참조하세요)는 JWT 조작에 도움이 되는 유ти리티 패키지입니다. `passport-jwt` 패키지는 JWT 전략을 구현하는 Passport 패키지이며 `@types/passport-jwt`는 TypeScript 유형 정의를 제공합니다.

POST `/auth/login` 요청이 어떻게 처리되는지 자세히 살펴봅시다. 패스포트-로컬 전략에서 제공하는 내장 `AuthGuard`를 사용하여 경로를 꾸몄습니다. 즉

- . 경로 핸들러는 사용자가 유효성이 검사된 경우에만 호출됩니다.
- . `req` 매개변수에는 `사용자` 속성이 포함됩니다(여권-로컬 인증 흐름 중에 Passport에 의해 채워짐).

이를 염두에 두고 이제 실제 JWT를 생성하고 이 경로를 통해 반환할 수 있습니다. 서비스를 깔끔하게 모듈화하기 위해 `authService`에서 JWT 생성을 처리하겠습니다. `auth` 폴더에서 `auth.service.ts` 파일을 열고 `login()` 메서드를 추가한 후 그림과 같이 `JwtService`를 가져옵니다:

@@파일명(auth/auth.service)

'@nestjs/common'에서 { Injectable }을 임포트합니다;
'./users/users.service'에서 { UsersService }를 임포트하고,
'@nestjs/jwt'에서 { JwtService }를 임포트합니다;

@Injectable()

내보내기 클래스 AuthService { 생

성자(

개인 사용자 서비스: UsersService, 비공개

jwtService: JwtService

) {}

async validateUser(사용자 이름: 문자열, 패스: 문자열): Promise<any> { const

user = await this.usersService.findOne(username);

if (user && user.password === pass) {

const { password, ...result } = user; 반

환 결과;

}

널을 반환합니다;

```

    }

    async login(user: any) {
      const payload = { username: user.username, sub: user.userId };
      return {
        액세스_토큰: this.jwtService.sign(페이로드),
      };
    }
  }

@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 가져오고,
'./users/users.service'에서 { UsersService }를 가져오고,
'@nestjs/jwt'에서 { JwtService }를 가져오세요;

@Dependencies(UsersService, JwtService)
@Injectable()
export class AuthService {
  constructor(usersService, jwtService) {
    this.usersService = usersService;
    this.jwtService = jwtService;
  }

  async validateUser(username, pass) {
    const user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = 사용자;
      반환 결과;
    }
    널을 반환합니다;
  }

  async login(user) {
    const payload = { username: user.username, sub: user.userId };
    return {
      access_token: this.jwtService.sign(payload),
    };
  }
}

```

사용자 객체 속성의 하위 집합에서 JWT를 생성하는 `sign()` 함수를 제공하는 `@nestjs/jwt` 라이브러리를 사용하고 있으며, 이 라이브러리는 단일 `access_token` 속성을 가진 간단한 객체로 반환합니다. 참고: JWT 표준과 일관성을 유지하기 위해 `sub`라는 속성 이름을 선택하여 `userId` 값을 보유합니다. 인증 서비스에 JwtService 공급자를 삽입하는 것을 잊지 마세요.

이제 새 종속성을 가져오고 `JwtModule`을 구성하기 위해 `AuthModule`을 업데이트해야 합니다. 먼저 `auth` 폴더

에 `constants.ts`를 생성하고 다음 코드를 추가합니다:

```
@@파일명(auth/constants)
export const jwtConstants = {
```

```
비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게 보  
관하세요.',  
};  
@@switch  
export const jwtConstants = {  
  비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게 보  
관하세요.',  
};
```

이 키를 사용하여 JWT 서명 및 확인 단계 간에 키를 공유할 것입니다.

경고 경고 이 키를 공개적으로 노출하지 마세요. 여기서는 코드가 수행하는 작업을 명확히 하기 위해 공개 했지만, 프로덕션 시스템에서는 시크릿 볼트, 환경 변수 또는 구성 서비스 등의 적절한 조치를 사용하여 이 키를 보호해야 합니다.

이제 인증 폴더에서 `auth.module.ts`를 열고 다음과 같이 업데이트합니다:

@@파일명(auth/auth.module)

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다  
;  
'./local.strategy'에서 { LocalStrategy }를 임포트하고,  
'../users/users.module'에서 { UsersModule }을 임포트하고,  
'@nestjs/passport'에서 { PassportModule }을 임포트하고,  
'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;  
import { jwtConstants } './constants'에서 임포트합니다;
```

모듈({ import: [

```
UsersModule,  
PassportModule,  
JwtModule.register({  
    비밀: jwtConstants.secret,  
    signOptions: { expiresIn: '60s' },  
}),  
],
```

공급자: [AuthService, LocalStrategy], 내보내기:

```
[AuthService],  
})
```

내보내기 클래스 AuthModule {}

@@switch

```
'@nestjs/common'에서 { Module }을 가져오고,  
'./auth.service'에서 { AuthService }를 가져옵니다  
;  
'./local.strategy'에서 { LocalStrategy }를 임포트하고,  
'../users/users.module'에서 { UsersModule }을 임포트하고,  
'@nestjs/passport'에서 { PassportModule }을 임포트하고,  
'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;  
import { jwtConstants } './constants'에서 임포트합니다;
```

모듈({ import: [

```
UsersModule,
```

```
PassportModule,  
JwtModule.register({  
    비밀: jwtConstants.secret,  
    signOptions: { expiresIn: '60s' },  
}),  
],  
공급자: [AuthService, LocalStrategy], 내보내기:  
[AuthService],  
})  
내보내기 클래스 AuthModule {}
```

구성 객체를 전달하여 `register()`를 사용하여 `JwtModule`을 구성합니다. Nest `JwtModule`에 대한 자세한 내용은 [여기를](#), 사용 가능한 구성 옵션에 대한 자세한 내용은 [여기를](#) 참조하세요.

이제 `/auth/login` 경로를 업데이트하여 JWT를 반환할 수 있습니다.

@@파일명(앱.컨트롤러)

```
'@nestjs/common'에서 { Controller, Request, Post, UseGuards }를 임포트하고,  
'./auth/local-auth.guard'에서 { LocalAuthGuard }를 임포트합니다;  
'./auth/auth.service'에서 { AuthService }를 가져옵니다;
```

컨트롤러()

```
내보내기 클래스 AppController {  
    constructor(private authService: AuthService) {}
```

사용가드(LocalAuthGuard)

```
@Post('auth/login')  
async login(@Request() req) {  
    this.authService.login(req.user)을 반환합니다;  
}  
}  
@switch  
'@nestjs/common'에서 { Controller, Bind, Request, Post, UseGuards }를 임포트합니다;  
'./auth/local-auth.guard'에서 { LocalAuthGuard }를 가져오고,  
'./auth/auth.service'에서 { AuthService }를 가져옵니다;
```

컨트롤러()

```
내보내기 클래스 AppController {  
    constructor(private authService: AuthService) {}  
  
    UseGuards(LocalAuthGuard)  
    @Post('auth/login')  
    @Bind(Request())  
    async login(req) {  
        this.authService.login(req.user)을 반환합니다;  
    }  
}
```

계속해서 cURL을 사용하여 경로를 다시 테스트해 보겠습니다. [UsersService](#)에 하드코딩된 모든 [사용자](#) 객체를 사용하여 테스트할 수 있습니다.

```
인증/로그인에 $ # POST 보내기
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
$ # 결과 -> {"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}'
참고: 위 JWT는 잘렸습니다.
```

패스포트 JWT 구현하기

이제 마지막 요구 사항인 요청에 유효한 JWT가 있어야 엔드포인트를 보호할 수 있습니다. 여기서도 Passport가 도움이 될 수 있습니다. JSON 웹 토큰으로 RESTful 엔드포인트를 보호하기 위한 [passport-jwt](#) 전략을 제공합니다.

먼저 [인증 폴더](#)에 `jwt.strategy.ts`라는 파일을 만들고 다음 코드를 추가합니다:

@@파일명(auth/jwt.strategy)

'passport-jwt'에서 { ExtractJwt, Strategy }를 임포트하고, '@nestjs/passport'에서 { PassportStrategy }를 임포트하고, '@nestjs/common'에서 { Injectable }을 임포트합니다;

```
import { jwtConstants } './constants'에서 임포트합니다;

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      비밀 또는 키: jwtConstants.secret,
    });
  }

  async validate(payload: any) {
    반환 { userId: payload.sub, username: payload.username };
  }
}
@@switch
'passport-jwt'에서 { ExtractJwt, Strategy }를 임포트하고, '@nestjs/passport'에서 { PassportStrategy }를 임포트하고, '@nestjs/common'에서 { Injectable }을 임포트합니다;
import { jwtConstants } './constants'에서 임포트합니다;

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      비밀 또는 키: jwtConstants.secret,
```

```
        } );  
    }  
  
    async validate(payload) {  
        반환 { userId: payload.sub, username: payload.username };  
    }  
}
```

JwtStrategy에서는 모든 패스포트 전략에 대해 앞서 설명한 것과 동일한 레시피를 따랐습니다. 이 전략은 약간의 초기화가 필요하므로 `super()` 호출에서 옵션 객체를 전달하여 초기화합니다. 사용 가능한 옵션에 대한 자세한 내용은 [여기에서](#) 확인할 수 있습니다. 저희의 경우 옵션은 다음과 같습니다:

- `jwtFromRequest`: 요청에서 JWT를 추출하는 방법을 제공합니다. 저희는 API 요청의 인증 헤더에 무기명 토큰을 제공하는 표준 방식을 사용합니다. 다른 옵션은 [여기에](#) 설명되어 있습니다.
- `무시 만료`: 명확히 하기 위해 기본값인 `거짓` 설정을 선택했는데, 이 설정은 JWT가 만료되지 않았는지 확인하는 책임을 패스포트 모듈에 위임합니다. 즉, 만료된 JWT가 경로에 제공되면 요청이 거부되고 [401 권한 없음](#) 응답이 전송됩니다. Passport는 이 작업을 자동으로 처리해 줍니다.
- `secretOrKey`: 저희는 토큰 서명을 위해 대칭형 비밀을 제공하는 편리한 옵션을 사용하고 있습니다. PEM 인코딩된 공개 키와 같은 다른 옵션이 프로덕션 앱에 더 적합할 수 있습니다(자세한 내용은 [여기를 참조하세요](#)). 어떤 경우든 앞서 주의한 대로 이 비밀을 공개적으로 노출하지 마세요.

[유효성 검사\(\)](#) 메서드에 대해 설명할 필요가 있습니다. jwt 전략의 경우, Passport는 먼저 JWT의 서명을 확인하고 JSON을 디코딩합니다. 그런 다음 디코딩된 JSON을 단일 매개변수로 전달하는 `validate()` 메서드를 호출합니다. JWT 서명이 작동하는 방식에 따라 이전에 서명하고 유효한 사용자에게 발급한 유효한 토큰을 수신하고 있다는 것을 보장할 수 있습니다.

이 모든 작업의 결과로 `validate()` 콜백에 대한 응답은 간단합니다. `userId` 및 사용자 이름 프로퍼티가 포함된 객체를 반환하기만 하면 됩니다. 다시 한 번 기억해 두세요. Passport는 `validate()` 메서드의 반환값을 기반으로 [사용자](#) 객체를 빌드하고 이를 [요청](#) 객체에 프로퍼티로 첨부합니다.

또한 이 접근 방식은 프로세스에 다른 비즈니스 로직을 삽입할 수 있는 여지('후크')를 남긴다는 점도 지적할 가치가 있습니다. 예를 들어, [유효성 검사\(\)](#) 메서드에서 데이터베이스 조회를 수행하여 사용자에 대한 더 많은 정보를 추출하여 [요청에서](#) 더 풍부한 [사용자](#) 객체를 사용할 수 있게 할 수 있습니다. 또한 해지된 토큰 목록에서 `userId`를 조회하여 토큰 해지를 수행하는 등 추가적인 토큰 유효성 검사를 수행할 수도 있습니다. 여기 샘플 코드에서 구현한 모델은 빠른 "상태 비저장형 JWT" 모델로, 각 API 호출은 유효한 JWT의 존재 여부에 따라 즉시

승인되며 요청자에 대한 약간의 정보(사용자 ID 및 사용자 이름)를 요청 파이프라인에서 사용할 수 있습니다.

AuthModule에 새 JwtStrategy를 공급자로 추가합니다:

```
@@파일명(auth/auth.module)
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'./local.strategy'에서 { LocalStrategy }를 가져옵니다;
```

```
'./jwt.strategy'에서 { JwtStrategy }를 가져옵니다;
'./users/users.module'에서 { UsersModule }을 임포트하고
, '@nestjs/passport'에서 { PassportModule }을 임포트하고
, '@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { jwtConstants } './constants'에서 임포트합니다;

모듈({ import:
[
  UsersModule,
  PassportModule,
  JwtModule.register({
    비밀: jwtConstants.secret, signOptions:
    { expiresIn: '60s' },
  }),
],
공급자: [AuthService, LocalStrategy, JwtStrategy], 내보내기:
[AuthService],
})

내보내기 클래스 AuthModule {}

@@switch
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;

'./local.strategy'에서 { LocalStrategy }를 가져오고,
'./jwt.strategy'에서 { JwtStrategy }를 가져옵니다;
'./users/users.module'에서 { UsersModule }을 임포트하고
, '@nestjs/passport'에서 { PassportModule }을 임포트하고
, '@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { jwtConstants } './constants'에서 임포트합니다;

모듈({ import:
[
  UsersModule,
  PassportModule,
  JwtModule.register({
    비밀: jwtConstants.secret, signOptions:
    { expiresIn: '60s' },
  }),
],
공급자: [AuthService, LocalStrategy, JwtStrategy], 내보내기:
[AuthService],
})
```

내보내기 클래스 AuthModule {}

JWT에 서명할 때 사용한 것과 동일한 비밀번호를 가져옴으로써 Passport에서 수행하는 확인 단계와 AuthService에서 수행하는 서명 단계가 공통 비밀번호를 사용하도록 합니다.

마지막으로, 기본 제공 [AuthGuard](#)를 확장하는 [JwtAuthGuard](#) 클래스를 정의합니다:

```
@@파일명(auth/jwt-auth.guard)
'@nestjs/common'에서 { Injectable }을 임포트하고,
'@nestjs/passport'에서 { AuthGuard }를 임포트합니다
;
```

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

보호 경로 및 JWT 전략 가드 구현

이제 보호 경로와 관련 가드를 구현할 수 있습니다. `app.controller.ts` 파일을 열고 아래와 같이 업데이트합니다:

@@파일명(앱.컨트롤러)

'@nestjs/common'에서 { Controller, Get, Request, Post, UseGuards }를 임포트합니다;

'./auth/jwt-auth.guard'에서 { JwtAuthGuard }를 가져오고,
 './auth/local-auth.guard'에서 { LocalAuthGuard }를 가져오고,
 './auth/auth.service'에서 { AuthService }를 가져오세요;

컨트롤러()

내보내기 클래스 AppController {
 constructor(private authService: AuthService) {}

사용가드(LocalAuthGuard)

@Post('auth/login')
 async login(@Request() req) {
 this.authService.login(req.user)을 반환합니다;
 }

UseGuards(JwtAuthGuard)

@Get('profile')
 getProfile(@Request() req) {
 req.user를 반환합니다;
 }
}

@@switch

'@nestjs/common'에서 { 컨트롤러, 종속성, 바인드, 가져오기, 요청, 게시, 사용가드 }를 임포트합니다;

'./auth/jwt-auth.guard'에서 { JwtAuthGuard }를 가져오고,

'./auth/local-auth.guard'에서 { LocalAuthGuard }를 가져오고,

'./auth/auth.service'에서 { AuthService }를 가져오세요;

```
종속성(AuthService)
@Controller()
export class AppController {
  constructor(authService) {
    this.authService = authService;
  }

  UseGuards(LocalAuthGuard)
  @Post('auth/login')
  @Bind(Request())
  async login(req) {
```

```

    this.authService.login(req.user)을 반환합니다;
}

UseGuards(JwtAuthGuard)
@Get('profile')
@Bind(Request())
getProfile(req) {
    req.user를 반환합니다;
}
}

```

다시 한 번, passport-jwt 모듈을 구성할 때 `@nestjs/passport` 모듈이 자동으로 프로비저닝한 `AuthGuard`를 적용하고 있습니다. 이 가드는 기본 이름인 `jwt`로 참조됩니다. `GET /profile` 경로가 호출되면 가드가 자동으로 `passport-jwt` 사용자 지정 구성 전략을 호출하고 JWT의 유효성을 검사한 후 `사용자` 속성을 `요청` 객체에 할당합니다.

앱이 실행 중인지 확인하고 `cURL`을 사용하여 경로를 테스트합니다.

```

$ # GET /profile
curl http://localhost:3000/profile
결과 -> {"statusCode":401,"message":"승인되지 않음"}

$ # POST /auth/login
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
$ # 결과 ->
{"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2Vybm..."}

```

이전 단계에서 무기명 코드로 반환된 `access_token`을 사용하여 `/profile`을 `GET`합니다.

```

curl http://localhost:3000/profile -H "인증: 무기명
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJlc2Vybm...
$ # result -> {"userId":1,"username":"john"}

```

`인증 모듈에서` `JWT`의 만료 시간을 `60초` 구성했습니다. 이는 너무 짧은 만료일 수 있으며 토큰 만료 및 새로 고침에 대한 세부 사항을 다루는 것은 이 문서의 범위를 벗어납니다. 하지만 `JWT`의 중요한 특성과 패스포트-`JWT` 전략을 설명하기 위해 이 방법을 선택했습니다. 인증 후 60초를 기다렸다가 `GET /프로필` 요청을 시도하면 `401 권한 없음` 응답을 받게 됩니다. 이는 `Passport`가 자동으로 `JWT`의 만료 시간을 확인하므로 애플리케이션에서 직접 확인해야 하는 수고를 덜어주기 때문입니다.

이제 `JWT` 인증 구현이 완료되었습니다. 이제 자바스크립트 클라이언트(예: `Angular/React/Vue`) 및 기타 자바스크립트 앱이 트위터 API 서버와 안전하게 인증하고 통신할 수 있습니다.

가드 확장

대부분의 경우 제공된 `AuthGuard` 클래스를 사용하는 것으로 충분합니다. 그러나 기본 오류 처리 또는 인증 로직을 간단히 확장하려는 사용 사례가 있을 수 있습니다. 이를 위해 다음을 확장할 수 있습니다.

를 사용하여 내장 클래스와 하위 클래스 내의 메서드를 재정의할 수 있습니다.

```
import {
  ExecutionContext,
  Injectable,
  UnauthorizedException,
}를 '@nestjs/common'에서 가져옵니다;
'@nestjs/passport'에서 { AuthGuard }를 임포트합니다;

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  canActivate(context: ExecutionContext) {
    // 여기에 사용자 지정 인증 로직 추가
    // 예를 들어, super.logIn(request)를 호출하여 세션을 설정합니다. 반환
    super.canActivate(context);
  }

  handleRequest(err, user, info) {
    // "info" 또는 "err" 인자를 기반으로 예외를 던질 수 있습니다.
    if (err || !user) {
      throw err || new UnauthorizedException();
    }
    사용자를 반환합니다;
  }
}
```

기본 오류 처리 및 인증 로직을 확장하는 것 외에도 인증이 일련의 전략을 거치도록 허용할 수 있습니다. 첫 번째 전략이 성공, 리디렉션 또는 오류를 일으키면 체인이 중단됩니다.

인증 실패는 각 전략을 통해 연쇄적으로 진행되며, 모든 전략이 실패하면 최종적으로 인증이 실패합니다.

```
export class JwtAuthGuard extends AuthGuard(['strategy_jwt_1',
  'strategy_jwt_2', '...']) { ... }
```

전 세계적으로 인증 사용

대부분의 엔드포인트를 기본적으로 보호해야 하는 경우, 인증 가드를 [전역 가드로](#) 등록하고 각 컨트롤러 위에 `@UseGuards()` 데코레이터를 사용하는 대신 어떤 경로를 공개해야 하는지 플래그를 지정하면 됩니다.

먼저, 모듈에 관계없이 다음 구성을 사용하여 `JwtAuthGuard`를 전역 가드로 등록합니다:

```
제공자: [
  {
    제공: APP_GUARD,
    useClass: JwtAuthGuard,
```

```
},  
],
```

이렇게 하면 Nest는 자동으로 모든 엔드포인트에 `JwtAuthGuard`를 바인딩합니다.

이제 경로를 공개로 선언하는 메커니즘을 제공해야 합니다. 이를 위해 `SetMetadata` 데코레이터 팩토리 함수를 사용하여 사용자 정의 데코레이터를 만들 수 있습니다.

```
import { SetMetadata } from '@nestjs/common';  
  
export const IS_PUBLIC_KEY = 'isPublic';  
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

위 파일에서 두 개의 상수를 내보냈습니다. 하나는 `IS_PUBLIC_KEY`라는 메타데이터 키이고, 다른 하나는 `Public`이라고 부를 새 데코레이터 자체입니다(프로젝트에 맞는 다른 이름을 지정할 수 있습니다).

이제 사용자 정의 `@Public()` 데코레이터가 생겼으므로 다음과 같이 모든 메서드를 데코레이션하는 데 사용할 수 있습니다:

```
@Public()  
@Get()  
findAll() {  
  반환 [];  
}
```

마지막으로, "isPublic" 메타데이터가 발견될 때 참을 반환하도록 `JwtAuthGuard`가 필요합니다. 이를 위해 `Reflector` 클래스를 사용하겠습니다(자세한 내용은 [여기](#)를 참조하세요).

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
    const isPublic = this.reflector.getAllAndOverride<boolean>
    (IS_PUBLIC_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublic) { 참
      을 반환합니다;
    }
    반환 super.canActivate(context);
  }
}
```

요청 범위 전략

패스포트 API는 라이브러리의 글로벌 인스턴스에 전략을 등록하는 것을 기반으로 합니다. 따라서 전략은 요청 종 속 옵션을 갖거나 요청별로 동적으로 인스턴스화되도록 설계되지 않았습니다(요청 범위 제공자에 대한 자세한 내용을 참조하세요). 전략을 요청 범위로 구성하면 Nest는 특정 경로에 묶이지 않기 때문에 전략을 인스턴스화하지 않습니다. 요청별로 어떤 "요청 범위" 전략을 실행해야 하는지 물리적으로 결정할 수 있는 방법은 없습니다.

하지만 전략 내에서 요청 범위가 지정된 공급자를 동적으로 해결할 수 있는 방법이 있습니다. 이를 위해 [모듈 참조](#) 기능을 활용합니다.

먼저 `local.strategy.ts` 파일을 열고 일반적인 방법으로 `ModuleRef`를 삽입합니다:

```
constructor(private moduleRef: ModuleRef) {
  super({
    passReqToCallback: true,
  });
}
```

정보 힌트 `ModuleRef` 클래스는 `@nestjs/core` 패키지에서 가져옵니다.

위와 같이 `passReqToCallback` 구성 속성을 `true`로 설정해야 합니다.

다음 단계에서는 새 컨텍스트 식별자를 생성하는 대신 요청 인스턴스를 사용하여 현재 컨텍스트 식별자를 가져옵니다([여기에서](#) 요청 컨텍스트에 대해 자세히 읽어보세요).

이제 `LocalStrategy` 클래스의 `validate()` 메서드 내에서 `ContextIdFactory` 클래스의 `getByRequest()` 메서드를 사용하여 요청 객체를 기반으로 컨텍스트 ID를 생성한 다음 이를 `resolve()` 호출에 전달합니다 :

```
비동기 유효성 검사( 요청
  청: 요청, 사용자명: 문자열, 비밀번호: 문자열,
) {
  const contextId = ContextIdFactory.getByRequest(request);
  // "AuthService"는 요청 범위가 지정된 제공자입니다.
  const authService = await this.moduleRef.resolve(AuthService,
contextId);
```

위의 예제에서 `resolve()` 메서드는 `AuthService` 공급자의 요청 범위 인스턴스를 비동기적으로 반환합니다 (`AuthService`가 요청 범위 공급자로 표시되어 있다고 가정했습니다).

여권 사용자 지정

모든 표준 패스포트 사용자 지정 옵션은 [등록\(\)](#) 메서드를 사용하여 같은 방식으로 전달할 수 있습니다. 사용 가능한 옵션은 구현 중인 전략에 따라 다릅니다. 예를 들어

```
PassportModule.register({ 세션: true });
```

생성자에 옵션 객체를 전달하여 전략을 구성할 수도 있습니다. 로컬 전략의 경우 예를 들어 다음과 같이 전달할 수 있습니다:

```
constructor(private authService: AuthService) {
  super({
    usernameField: '이메일',
    passwordField: '비밀번호',
  });
}
```

숙소 이름은 공식 [패스포트 웹사이트에서](#) 확인하세요. 네이밍 전

략

전략을 구현할 때 [PassportStrategy](#) 함수에 두 번째 인수를 전달하여 전략의 이름을 지정할 수 있습니다. 이렇게 하지 않으면 각 전략에 기본 이름이 지정됩니다(예: jwt-전략의 경우 'jwt'):

내보내기 클래스 JwtStrategy는 PassportStrategy를 확장합니다([Strategy](#), 'myjwt').

그런 다음 [@UseGuards\(AuthGuard\('myjwt'\)\)](#) 같은 데코레이터를 통해 이를 참조합

니다. GraphQL

[GraphQL과](#) 함께 AuthGuard를 사용하려면 기본 제공 AuthGuard 클래스를 확장하고 [getRequest\(\)](#) 메서드를 재정의하세요.

```
@Injectable()
export class GqlAuthGuard extends AuthGuard('jwt') {
  getRequest(context: ExecutionContext) {
    const ctx = GqlExecutionContext.create(context);
    return ctx.getContext().req;
  }
}
```

그래프 쿼리 리졸버에서 현재 인증된 사용자를 가져오려면 `@CurrentUser()`를 정의하면 됩니다.

데코레이터:

```
'@nestjs/common'에서 { createParamDecorator, ExecutionContext }를 임포트하고  
, '@nestjs/graphql'에서 { GqlExecutionContext }를 임포트합니다;
```

```
export const CurrentUser = createParamDecorator( (데이  
터: 알 수 없음, 컨텍스트: 실행 컨텍스트) => {  
    const ctx = GqlExecutionContext.create(context);  
    return ctx.getContext().req.user;  
},  
);
```

리졸버에서 위의 데코레이터를 사용하려면 쿼리 또는 변이의 매개 변수로 포함해야 합니다:

```
쿼리(반환 => 사용자) @사용가드(GqlAuthGuard)  
whoAmI(@CurrentUser() 사용자: User) {  
    this.usersService.findById(user.id)를 반환합니다;  
}
```

핫 리로드

애플리케이션의 부트스트랩 프로세스에 가장 큰 영향을 미치는 것은 TypeScript 컴파일입니다. 다행히도 웹팩 HMR(핫 모듈 교체)을 사용하면 변경 사항이 발생할 때마다 전체 프로젝트를 다시 컴파일할 필요가 없습니다. 따라서 애플리케이션을 인스턴스화하는 데 필요한 시간이 크게 줄어들고 반복 개발이 훨씬 쉬워집니다.

경고 웹팩은 에셋(예: `graphql` 파일)을 `dist` 폴더에 자동으로 복사하지 않습니다. 마찬가지로 웹팩은 글로브 정적 경로(예: `TypeOrmModule`의 `entities` 속성)와 호환되지 않습니다.

CLI 사용

Nest CLI를 사용하는 경우 구성 프로세스는 매우 간단합니다. CLI는 웹팩을 래핑하여 `HotModuleReplacementPlugin`을 사용할 수 있도록 합니다.

설치

먼저 필요한 패키지를 설치합니다:

```
$ npm i --save-dev 웹팩-노드-외부 실행 스크립트 웹팩 플러그인 웹팩
```

정보 힌트 클래식 양이 아닌 얀 베리를 사용하는 경우 웹팩-노드-외부 패키지를 웹팩-node-외부 대신 설치하세요.

구성

설치가 완료되면 애플리케이션의 루트 디렉터리에 `webpack-hmr.config.js` 파일을 생성합니다.

```
const nodeExternals = require('webpack-node-externals');
const { RunScriptWebpackPlugin } = require('run-script-webpack-plugin');

module.exports = 함수 (옵션, 웹팩) { 반환 {
    ...옵션,
    항목: ['webpack/hot/poll?100', options.entry], 외부: [
        nodeExternals({
            허용 목록: ['webpack/hot/poll?100'],
        }),
    ],
    플러그인: [
        ...options.plugins,
        새로운 웹팩.핫모듈교체플러그인(), 새로운 웹팩.위치
        무시플러그인({
            경로: [/\.js$/, /\.d\.ts$/],
        })
    ],
}
```

```

    },
    새로운 런스크립트웹팩 플러그인({ 이름: options.output.filename, 자동 재시작:
false }),
],
};

}
;
}
;
```

정보 힌트 Yaml 베리(클래식 Yaml이 아님)의 경우, 외부 구성 속성의 `nodeExternals`를 사용하는

대신 `webpack-pnp-externals` 패키지의 `WebpackPnpExternals`를 사용하세요:

`WebpackPnpExternals({{ '{' }} exclude: ['webpack/hot/poll? 100'] {{ '}' }}).`

이 함수는 기본 웹팩 구성이 포함된 원본 객체를 첫 번째 인수로, Nest CLI에서 사용하는 기본 웹팩 패키지에 대한 참조를 두 번째 인수로 받습니다. 또한 이 함수는 `HotModuleReplacementPlugin`, `WatchIgnorePlugin` 및 `RunScriptWebpackPlugin` 플러그인을 사용하여 수정된 웹팩構성을 반환합니다.

핫 모듈 교체

HMR을 활성화하려면 애플리케이션 입력 파일(`main.ts`)을 열고 다음 웹팩 관련 지침을 추가합니다:

```

선언하다 const module: any;

async 함수 bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}

부트스트랩();
```

실행 프로세스를 간소화하려면 `package.json` 파일에 스크립트를 추가하세요.

```
"start:dev": "nest build --webpack --webpackPath webpack-hmr.config.js --watch"
```

이제 명령줄을 열고 다음 명령을 실행하기만 하면 됩니다:

```
$ npm 실행 시작:dev
```

CLI 사용 안 함

Nest CLI를 사용하지 않는 경우 구성이 약간 더 복잡해집니다(수동 단계가 더 필요함).

설치

먼저 필요한 패키지를 설치합니다:

```
$ npm i --save-dev 웹팩 웹팩-cli 웹팩-노드-외부 ts-로더 실행 스크립트-웹팩-플러그  
인
```

정보 힌트 클래식 양이 아닌 얀 베리를 사용하는 경우 웹팩-노드-외부 패키지를 웹팩-node-외부 대신 설치하세요.

구성

설치가 완료되면 애플리케이션의 루트 디렉터리에 `webpack.config.js` 파일을 생성합니다.

```
const webpack = require('webpack');
const path = require('path');
const nodeExternals = require('webpack-node-externals');
const { RunScriptWebpackPlugin } = require('run-script-webpack-plugin');

module.exports = {
  항목: ['webpack/hot/poll?100', './src/main.ts'], target:
  'node',
  외부: [
    nodeExternals({
      허용 목록: ['webpack/hot/poll?100'],
    }),
  ],
  모듈: { rules:
    [
      {
        test: /\.tsx?$/, 사용:
        용: 'ts-loader',
        제외합니다: /node_modules/,
      },
    ],
  },
  모드: '개발', 해결: {
    확장자: ['.tsx', '.ts', '.js'],
  },
  플러그인: [
    새로운 웹팩.핫모듈교체플러그인(),
    새로운 런스크립트웹팩 플러그인({ 이름: 'server.js', 자동재시작: false }),
  ],
  출력합니다: {
  }
}
```

```
경로: path.join(__dirname, 'dist'),
파일명: 'server.js',
},
};
```

정보 힌트 Yaml 베리(클래식 Yaml이 아님)의 경우, 외부 구성 속성의 `nodeExternals`를 사용하는

대신 `webpack-pnp-externals` 패키지의 `WebpackPnpExternals`를 사용하세요:

```
WebpackPnpExternals({{ '{
'}} exclude: ['webpack/hot/poll? 100'] {{ '}' }}).
```

이 설정은 엔트리 파일의 위치, 컴파일된 파일을 저장하는 데 사용할 디렉토리, 소스 파일을 컴파일하는 데 사용할
로더의 종류 등 애플리케이션에 대한 몇 가지 필수 사항을 웹팩에 알려줍니다. 일반적으로 모든 옵션을 완전히 이해하지
못하더라도 이 파일을 그대로 사용할 수 있습니다.

핫 모듈 교체

HMR을 활성화하려면 애플리케이션 입력 파일(`main.ts`)을 열고 다음 웹팩 관련 지침을 추가합니다:

```
선언하다 const module: any;

async 함수 bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}

부트스트랩();
```

실행 프로세스를 간소화하려면 `package.json` 파일에 스크립트를 추가하세요.

```
"start:dev": "webpack --config webpack.config.js --watch"
```

이제 명령줄을 열고 다음 명령을 실행하기만 하면 됩니다:

```
$ npm 실행 시작:dev
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

MikroORM

이 레시피는 Nest에서 MikroORM을 시작하는 사용자를 돋기 위해 마련되었습니다. MikroORM은 데이터 매퍼, 작업 단위 및 ID 맵 패턴을 기반으로 하는 Node.js용 TypeScript ORM입니다. TypeORM을 대체할 수 있는 훌륭한 대안이며 TypeORM에서 마이그레이션하는 것은 매우 쉽습니다. MikroORM에 대한 전체 문서는 [여기에서 확인할 수 있습니다.](#)

정보 @mikro-orm/nestjs는 타사 패키지이며 NestJS 코어 팀에서 관리하지 않습니다. 라이브러리 와 관련된 문제가 발견되면 [해당 리포지토리에](#) 보고해 주세요.

설치

Nest에 MikroORM을 통합하는 가장 쉬운 방법은 [@mikro-orm/nestjs 모듈을](#) 사용하는 것입니다. Nest, MikroORM 및 기본 드라이버 옆에 설치하기만 하면 됩니다:

```
$ npm i @mikro-orm/core @mikro-orm/nestjs @mikro-orm/mysql # for mysql/mariadb
```

MikroORM은 [포스트그레스](#), [sqlite](#), [몽고도](#) 지원합니다. 모든 드라이버에 대한 [공식 문서를 참조하세요](#).

설치 프로세스가 완료되면 [MikroOrmModule](#)을 루트 앱모듈로 가져올 수 있습니다.

```
모듈({ import: [
  MikroOrmModule.forRoot({
    entities: ['./dist/entities'],
    entitiesTs: ['./src/entities'],
    dbName: 'my-db-name.sqlite3',
    type: 'sqlite',
  }),
],
컨트롤러: [AppController], 공급자: [
  앱서비스],
})
내보내기 클래스 AppModule {}
```

[forRoot\(\)](#) 메서드는 MikroORM 패키지의 [init\(\)](#)과 동일한 구성 객체를 받습니다. 전체 구성 설명서는 [이 페이지에서 확인하세요](#).

또는 구성 파일 `mikro-orm.config.ts`를 생성하여 [CLI를 구성한](#) 다음 인자 없이 `forRoot()`를 호출할 수 있습니다. 트리 쉐이킹을 사용하는 빌드 툴을 사용하는 경우에는 이 방법이 작동하지 않습니다.

```
모듈({ import: [
  MikroOrmModule.forRoot(),
```

```
],
...
})
내보내기 클래스 AppModule {}
```

그 후에는 다른 곳에서 모듈을 임포트하지 않고도 전체 프로젝트에 `EntityManager`를 삽입할 수 있습니다.

```
'@mikro-orm/core'에서 { MikroORM }을 가져옵니다;
// 드라이버 패키지에서 EntityManager를 가져오거나 `@mikro-orm/knex`에서 {
EntityManager }를 가져옵니다;

@Injectable()
내보내기 클래스 MyService { 생성
자(
  비공개 읽기 전용ORM: 비공개 읽기 전용 em:
  EntityManager,
)
정보 EntityManager는 @mikro-orm/driver 패키지에서 가져오는데, 여기서 driver는 mysql,
sqlite, postgres 또는 사용 중인 드라이버입니다. 종속성으로 @mikro-orm/knex가 설치되어 있는
경우, 거기에서 EntityManager를 가져올 수도 있습니다.
```

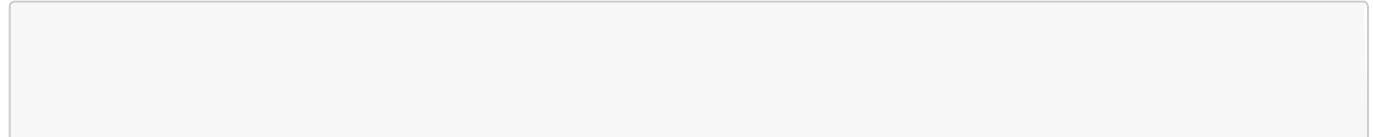
리포지토리

MikroORM은 리포지토리 디자인 패턴을 지원합니다. 모든 엔티티에 대해 리포지토리를 만들 수 있습니다. [여기에서](#) 리포지토리에 대한 전체 문서를 읽어보세요. 현재 범위에 등록할 리포지토리를 정의하려면 `forFeature()` 메서드를 사용할 수 있습니다. 예를 들면 다음과 같습니다:

정보 기본 엔티티는 리포지토리가 없으므로 `forFeature()`를 통해 등록해서는 안 됩니다. 반면에 기본 엔티티는 `forRoot()`(또는 일반적으로 ORM 구성)의 목록에 포함되어야 합니다.

```
// photo.module.ts
@Module({
  임포트합니다: [MikroOrmModule.forFeature([Photo])], 제공자:
  [PhotoService],
  컨트롤러: [포토컨트롤러],
})
내보내기 클래스 PhotoModule {}
```

를 생성하고 루트 `AppModule`로 가져옵니다:



```
    임포트: [MikroOrmModule.forRoot(...), PhotoModule],
  })
내보내기 클래스 AppModule {}
```

이런 식으로 `@InjectRepository()` 메서드를 사용하여 `PhotoService`에 `PhotoRepository`를 주입할 수 있습니다.

데코레이터:

```
@Injectable()
내보내기 클래스 PhotoService {
  생성자(
    인젝트 리포지토리(사진)
    비공개 읽기 전용 사진 저장소: 엔티티저장소<사진>,
  ) {}
}
```

사용자 지정 리포지토리 사용

사용자 정의 리포지토리를 사용할 때 리포지토리의 이름을 `getRepositoryToken()` 메서드와 같은 방식으로 지정하면 `@InjectRepository()` 데코레이터를 사용하지 않아도 됩니다:

```
export const getRepositoryToken = <T>(엔티티: 엔티티이름<T>) => =>
` ${Utils.className(entity)}Repository`;
```

즉, 리포지토리의 이름을 엔티티의 이름과 동일하게 지정하고 `Repository`

접미사를 추가하면 리포지토리가 Nest DI 컨테이너에 자동으로 등록됩니다.

```
// `**./author.entity.ts**`
@Entity()
내보내기 클래스 Author {
  // 에서 추론할 수 있도록 `em.getRepository()` [EntityRepositoryType]?
  AuthorRepository;
}

// `**./author.repository.ts**`
@Repository(저자)
내보내기 클래스 AuthorRepository extends EntityRepository<Author> {
  @Injectable() 사용자 지정 메소드...
}

내보내기 클래스 MyService {
```

사용자 지정 리포지토리 이름이 `getRepositoryToken()`이 반환하는 이름과 동일하므로 더 이상

`@InjectRepository()` 데코레이터가 필요하지 않습니다:

```
생성자(비공개 읽기 전용 저장소: AuthorRepository) {}

}
```

엔티티 자동 로드

정보 자동로드 엔티티 옵션이 v4.1.0에 추가되었습니다.

연결 옵션의 엔티티 배열에 엔티티를 수동으로 추가하는 작업은 번거로울 수 있습니다. 또한 루트 모듈에서 엔티티를 참조하면 애플리케이션 도메인 경계가 깨지고 애플리케이션의 다른 부분으로 구현 세부 정보가 유출될 수 있습니다. 이 문제를 해결하기 위해 정적 글로브 경로를 사용할 수 있습니다.

그러나 글로브 경로는 웹팩에서 지원되지 않으므로 모노레포 내에서 애플리케이션을 빌드하는 경우 사용할 수 없습니다. 이 문제를 해결하기 위해 대체 솔루션이 제공됩니다. 엔티티를 자동으로 로드하려면 아래 그림과 같이

구성 객체(`forRoot()` 메서드에 전달됨)의 `autoLoadEntities` 속성을 `true`로 설정합니다:

```
모듈({ import: [
    MikroOrmModule.forRoot({
        ...
        autoLoadEntities: true,
    }),
],
})

내보내기 클래스 AppModule {}
```

이 옵션을 지정하면 `forFeature()` 메서드를 통해 등록된 모든 엔티티가 구성 개체의 엔티티 배열에 자동으로 추가됩니다.

정보 정보 `forFeature()` 메서드를 통해 등록되지 않고 (관계를 통해서만) 엔티티에서 참조되는 엔티티는 **자동 로드 엔티티** 설정을 통해 포함되지 않습니다.

정보 정보 **자동 로드** 엔티티를 사용해도 MikroORM CLI에는 영향을 미치지 않습니다. 여전히 전체 엔티티 목록이 포함된 CLI 구성이 필요하기 때문입니다. 반면에 CLI가 웹팩을 거치지 않으므로 글로브를 사용할 수 있습니다.

직렬화

경고 참고 MikroORM은 더 나은 유형 안전성을 제공하기 위해 모든 단일 엔티티 관계를 참조<T> 또는 컬렉션<T> 객체로 래핑합니다. 이렇게 하면 Nest의 내장 직렬화기가 래핑된 관계에 대해 블라인드 처리됩니다. 다시 말해, HTTP 또는 웹소켓 핸들러에서 MikroORM 엔티티를 반환하는 경우, 해당 엔티티의 모든 관계가 직렬화되지 않습니다.

다행히도 MikroORM은 다음을 대신하여 사용할 수 있는 직렬화 API를 제공합니다.

[ClassSerializerInterceptor](#).

엔티티()

```
내보내기 클래스 Book {
    @Property({ hidden: true }) // 클래스 트랜스포머와 동일합니다.
    `@제외`
    숨겨진 필드 = Date.now();

    @Property({ persist: false }) // 클래스 트랜스포머와 유사합니다.
    `@Expose()`. 메모리에만 존재하며 직렬화됩니다. count?: 숫자;

    @ManyToOne({
        serializer: (value) => value.name,
        serializedName: '작성자 이름',
    }) // 클래스 트랜스포머의 `@Transform()` 작성자와 동일합니다:
    Author;
}
```

대기열의 범위 지정 핸들러 요청

v4.1.0에 `@UseRequestContext()` 데코레이터가 추가되었습니다.

문서에서 언급했듯이 각 요청에 대해 깨끗한 상태가 필요합니다. 이 작업은 미들웨어를 통해 등록된 `RequestContext` 헬퍼 덕분에 자동으로 처리됩니다.

하지만 미들웨어는 일반 HTTP 요청 처리에 대해서만 실행되는데, 그 외의 요청 범위 메서드가 필요하다면 어떻게 해야 할까요? 한 가지 예로 큐 핸들러나 예약된 작업을 들 수 있습니다.

사용 요청 컨텍스트() 데코레이터를 사용할 수 있습니다. 이 데코레이터를 사용하려면 먼저 현재 컨텍스트에 `MikroORM` 인스턴스를 주입해야 하며, 그 다음 이 인스턴스를 사용하여 컨텍스트를 생성합니다. 내부적으로 데코레이터는 메서드에 대한 새 요청 컨텍스트를 등록하고 컨텍스트 내에서 메서드를 실행합니다.

@Injectable()

```
내보내기 클래스 MyService {
    constructor(private readonly orm: MikroORM) {}

    @UseRequestContext()
    async doSomething() {
        // 별도의 컨텍스트에서 실행됩니다.
    }
}
```

요청 컨텍스트에 **AsyncLocalStorage** 사용

기본적으로 **도메인** API는 **RequestContext** 헬퍼에서 사용됩니다. 최신 노드 버전을 사용 중이라면 `@mikro-orm/core@4.0.3` 이후 새로운 **AsyncLocalStorage**도 사용할 수 있습니다:

```
// 새 (글로벌) 스토리지 인스턴스 생성
const storage = new AsyncLocalStorage<EntityManager>();

모듈({ import: [
  MikroOrmModule.forRoot({
    // ...
    registerRequestContext: false, // 자동 미들웨어 비활성화 context: () =>
    storage.getStore(), // AsyncLocalStorage 사용
  },
  ],
  컨트롤러: [AppController], 공급자: [
    앱서비스],
})
내보내기 클래스 AppModule {}}

// 요청 컨텍스트 미들웨어 등록
const app = await NestFactory.create(AppModule, { ... });
const orm = app.get(MikroORM);

app.use((req, res, next) => {
  storage.run(orm.em.fork(true, true), next);
});
```

테스트

`mikro-orm/nestjs` 패키지는 리포지토리를 모킹할 수 있도록 주어진 엔티티를 기반으로 준비된 토큰을 반환하는 `getRepositoryToken()` 함수를 노출합니다.

```
모듈({ providers: [
  {
    포토서비스,
    {
      제공: getRepositoryToken(사진), 사용값
        : 모의 저장소,
    },
  ],
})
```

내보내기 클래스 PhotoModule {}

예

MikroORM을 사용한 NestJS의 실제 예제는 [여기에서](#) 확인할 수 있습니다.

SQL(TypeORM)

이 장은 TypeScript에만 적용됩니다.

경고 이 글에서는 사용자 정의 공급자 메커니즘을 사용하여 TypeORM 패키지를 기반으로 데이터베이스 모듈을 처음부터 만드는 방법을 배웁니다. 결과적으로 이 솔루션에는 바로 사용할 수 있고 즉시 사용 가능한 전용 [@nestjs/typeorm](#) 패키지를 사용하면 생략할 수 있는 많은 오버헤드가 포함되어 있습니다. 자세한 내용은 [여기를 참조하세요](#).

TypeORM은 node.js 세계에서 사용할 수 있는 가장 성숙한 객체 관계형 매퍼(ORM)입니다. TypeScript로 작성되었기 때문에 Nest 프레임워크와 매우 잘 작동합니다.

시작하기

이 라이브러리로 모험을 시작하려면 필요한 모든 종속 요소를 설치해야 합니다:

```
$ npm install --save typeorm mysql2
```

가장 먼저 해야 할 일은 [typeorm](#) 패키지에서 가져온 새로운 `DataSource().initialize()` 클래스를 사용하여 데이터베이스와의 연결을 설정하는 것입니다. `초기화()` 함수는 [프로미스](#)를 반환하므로 [비동기 프로바이더](#)를 만들어야 합니다.

@@파일명(database.providers) 'typeorm'

에서 { DataSource }를 가져옵니다;

```
export const databaseProviders = [
  {
    제공: 'DATA_SOURCE',
    useFactory: async () => {
      const dataSource = new DataSource({
        type: 'mysql',
        호스트: 'localhost',
        포트: 3306, 사용자 이
        름: 'root', 비밀번호:
        'root', 데이터베이스:
        'test', entities:
        [
          __dirname + '/../**/*.{entity{.ts,.js}}',
        ],
        동기화: true,
      });
    },
  },
];
```

데이터소스 초기화()를 반환합니다;

경고 **동기화** 설정: true는 프로덕션 환경에서 사용해서는 안 됩니다. 그렇지 않으면 프로덕션 데이터가 손실될 수 있습니다.

정보 힌트 모범 사례에 따라 사용자 정의 공급자를 분리된 파일에 선언했습니다.

*.providers.ts 접미사.

그런 다음 나머지 애플리케이션에서 액세스할 수 있도록 이러한 공급자를 내보내야 합니다.

@@파일명(데이터베이스.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./database.providers'에서 { databaseProviders }를 가져옵니다;
```

모듈({

```
제공자: [...데이터베이스 제공자], 내보내기:  
[...databaseProviders],  
})
```

데이터베이스 모듈 **클래스** {} 내보내기

이제 **@Inject()** 데코레이터를 사용하여 **DATA_SOURCE** 객체를 주입할 수 있습니다. **DATA_SOURCE** 비동기 프로바이더에 의존하는 각 클래스는 **프로미스가** 해결될 때까지 기다립니다.

리포지토리 패턴

TypeORM은 리포지토리 디자인 패턴을 지원하므로 각 엔티티에는 자체 리포지토리가 있습니다. 이러한 리포지토리는 데이터베이스 연결에서 얻을 수 있습니다.

하지만 먼저 엔티티가 하나 이상 필요합니다. 공식 문서에 있는 **사진** 엔티티를 재사용하겠습니다.

@@파일명 (사진.엔티티)

'typeorm'에서 { Entity, Column, PrimaryGeneratedColumn }을 가져옵니다;

엔티티()

```
export class Photo {  
    @PrimaryGeneratedColumn()  
    id: number;  
  
    @Column({ length: 500 })  
    name: 문자열;
```

열('text') 설명: 문자열

입니다;

@Column() 파일명:

문자열;

열('int') 보기:

숫자;

@Column()

```
    isPublished: 부울입니다;
}
```

사진 엔티티는 사진 딕렉터리에 속합니다. 이 딕렉토리는 PhotoModule을 나타냅니다. 이제 리포지토리 공급자를 만들어 보겠습니다:

```
@@파일명(사진.제공자)
'typeorm'에서 { DataSource } 가져오기; './사
진.entity'에서 { Photo } 가져오기;

export const photoProviders = [
{
    제공: '사진_저장소',
    사용 팩토리: (데이터 소스: 데이터 소스) => 데이터 소스.get 리포
지토리(사진),
    주입합니다: ['data_source'],
},
경고 경고 실제 애플리케이션에서는 마법의 문자열을 피해야 합니다. 둘 다
```

사진 저장소 및 데이터 소스는 별도의 constants.ts 파일에 보관해야 합니다.

이제 @Inject() 데코레이터를 사용하여 리포지토리<사진>을 PhotoService에 삽입할 수 있습니다:

```
@@파일명(사진.서비스)
'@nestjs/common'에서 { Injectable, Inject }를 가져오고,
'typeorm'에서 { Repository }를 가져옵니다;
'./사진.entity'에서 { 사진 }을 가져옵니다;

@Injectable()
내보내기 클래스 PhotoService { 생성자(
    @Inject('PHOTO_REPOSITORY')
    비공개 사진 저장소: 리포지토리<사진>,
) {}

비동기 findAll(): Promise<Photo[]> {
    return this.photoRepository.find();
}
}
```

데이터베이스 연결은 비동기식이지만 Nest는 이 프로세스를 최종 사용자에게 완전히 보이지 않게 합니다.

PhotoRepository는 데이터베이스 연결을 기다리고 있으며, **PhotoService**는 리포지토리를 사용할 준비가 될 때까지 지연됩니다. 각 클래스가 인스턴스화되면 전체 애플리케이션이 시작될 수 있습니다.

다음은 최종 **포토모듈입니다**:



@@파일명 (사진.모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./database/database.module'에서 { DatabaseModule } 가져오기, './  
사진 제공자'에서 { photoProviders } 가져오기;  
'./사진.서비스'에서 { PhotoService }를 가져옵니다;
```

모듈 ({

임포트: [데이터베이스 모듈], 공급

자: [

...사진 제공자, 사진 서비스,

],

)

내보내기 클래스 PhotoModule {}

정보 힌트 **포토모듈을 루트 앱모듈로 가져오는 것을 잊지 마세요.**

몽고DB(몽구스)

경고 이 문서에서는 사용자 정의 컴포넌트를 사용하여 몽구스 패키지를 기반으로 데이터베이스 모듈을 처음부터 만드는 방법을 알아봅니다. 결과적으로 이 솔루션에는 바로 사용할 수 있고 즉시 사용 가능한 전용 [@nestjs/mongoose](#) 패키지를 사용하면 생략할 수 있는 많은 오버헤드가 포함되어 있습니다. 자세한 내용은 [여기](#)를 참조하세요.

몽구스는 가장 널리 사용되는 [MongoDB](#) 객체 모델링 도구입니다.

시작하기

이 라이브러리로 모험을 시작하려면 필요한 모든 종속 요소를 설치해야 합니다:

```
$ npm install --save mongoose
```

가장 먼저 해야 할 일은 [connect\(\)](#) 함수를 사용하여 데이터베이스와의 연결을 설정하는 것입니다. [connect\(\)](#) 함수는 [Promise](#)를 반환하므로 [비동기 프로바이더](#)를 만들어야 합니다.

[@@파일명](#) (데이터베이스.제공자) '몽구스'에서 *

를 몽구스로 가져옵니다;

```
export const databaseProviders = [
  {
    제공: '데이터베이스_연결',
    useFactory: (): Promise<유형 몽구스> =>
      mongoose.connect('mongodb://localhost/nest'),
  },
];
```

[@@switch](#)

'몽구스'에서 *를 몽구스로 가져옵니다;

```
export const databaseProviders = [
  {
    제공: '데이터베이스_연결',
    사용 팩토리: () => 몽구스.연결('mongodb://localhost/nest'),
  },
];
```

[정보](#) 힌트 모범 사례에 따라 사용자 정의 공급자를 분리된 파일에 선언했습니다.

*.providers.ts 접미사.

그런 다음 이러한 공급자를 내보내서 애플리케이션의 나머지 부분에서 액세스할 수 있도록 해야 합니다.

@@파일명 (데이터베이스.모듈)

'@nestjs/common'에서 { Module }을 가져옵니다;

'./database.providers'에서 { databaseProviders }를 가져옵니다;

```
모듈({  
    제공자: [...데이터베이스 제공자], 내보내기:  
    [...databaseProviders],  
})  
데이터베이스 모듈 클래스 {} 내보내기
```

이제 `@Inject()` 데코레이터를 사용하여 `Connection` 객체를 주입할 수 있습니다. `Connection` 비동기 프로바이더에 종속되는 각 클래스는 `프로미스가` 해결될 때까지 기다립니다.

모델 주입

몽구스에서는 모든 것이 `스키마에서` 파생됩니다. `CatSchema`를 정의해 보겠습니다:

```
@@파일명(schemas/cat.schema)  
'mongoose'에서 mongoose로 *를 가져옵니다;  
  
export const CatSchema = new mongoose.Schema({  
    name: String,  
    나이: 숫자, 품종  
    : 문자열,  
});
```

`CatSchema`은 `cats` 디렉터리에 속합니다. 이 디렉토리는 `CatsModule`을 나타냅니다. 이제 모델 프로바이더를 생성할 차례입니다:

```
@@파일명(cats.providers)
'몽구스'에서 { Connection }을 가져옵니다;
'./schemas/cat.schema'에서 { CatSchema }를 가져옵니다;

export const catsProviders = [
  {
    제공: 'CAT_MODEL',
    사용 팩토리: (연결: 연결) => 연결.모델('Cat', CatSchema),
    주입합니다: ['데이터베이스_연결'],
  },
];
@@switch
'./schemas/cat.schema'에서 { CatSchema }를 가져옵니다;

export const catsProviders = [
  {
    제공: 'CAT_MODEL',
    useFactory: (connection) => connection.model('Cat', CatSchema),
    inject: ['database_connection'],
  },
];
```

경고 경고 실제 애플리케이션에서는 매직 문자열을 피해야 합니다. `CAT_MODEL`

및 `DATABASE_CONNECTION`은 별도의 `constants.ts` 파일에 보관해야 합니다.

이제 `@Inject()` 데코레이터를 사용하여 `CAT_MODEL`을 `CatsService`에 주입할 수 있습니다:

```
@@파일명(cats.service)
'몽구스'에서 { 모델 }을 가져옵니다;

'@nestjs/common'에서 { Injectable, Inject }를 임포트하고
, './interfaces/cat.interface'에서 { Cat }을 임포트하고,
'./dto/create-cat.dto'에서 { CreateCatDto}를 임포트합니다
;

@Injectable()
내보내기 클래스 CatsService { 생성
  자(
    @Inject('CAT_MODEL')
    private catModel: Model<Cat>,
  ) {}

  async create(createCatDto: CreateCatDto): Promise<Cat> {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  비동기 findAll(): Promise<Cat[]> {
    return this.catModel.find().exec();
  }
}
@@switch
'@nestjs/common'에서 { 주입 가능, 종속성 }을 가져옵니다;

주입 가능()
@Dependencies('CAT_MODEL')
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
  }

  async create(createCatDto) {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  async findAll() {
```

```
    이.catModel.find().exec()를 반환합니다;  
}  
}
```

위의 예에서는 **Cat** 인터페이스를 사용했습니다. 이 인터페이스는 몽구스 패키지에서 [문서를](#) 확장합니다:



'몽구스'에서 { 문서 }를 가져옵니다;

```
내보내기 인터페이스 Cat 확장 문서 { 읽기 전용 이  
름: 문자열;  
읽기 전용 나이: 숫자; 읽기 전용  
품종: 문자열;  
}
```

데이터베이스 연결은 비동기식이지만 Nest는 이 프로세스를 최종 사용자에게 완전히 보이지 않게 합니다.

`CatModel` 클래스는 데이터베이스 연결을 기다리고 있으며, `CatsService`는 모델을 사용할 준비가 될 때까지 지연됩니다. 각 클래스가 인스턴스화되면 전체 애플리케이션이 시작될 수 있습니다.

다음은 최종 `CatsModule`입니다:

```
@@파일명(cats.module)  
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./cats.controller'에서 { CatsController }를 임포트하  
고, './cats.service'에서 { CatsService }를 임포트하고,  
'./cats.providers'에서 { catsProviders }를 임포트합니다  
;  
'../database/database.module'에서 { DatabaseModule }을 가져옵니다;
```

모듈({

```
    임포트: [데이터베이스 모듈], 컨트롤러:  
        [CatsController], 제공자: [  
            CatsService,  
            ...고양이제공자,
```

정보 힌트 `CatsModule`을 루트 앱모듈로 가져오는 것을 잊지 마세요.
})

내보내기 클래스 CatsModule {}

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

SQL(시퀄라이즈)

이 장은 TypeScript에만 적용됩니다.

경고 이 글에서는 사용자 정의 컴포넌트를 사용하여 Sequelize 패키지를 기반으로 [DatabaseModule](#)을 처음부터 새로 만드는 방법을 알아봅니다. 결과적으로 이 기법에는 많은 오버헤드가 포함되어 있지만, 바로 사용할 수 있는 전용 [@nestjs/sequelize](#) 패키지를 사용하면 피할 수 있습니다. 자세한 내용은 [여기](#)를 참조하세요.

[시퀄라이즈](#)는 바닐라 자바스크립트로 작성된 인기 있는 객체 관계형 매퍼(ORM)이지만, 기본 시퀄라이즈에 데코레이터 세트와 기타 추가 기능을 제공하는 [시퀄라이즈](#) 타입스크립트 타입스크립트 래퍼가 있습니다.

시작하기

이 라이브러리로 모험을 시작하려면 다음 종속성을 설치해야 합니다:

```
$ npm install --save sequelize sequelize-typescript mysql2
$ npm install --save-dev @types/sequelize
```

가장 먼저 해야 할 일은 생성자에 옵션 객체를 전달하여 Sequelize 인스턴스를 생성하는 것입니다. 또한 모든 모델을 추가하고([모델](#) 경로 속성을 사용하는 대안도 있습니다) 데이터베이스 테이블을 [동기화\(\)](#)해야 합니다.

@@파일명 (데이터베이스.공급자)

'sequelize-typescript'에서 { Sequelize }를 임포트하고,
'./cats/cat.entity'에서 { Cat }을 임포트합니다;

```
export const databaseProviders = [  
  {  
    제공: 'SEQUELIZE',  
    useFactory: async () => {  
      const sequelize = new Sequelize({  
        dialect: 'mysql',  
        호스트: 'localhost',  
        포트: 3306, 사용자명:  
        'root', 비밀번호:  
        'password', 데이터베이  
        스: 'nest',  
      });  
      sequelize.addModels([Cat]);  
      await sequelize.sync();  
      return sequelize;  
    },  
  },  
];
```

정보 힌트 모범 사례에 따라 사용자 정의 공급자를 분리된 파일에 선언했습니다.

*.providers.ts 접미사.

그런 다음 이러한 공급자를 내보내서 애플리케이션의 나머지 부분에서 액세스할 수 있도록 해야 합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./database.providers'에서 { databaseProviders }를 가져옵니다;
```

```
모듈 ({  
  제공자: [...데이터베이스 제공자], 내보내기:  
  [...databaseProviders],  
})
```

데이터베이스 모듈 클래스 {} 내보내기

이제 `@Inject()` 데코레이터를 사용하여 `Sequelize` 객체를 주입할 수 있습니다. `Sequelize` 비동기 프로파일러에 종속되는 각 클래스는 `프로미스가` 해결될 때까지 기다립니다.

모델 주입

시퀄라이즈에서 모델은 데이터베이스의 테이블을 정의합니다. 이 클래스의 인스턴스는 데이터베이스 행을 나타냅니다. 먼저 엔티티가 하나 이상 필요합니다:

```
@@파일명(cat.entity)  
'sequelize-typescript'에서 { 테이블, 열, 모델 }을 가져옵니다;
```

테이블

내보내기 클래스 Cat extends Model {

`@Column`

이름: 문자열;

칼럼

나이: 숫자;

열 종류: 문자열;

}

`Cat` 엔티티는 `cats` 디렉토리에 속합니다. 이 디렉토리는 `CatsModule`을 나타냅니다. 이제 리포지토리 공급자를 생성할 차례입니다:

@@파일명(cats.providers)

'./cat.entity'에서 { Cat }을 가져옵니다;

```
export const catsProviders = [
```

```
{
```

```
    제공: 'CATS_REPOSITORY',
```

```
    useValue: Cat,
```

```
},  
];
```

경고 경고 실제 애플리케이션에서는 마법의 문자열을 피해야 합니다. 둘 다

CATS_REPOSITORY와 SEQUELIZE는 별도의 `constants.ts` 파일에 보관해야 합니다.

Sequelize에서는 정적 메서드를 사용하여 데이터를 조작하므로 여기에 별칭을 만들었습니다. 이제

`@Inject()` 데코레이터를 사용하여 CATS_REPOSITORY를 CatsService에 주입할 수 있습니다:

```
@@파일명(cats.service)  
'@nestjs/common'에서 { Injectable, Inject }를 임포트하고,  
'./dto/create-cat.dto'에서 { CreateCatDto }를 임포트하고,  
'./cat.entity'에서 { Cat }을 임포트합니다;
```

```
@Injectable()  
내보내기 클래스 CatsService { 생  
성자(  
    @Inject('CATS_REPOSITORY')  
    비공개 고양이저장소: 고양이 유형  
) {}
```

```
비동기 findAll(): Promise<Cat[]> {  
    this.catsRepository.findAll<Cat>()을 반환합니다;  
}
```

데이터베이스 연결은 비동기식이지만 Nest는 이 프로세스를 최종 사용자에게 완전히 보이지 않게 합니다. CATS_REPOSITORY 프로바이더는 데이터베이스 연결을 기다리고 있으며, CatsService는 리포지토리를 사용할 준비가 될 때까지 지연됩니다. 각 클래스가 인스턴스화되면 전체 애플리케이션이 시작될 수 있습니다.

다음은 최종 CatsModule입니다:

@@파일명 (cats.module)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./cats.controller'에서 { CatsController }를 임포트하  
고, './cats.service'에서 { CatsService }를 임포트하고,  
'./cats.providers'에서 { catsProviders }를 임포트합니다  
;  
'./database/database.module'에서 { DatabaseModule }을 가져옵니다;
```

모듈({

임포트: [데이터베이스 모듈], 컨트롤러:

[CatsController], 제공자: [

CatsService,

...고양이제공자,

],

)

내보내기 클래스 CatsModule {}

정보 힌트 `CatsModule`을 루트 앱모듈로 가져오는 것을 잊지 마세요.

라우터 모듈

정보 힌트 이 장은 HTTP 기반 애플리케이션에만 해당됩니다.

HTTP 애플리케이션(예: **REST** API)에서 핸들러의 경로 경로는 컨트롤러에 대해 선언된 (선택적) 접두사 (`@Controller` 데코레이터 내부)와 메서드의 데코레이터에 지정된 모든 경로(예: `@Get('users')`)를 연결하여 결정됩니다.) 이에 대한 자세한 내용은 [이 섹션에서](#) 확인할 수 있습니다.

또한 애플리케이션에 등록된 모든 경로에 [글로벌 접두사를 정의하거나 버전 관리를 활성화할 수 있습니다.](#)

또한 모듈 수준에서 접두사를 정의하는 것이(따라서 해당 모듈에 등록된 모든 컨트롤러에 대해) 유용할 수 있는 예지 케이스도 있습니다. 예를 들어 "대시보드"라는 애플리케이션의 특정 부분에서 사용되는 여러 가지 엔드포인트를 노출하는 **REST** 애플리케이션을 상상해 보세요. 이 경우 각 컨트롤러 내에서 `/dashboard` 접두사를 반복하는 대신 다음과 같이 [라우터 모듈](#)을 사용할 수 있습니다:

```
모듈({ import: [
  대시보드 모듈, 라우터 모듈.등록([
    {
      경로: '대시보드', 모듈: 대시
      보드모듈,
    },
    ],
  ],
},
내보내기 클래스 AppModule {}
```

정보 힌트 [라우터모듈](#) 클래스는 `@nestjs/core` 패키지에서 내보냅니다.

또한 계층 구조를 정의할 수 있습니다. 즉, 각 모듈은 [하위](#) 모듈을 가질 수 있습니다. 자식 모듈은 부모의 접두사를 상속합니다. 다음 예제에서는 [관리 모듈](#)을 `DashboardModule` 및 `MetricsModule`의 부모 모듈로 등록하겠습니다.

```
모듈({ import: [
  AdminModule,
  DashboardModule,
  MetricsModule,
  RouterModule.register([
    {
      경로: 'admin',
      module: 관리자 모듈, 아
      이들: [
        {
          경로: '대시보드', 모듈: 대시
          보드모듈,
        },
      ],
    },
  ],
}]);
```

```
{  
    경로: 'metrics',  
    module: MetricsModule,  
},  
],  
},  
])  
],  
});
```

정보 힌트 이 기능을 과도하게 사용하면 코드를 작성하기 어려울 수 있으므로 매우 신중하게 사용해야 합니다.
시간이 지나도 유지됩니다.

위의 예시에서 `DashboardModule` 내부에 등록된 모든 컨트롤러에는 추가

`/관리/대시보드` 접두사를 사용합니다(모듈은 경로를 위에서 아래로 - 재귀적으로 - 부모에서 자식으로 연결하므로). 마찬가지로, `MetricsModule` 내에 정의된 각 컨트롤러에는 모듈 수준 접두사 `/admin/metrics`가 추가됩니다.

건강 상태 확인(종료)

터미널 통합은 준비 상태/활력 상태 확인 기능을 제공합니다. 상태 점검은 복잡한 백엔드 설정에 있어 매우 중요합니다. 간단히 말해, 웹 개발 영역에서의 상태 점검은 일반적으로 <https://my-website.com/health/readiness> 같은 특수 주소로 구성됩니다. 서비스나 인프라의 구성 요소(예: Kubernetes)는 이 주소를 지속적으로 확인합니다. 이 주소에 대한 GET 요청에서 반환된 HTTP 상태 코드에 따라 서비스는 "비정상" 응답을 수신하면 조치를 취합니다. "정상" 또는 "비정상"의 정의는 제공하는 서비스 유형에 따라 다르므로 Terminus 통합은 일련의 상태 지표를 통해 지원합니다.

예를 들어, 웹 서버가 데이터를 저장하기 위해 MongoDB를 사용하는 경우, MongoDB가 여전히 가동 중인지 여부는 중요한 정보가 될 것입니다. 이 경우, [몽구스헬스 인디케이터](#)를 사용할 수 있습니다. 올바르게 구성한 경우(나중에 자세히 설명), 상태 확인 주소는 MongoDB가 실행 중인지 여부에 따라 정상 또는 비정상 HTTP 상태 코드를 반환합니다.

시작하기

nestjs/terminus를 시작하려면 필요한 종속성을 설치해야 합니다.

```
npm install --save @nestjs/terminus
```

상태 확인 설정

상태 확인은 상태 지표의 요약을 나타냅니다. 상태 점검은 서비스가 정상 상태인지 비정상 상태인지에 대한 점검을 실행합니다. 할당된 모든 상태 표시기가 실행 중이면 상태 확인이 긍정적입니다. 많은 애플리케이션에 유사한 상태 표시기가 필요하기 때문에 [@nestjs/terminus](#)에서는 다음과 같이 미리 정의된 표시기 집합을 제공합니다:

- [HttpHealthIndicator](#)
- [TypeOrmHealthIndicator](#)

[몽구스헬스인디케이터](#)

- [시퀄라이즈헬스인디케이터](#)
- [마이](#)

[크로오름헬스인디케이터](#)

- [프리즈마](#)

헬스인디케이터

- 마이크로서비스건강지표•

GRPCHealthIndicator

- MemoryHealthIndicator
- DiskHealthIndicator

첫 번째 상태 확인을 시작하기 위해 `HealthModule`을 생성하고 `TerminusModule`을 임포트해 보겠습니다.

를 가져오기 배열에 넣습니다.

정보 힌트 [Nest CLI](#)를 사용하여 모듈을 생성하려면 `$ nest g 모듈 상태`를 실행하기만 하면 됩니다.

명령을 사용합니다.

```
@@파일명(health.module)
```

```
'@nestjs/common'에서 { Module }을 가져옵니다;
```

```
'@nestjs/terminus'에서 { TerminusModule }을 가져옵니다;
```

```
모듈({
```

```
  수입: [종단 모듈]
```

```
)
```

```
내보내기 클래스 HealthModule {}
```

Nest CLI를 사용하여 쉽게 설정할 수 있는 [컨트롤러](#)를 사용하여 상태 확인을 실행할 수 있습니다.

```
nest g 컨트롤러 상태
```

정보 애플리케이션에서 종료 후크를 활성화하는 것이 좋습니다. 종료 후크를 활성화하면 종료 통합이

이 수명 주기 이벤트를 사용합니다. 종료 후크에 대한 자세한 내용은 [여기](#)를 참조하세요.

HTTP 상태 확인

`nestjs/terminus`를 설치하고 `TerminusModule`을 임포트하고 새 컨트롤러를 생성했으면 상태 검사를 만들 준비가 된 것입니다.

`HTTPHealthIndicator`를 사용하려면 `@nestjs/axios` 패키지가 필요하므로 반드시 설치해야 합니다:

```
npm i --save @nestjs/axios axios
```

이제 `헬스` 컨트롤러를 설정할 수 있습니다:

@@파일명(health.controller)

'@nestjs/common'에서 { Controller, Get }을 가져옵니다;
'@nestjs/terminus'에서 { HealthCheckService, HttpHealthIndicator, HealthCheck }를 임포트합니다;

컨트롤러('health')

```
내보내기 클래스 HealthController { 생성자(  
    개인 건강: HealthCheckService, 비공개  
    http: HttpHealthIndicator,  
) {}  
  
Get()  
@HealthCheck()  
check() {  
    return this.health.check([  
        () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),  
    ]);  
}
```

```

}
@@switch
'@nestjs/common'에서 { Controller, Dependencies, Get }를 임포트하고,
'@nestjs/terminus'에서 { HealthCheckService, HttpHealthIndicator,
HealthCheck }를 임포트합니다;

Controller('health') @Dependencies(HealthCheckService,
HttpHealthIndicator) export class HealthController {
  생성자( 개인 건강,
  개인 http,
) { }

Get()
@HealthCheck()
healthCheck() {
  return this.health.check([
    () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),
  ])
}
}
}

```

@@파일명(health.module)

```

'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/terminus'에서 { TerminusModule } 가져오기;
'@nestjs/axios'에서 { HttpModule } 가져오기;
'./health.controller'에서 { HealthController }를 가져옵니다;

```

모듈({

```

  임포트: [TerminusModule, HttpModule], 컨
  트롤러: [HealthController],
})
내보내기 클래스 HealthModule {}

@@switch
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/terminus'에서 { TerminusModule } 가져오기;
'@nestjs/axios'에서 { HttpModule } 가져오기;
'./health.controller'에서 { HealthController }를 가져옵니다;

```

모듈({

```

  임포트: [TerminusModule, HttpModule], 컨
  트롤러: [HealthController],
})
내보내기 클래스 HealthModule {}
```

이제 상태 확인이 <https://docs.nestjs.com> 주소로 GET 요청을 보냅니다. 해당 주소에서 정상 응답을 받으면 <http://localhost:3000/health> 경로에서 200 상태 코드가 포함된 다음 객체를 반환합니다.



```
{
  "상태": "ok", "info": {
    "nestjs-docs": {
      "상태": "up"
    }
  },
  "error": {},
  "details": {
    "nestjs-docs": {
      "상태": "up"
    }
  }
}
```

이 응답 객체의 인터페이스는 [@nestjs/terminus](#) 패키지에서 다음과 같이 액세스할 수 있습니다.

HealthCheckResult 인터페이스.

상태	상태 표시기가 실패하면 상태는 ' 오류 '가 됩니다. NestJS 앱이 종료 중이지만 여전히 HTTP 요청을 수락하는 경우 상태 확인의 상태는 ' 종료 중 '입니다.	<code>'error' \ \ 'ok'</code> <code>\ </code> <code>'shutting_down'</code>
정보	각 상태 표시기의 정보를 포함하는 객체는 다음과 같습니다.	<code>객체</code>
오류	상태 ' 업 ', 즉 "건강"을 반환합니다.	<code>object</code>
세부 정보 각 상태 표시기의 모든 정보를 포함하는 객체		<code>객체</code>

특정 HTTP 응답 코드 확인

특정 경우에는 특정 기준을 확인하고 응답의 유효성을 검사해야 할 수도 있습니다. 예를 들어 <https://my-external-service.com> 가 응답 코드 [204](#)를 반환한다고 가정해 보겠습니다.

[HttpHealthIndicator](#).[responseCheck](#)를 사용하면 해당 응답 코드를 구체적으로 확인하고 다른 모든 코드를 비정상적인 것으로 판단할 수 있습니다.

[204](#) 이외의 다른 응답 코드가 반환되는 경우 다음 예제는 비정상적인 응답입니다. 세 번째 매개 변수는 응답이 정상([true](#)) 또는 비정상([false](#))으로 간주되는지 여부를 부울로 반환하는 함수(동기화 또는 비동기화)를 제공해

야 합니다.

```
@@파일명(health.controller)
// `HealthController` 클래스 내에서

Get()
@HealthCheck()
check() {
    return this.health.check([
        () =>])
}
```

```

    this.http.responseCheck(
      '내-외부-서비스',
      'https://my-external-service.com',
      (res) => res.status === 204,
    ),
  ]);
}

```

TypeOrm 상태 표시기

Terminus는 상태 확인에 데이터베이스 검사를 추가하는 기능을 제공합니다. 이 상태 표시기를 시작하려면 [데이터베이스 챕터를](#) 확인하여 애플리케이션 내에서 데이터베이스 연결이 설정되어 있는지 확인해야 합니다.

정보 힌트 뒤에서 `TypeOrmHealthIndicator`는 데이터베이스가 아직 살아 있는지 확인하는 데 자주 사용되는 `SELECT 1-SQL` 명령을 실행하기만 하면 됩니다. Oracle 데이터베이스를 사용하는 경우

```

@@파일명(health.controller)
@Controller('health')
내보내기 클래스 HealthController { 생성자(
  개인 건강: HealthCheckService, 비공개
  db: TypeOrmHealthIndicator,
) {}

Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.db.pingCheck('database'),
  ]);
}
}

@@스위치
@Controller('health')
@Dependencies(HealthCheckService, TypeOrmHealthIndicator)
export class HealthController {
  생성자( 개인 건강,
  개인 DB,
) { }

Get()
@HealthCheck()
healthCheck() {
  return this.health.check([
    () => this.db.pingCheck('database'),
  ])
}
}

```


데이터베이스에 연결할 수 있는 경우 이제 요청 시 다음과 같은 JSON 결과를 볼 수 있습니다.

<http://localhost:3000> 에 GET 요청을 보냅니다:

```
{  
  "상태": "ok", "info": {  
    "데이터베이스": {  
      "상태": "up"  
    },  
    "error": {},  
    "details": {  
      "데이터베이스": {  
        "상태": "up"  
      }  
    }  
  }  
}
```

앱에서 여러 데이터베이스를 사용하는 경우 각 연결을 [HealthController](#)에 삽입해야 합니다. 그런 다음 연결 참조를 [TypeOrmHealthIndicator](#)에 전달하기만 하면 됩니다.

```
@@파일명(health.controller)  
@Controller('health')  
내보내기 클래스 HealthController { 생성자(  
  개인 건강: HealthCheckService, 비공개  
  db: TypeOrmHealthIndicator,  
  @InjectConnection('albumsConnection')  
  비공개 albumsConnection: Connection,  
  @InjectConnection()  
  private defaultConnection: 연결,  
) {}  
  
Get()  
@HealthCheck()  
check() {  
  return this.health.check([  
    () => this.db.pingCheck('albums-database', { connection:  
      this.albumsConnection }),  
    () => this.db.pingCheck('database', { connection:  
      this.defaultConnection }),  
  ]);  
}
```

디스크 상태 표시기

DiskHealthIndicator를 사용하면 사용 중인 스토리지의 양을 확인할 수 있습니다. 시작하려면

DiskHealthIndicator를 HealthController에 삽입하세요. 다음 예제에서는

경로의 스토리지 사용량(또는 Windows의 경우 C:\\ 사용 가능)을 확인합니다. 전체의 50%를 초과하는 경우 저장 공간이 부족하면 건강하지 않은 상태 확인으로 응답합니다.

```
@@파일명(health.controller)
@Controller('health')
내보내기 클래스 HealthController { 생성자(
    개인 읽기 전용 상태: HealthCheckService, 비공개 읽기 전
    용 디스크: DiskHealthIndicator,
) {}

Get()
@HealthCheck()
check() {
    return this.health.check([
        () => this.disk.checkStorage('storage', { 경로: '/', 임계값 퍼센트:
0.5 }),
    ]);
}
}

@@스위치
@Controller('health')
@Dependencies(HealthCheckService, DiskHealthIndicator)
export class HealthController {
    constructor(health, disk) {}

    Get()
    @HealthCheck()
    healthCheck() {
        return this.health.check([
            () => this.disk.checkStorage('storage', { 경로: '/', 임계값 퍼센트:
0.5 }),
        ])
    }
}
```

DiskHealthIndicator.checkStorage 함수를 사용하면 고정된 공간도 확인할 수 있습니다. 다음 예는

/my-app/ 경로가 250GB를 초과하는 경우 건강하지 않은 것으로 간주합니다.

```
@@파일명(health.controller)
// `HealthController` 클래스 내에서

Get()
@HealthCheck()
check() {
    return this.health.check([
        () => this.disk.checkStorage('storage', {path      : '/',
                                                 threshold: 250
    })
}
```

```
    1024 * 1024 * 1024, })
]);
}
```

메모리 상태 표시기

프로세스가 특정 메모리 제한을 초과하지 않는지 확인하려면 [메모리 상태](#) 표시기를 사용할 수 있습니다. 다음 예제는 프로세스의 힙을 확인하는 데 사용할 수 있습니다.

정보 힌트 힙은 동적으로 할당된 메모리가 상주하는 메모리 부분(즉, malloc을 통해 할당된 메모리)입니다.

힙에서 할당된 메모리는 다음 중 하나가 발생할 때까지 할당된 상태로 유지됩니다:

- 메모리가 *비어* 있습니다.

프로그램이 종료됩니다.

```
@@파일명(health.controller)
@Controller('health')
내보내기 클래스 HealthController { 생성자(
    개인 건강: HealthCheckService, 개인 메모리:
    MemoryHealthIndicator,
) {}

Get()
@HealthCheck()
check() {
    return this.health.check([
        () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
    ]);
}
}

@@스위치
@Controller('health')
@Dependencies(HealthCheckService, MemoryHealthIndicator)
export class HealthController {
    constructor(health, memory) {}

    Get()
    @HealthCheck()
    healthCheck() {
        return this.health.check([
            () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
        ])
    }
}
```

`MemoryHealthIndicator.checkRSS`를 사용하여 프로세스의 메모리 RSS를 확인할 수도 있습니다. 이 예제는 프로세스가 150MB를 초과하는 경우 비정상 응답 코드를 반환합니다.

할당되었습니다.

정보 힌트 RSS는 상주 세트 크기이며 해당 프로세스에 할당된 메모리와 RAM에 있는 메모리의 양을 표시하는 데 사용됩니다. 스왑아웃된 메모리는 포함되지 않습니다. 공유 라이브러리의 페이지가 실제로 메모리에 있는 한 공유 라이브러리의 메모리는 포함됩니다. 모든 스택 및 힙 메모리는 포함됩니다.

@@파일명(health.controller)

```
// `HealthController` 클래스 내에서

Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.memory.checkRSS('memory_rss', 150 * 1024 * 1024),
  ]);
}
```

사용자 지정 상태 표시기

경우에 따라 [@nestjs/terminus](#)에서 제공하는 사전 정의된 상태 표시기가 모든 상태 확인 요구 사항을 충족하지 못할 수 있습니다. 이 경우 필요에 따라 사용자 정의 상태 지표를 설정할 수 있습니다.

사용자 지정 지표를 나타낼 서비스를 만드는 것으로 시작해 보겠습니다. 지표가 어떻게 구조화되는지에 대한 기본적인 이해를 돋기 위해 [DogHealthIndicator](#) 예제를 만들어 보겠습니다. 이 서비스는 모든 [Dog](#) 객체의 유형이 '[goodboy](#)'인 경우 '[up](#)' 상태를 가져야 합니다. 이 조건이 충족되지 않으면 오류가 발생해야 합니다.

@@파일명(dog.health)

'@nestjs/common'에서 { Injectable }을 가져옵니다;
'@nestjs/terminus'에서 { HealthIndicator, HealthIndicatorResult, HealthCheckError }를 가져옵니다;

내보내기 인터페이스 Dog {

 이름: 문자열;
 유형: 문자열;
}

@Injectable()

```
export class DogHealthIndicator extends HealthIndicator {
  private dogs: Dog[] = [
    { 이름: '피도', 유형: '굿보이' },
    { 이름: '렉스', 유형: '나쁜남자' },
  ];

  async isHealthy(키: 문자열): Promise<HealthIndicatorResult> { const
    badboys = this.dogs.filter(dog => dog.type === 'badboy'); const
    isHealthy = badboys.length === 0;
    const result = this.getStatus(key, isHealthy, { badboys:
```

```
badboys.length });

  if (isHealthy) {
    결과를 반환합니다
  ;
}

 새로운 HealthCheckError('도그체크 실패', 결과)를 던집니다;
}

@@switch
'@nestjs/common'에서 { Injectable }을 가져옵니다;
'@godaddy/terminus'에서 { HealthCheckError }를 가져옵니다;

@Injectable()
export class DogHealthIndicator extends HealthIndicator {
  dogs = [
    { 이름: '피도', 유형: '굿보이' },
    { 이름: '렉스', 유형: '나쁜남자' },
  ];

  async isHealthy(key) {
    const badboys = this.dogs.filter(dog => dog.type === 'badboy');
    const isHealthy = badboys.length === 0;
    const result = this.getStatus(key, isHealthy, { badboys:
      badboys.length });

    if (isHealthy) {
      결과를 반환합니다
    ;
}

  새로운 HealthCheckError('도그체크 실패', 결과)를 던집니다;
}
}
```

다음으로 해야 할 일은 상태 지표를 공급자로 등록하는 것입니다.

@@파일명 (health.module)

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/terminus'에서 { TerminusModule }을 임포트하고,  
'./dog.health'에서 { DogHealthIndicator }를 임포트합니다;
```

모듈 ({

컨트롤러: [HealthController], 임포트

: [TerminusModule], 제공자:

[DogHealthIndicator]

정보 힌트 실제 애플리케이션에서 DogHealthIndicator는 별도의 모듈(예: DogModule)에 제공되어야 하
내보내기 클래스 HealthModule {}
며, 이 모듈은 HealthModule에서 가져올 것입니다.

마지막 필수 단계는 필수 상태 확인 엔드포인트에 현재 사용 가능한 상태 표시기를 추가하는 것입니다. 이를 위해

HealthController로 돌아가서 검사 함수에 추가합니다.

```
@@파일명(health.controller)
'@nestjs/terminus'에서 { HealthCheckService, HealthCheck }를 임포트하고
, '@nestjs/common'에서 { Injectable, Dependencies, Get }을 임포트하고,
'./dog.health'에서 { DogHealthIndicator }를 임포트합니다;

@Injectable()
내보내기 클래스 HealthController { 생성자(
    개인 건강: 건강검진 서비스,
    비공개 dogHealthIndicator: DogHealthIndicator
) {}

Get()
@HealthCheck()
healthCheck() {
    return this.health.check([
        () => this.dogHealthIndicator.isHealthy('dog'),
    ])
}
}

@@switch
'@nestjs/terminus'에서 { HealthCheckService, HealthCheck }를 임포트하고
, '@nestjs/common'에서 { Injectable, Get }을 임포트합니다;
'./dog.health'에서 { DogHealthIndicator }를 가져옵니다;

@Injectable()
@Dependencies(HealthCheckService, DogHealthIndicator)
export class HealthController {
    생성자( 개인 건강,
    개인 개건강지표
) {}

Get()
@HealthCheck()
healthCheck() {
    return this.health.check([
        () => this.dogHealthIndicator.isHealthy('dog'),
    ])
}
}
```

로깅

종결은 예를 들어 상태 검사에 실패한 경우와 같은 오류 메시지만 기록합니다. `TerminusModule.forRoot()` 메서드를 사용하면 오류를 기록하는 방법을 더 잘 제어할 수 있을 뿐만 아니라 로깅 자체를 완전히 인수할 수 있습니다.

이 섹션에서는 사용자 정의 로거 `TerminusLogger`를 만드는 방법을 안내해 드리겠습니다. 이 로거는 기본 제공 로거를 확장합니다. 따라서 로거의 어느 부분을 덮어쓸지 선택할 수 있습니다.

정보 NestJS의 사용자 정의 로거에 대해 자세히 알아보려면 [여기에서 자세히 읽어보세요.](#)

@@파일명(`terminus-logger.service`)

```
'@nestjs/common'에서 { Injectable, Scope, ConsoleLogger }를 임포트합니다;

주사형({ 범위: Scope.TRANSIENT })
export class TerminusLogger extends ConsoleLogger {
  error(message: any, stack?: string, context?: string): void;
  error(message: any, ...optionalParams: any[]): void;
  error(
    메시지: 알 수 없음,
    스택?: 알 수 없음, 컨
    텍스트?: 알 수 없음,
    ...나머지: 알 수 없음[]
  ): void {
    // 오류 메시지를 기록하는 방법을 여기에 덮어씁니다.
  }
}
```

사용자 정의 로거를 생성한 후에는 해당 로거를

`TerminusModule.forRoot()`를 호출합니다.

@@파일명(`health.module`)

```
@Module({
  imports: [
    TerminusModule.forRoot({
      로거: TerminusLogger,
    }),
  ],
})
내보내기 클래스 HealthModule {}
```

오류 메시지를 포함하여 `Terminus`에서 오는 모든 로그 메시지를 완전히 차단하려면 `Terminus`를 이렇게 구성하세요.

```
@@파일명(health.module)
```

```
@Module({  
    imports: [  
        TerminusModule.forRoot({  
            logger: false,  
        }),  
    ],  
})
```

```
내보내기 클래스 HealthModule {}
```

터미널을 사용하면 로그에 상태 확인 오류를 표시하는 방법을 구성할 수 있습니다.

오류	설명	예제
로그		
스타일		
일		

json 오류 발생 시 상태 검사 결과 요약을 JSON 객체로 출력합니다.
(기본값)

예쁜 형식이 지정된 상자 내에 오류가 있는 경우 상태 검사 결과의 요약을 인쇄하고 성공/오류 결과를 강조 표시합니다.


다음 코드조각에서와 같이 `errorLogStyle` 구성 옵션을 사용하여 로그 스타일을 변경할 수 있습니다.

```
@@파일명(health.module)
@Module({
    임포트합니다: [
        TerminusModule.forRoot({
            errorLogStyle: 'pretty',
        }),
    ]
})
내보내기 클래스 HealthModule {}
```

더 많은 예제

더 많은 작업 예제는 [여기에서](#) 확인할 수 있습니다.

CQRS

간단한 **CRUD**(생성, 읽기, 업데이트 및 삭제) 애플리케이션의 흐름은 다음과 같이 설명할 수 있습니다:

- . 컨트롤러 계층은 HTTP 요청을 처리하고 서비스 계층에 작업을 위임합니다.
- . 서비스 계층은 대부분의 비즈니스 로직이 있는 곳입니다.
- . 서비스는 리포지토리/DAO를 사용하여 엔티티를 변경/유지합니다.
- . 엔티티는 값의 컨테이너 역할을 하며, 세터와 게터가 있습니다.

이 패턴은 일반적으로 중소규모 애플리케이션에는 충분하지만, 규모가 크고 복잡한 애플리케이션에는 최선의 선택이 아닐 수 있습니다. 이러한 경우에는 애플리케이션의 요구 사항에 따라 CQRS(명령 및 쿼리 책임 분리) 모델이 더 적합하고 확장성이 높을 수 있습니다. 이 모델의 장점은 다음과 같습니다:

- 관심사 분리. 이 모델은 읽기 작업과 쓰기 작업을 별도의 모델로 분리합니다. • 확장성. 읽기 및 쓰기 작업을 독립적으로 확장할 수 있습니다.
- 유연성. 이 모델을 사용하면 읽기 및 쓰기 작업에 서로 다른 데이터 저장소를 사용할 수 있습니다.
- 성능. 이 모델을 사용하면 읽기 및 쓰기 작업에 최적화된 다양한 데이터 저장소를 사용할 수 있습니다.

이 모델을 용이하게 하기 위해 Nest는 경량 **CQRS 모듈**을 제공합니다. 이 장에서는 이 모듈을 사용하는 방법

을 설명합니다. 설치

먼저 필요한 패키지를 설치합니다:

```
npm install --save @nestjs/cqrs
```

명령

명령은 애플리케이션 상태를 변경하는 데 사용됩니다. 데이터 중심이 아닌 작업 기반이어야 합니다. 명령이 전송되면 해당 명령 핸들러가 처리합니다. 핸들러는 애플리케이션 상태를 업데이트할 책임이 있습니다.

```
@@파일명(heroes-game.service) @Injectable()
export class HeroesGameService {
    constructor(private commandBus: CommandBus) {}

    async killDragon(heroId: 문자열, killDragonDto: KillDragonDto) {
        return this.commandBus.execute(
            새로운 KillDragonCommand(heroId, killDragonDto.dragonId)
        );
    }
}

@@스위치
@Injectable()
```

```

@Dependencies(CommandBus) export
class HeroesGameService {
  constructor(commandBus) {
    this.commandBus = commandBus;
  }

  async killDragon(heroId, killDragonDto) {
    return this.commandBus.execute(
      새로운 KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}

```

위의 코드 스니펫에서는 `KillDragonCommand` 클래스를 인스턴스화하여 `CommandBus`의 `실행()` 메서드에 전달합니다. 이것이 데모 명령 클래스입니다:

```

@@filename(kill-dragon.command)
export class KillDragonCommand {
  생성자(
    공개 읽기 전용 heroId: 문자열, 공개 읽기 전용 dragonId: 문자열,
  ) {}
}

@@switch
export class KillDragonCommand {
  constructor(heroId, dragonId) {
    this.heroId = heroId;
    this.dragonId = dragonId;
  }
}

```

`CommandBus`은 명령 스트림을 나타냅니다. 적절한 핸들러에 명령을 디스패치하는 역할을 담당합니다. `실행()` 메서드는 핸들러가 반환한 값으로 확인되는 프로미스를 반환합니다.

`KillDragonCommand` 명령에 대한 핸들러를 만들어 보겠습니다.

@@파일명(kill-dragon.handler)

@CommandHandler(KillDragonCommand) 내보내기

클래스 KillDragonHandler는

ICommandHandler<KillDragonCommand>를 구현

합니다 {

생성자(비공개 저장소: HeroRepository) {}

```
async execute(command: KillDragonCommand) {
    const { heroId, dragonId } = command;
    const hero = this.repository.findOneById(+heroId);

    hero.killEnemy(dragonId);
    await this.repository.persist(hero);
```

```

    }
}

@@스위치

@CommandHandler(KillDragonCommand)
@Dependencies(HeroRepository) 내보내

기 클래스 KillDragonHandler {
    constructor(repository) {
        this.repository = repository;
    }

    async execute(command) {
        const { heroId, dragonId } = command;
        const hero = this.repository.findOneById(+heroId);

        hero.killEnemy(dragonId);
        await this.repository.persist(hero);
    }
}

```

이 핸들러는 저장소에서 영웅 엔티티를 검색하고 `killEnemy()` 메서드를 호출한 다음 변경 사항을 유지합니다. `KillDragonHandler` 클래스는 `실행()` 메서드를 구현해야 하는 `ICommandHandler` 인터페이스를 구현합니다. `실행()` 메서드는 명령 객체를 인수로 받습니다.

쿼리

쿼리는 애플리케이션 상태에서 데이터를 검색하는 데 사용됩니다. 쿼리는 작업 기반이 아닌 데이터 중심이어야 합니다. 쿼리가 전송되면 해당 쿼리 핸들러가 처리합니다. 핸들러는 데이터를 검색할 책임이 있습니다.

쿼리버스는 명령버스와 동일한 패턴을 따릅니다. 쿼리 핸들러는 `IQueryHandler` 인터페이스에 `@QueryHandler()` 데코레이터를 사용하여 주석을 달 수 있습니다. 이벤트

이벤트는 애플리케이션 상태의 변경 사항을 애플리케이션의 다른 부분에 알리는 데 사용됩니다. 이벤트는 다음과 같습니다.

모델에 의해 디스패치되거나 `이벤트버스`를 사용하여 직접 디스패치됩니다. 이벤트가 디스패치되면 해당 이벤트 핸들러가 처리합니다. 그러면 핸들러는 예를 들어 읽기 모델을 업데이트할 수 있습니다.

데모를 위해 이벤트 클래스를 만들어 보겠습니다:

```
@@파일명(hero-killed-dragon.event) 내
```

```
보내기 클래스 HeroKilledDragonEvent {  
    생성자(  
        공개 읽기 전용 heroId: 문자열, 공개 읽  
        기 전용 dragonId: 문자열,  
    ) {}  
}
```

```
@@switch
```

```
내보내기 클래스 HeroKilledDragonEvent {
```

```

constructor(heroId, dragonId) {
    this.heroId = heroId;
    this.dragonId = dragonId;
}
}

```

이제 이벤트는 `EventBus.publish()` 메서드를 사용하여 직접 디스패치할 수도 있지만, 모델에서 디스패치할 수도 있습니다. `killEnemy()` 메서드가 호출될 때 `HeroKilledDragonEvent` 이벤트를 디스패치하도록 `Hero` 모델을 업데이트해 보겠습니다.

```

@@파일명(hero.model)
export class Hero extends AggregateRoot {
    constructor(private id: string) {
        super();
    }

    killEnemy(enemyId: 문자열) {
        // 비즈니스 로직
        this.apply(new HeroKilledDragonEvent(this.id, enemyId));
    }
}

@@switch
export class Hero extends AggregateRoot {
    constructor(id) {
        super();
        this.id = id;
    }

    killEnemy(enemyId) {
        // 비즈니스 로직
        this.apply(new HeroKilledDragonEvent(this.id, enemyId));
    }
}

```

`apply()` 메서드는 이벤트를 디스패치하는 데 사용됩니다. 이 메서드는 이벤트 객체를 인자로 받습니다. 하지만 우리 모델은 이벤트버스를 인식하지 못하기 때문에 모델과 연결해야 합니다. 이 작업은 `EventPublisher` 클래스를 사용하여 수행할 수 있습니다.

```
@@파일명(kill-dragon.handler)
@CommandHandler(KillDragonCommand) 내보내기
클래스 KillDragonHandler는
ICommandHandler<KillDragonCommand>를 구현
합니다 {
    생성자(
        비공개 저장소: 비공개 퍼블리셔인
        HeroRepository: 이벤트 퍼블리셔,
    ) {}

async execute(command: KillDragonCommand) {
    const { heroId, dragonId } = command;
```

```

    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}

@@스위치 @CommandHandler(킬드래곤 커맨드)
@Dependencies(HeroRepository, EventPublisher)
export class KillDragonHandler {
  constructor(repository, publisher) {
    this.repository = repository;
    this.publisher = publisher;
  }

  async execute(command) {
    const { heroId, dragonId } = command;
    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}

```

이벤트 퍼블리셔#병합 오브젝트 컨텍스트 메서드는 이벤트 퍼블리셔를 제공된 오브젝트에 병합하므로 이제 오브젝트가 이벤트 스트림에 이벤트를 게시할 수 있게 됩니다.

이 예제에서는 모델에서 `commit()` 메서드도 호출합니다. 이 메서드는 미결 이벤트를 디스패치하는 데 사용됩니다. 이벤트를 자동으로 디스패치하려면 `autoCommit` 속성을 `true`로 설정하면 됩니다:

```

export class Hero extends AggregateRoot {
  constructor(private id: string) {
    super();
    this.autoCommit = true;
  }
}

```

이벤트 퍼블리셔를 존재하지 않는 오브젝트가 아닌 클래스에 병합하려는 경우 **이벤트 퍼블리셔 #mergeClassContext** 메서드를 사용할 수 있습니다:

```

const HeroModel = this.publisher.mergeClassContext(Hero);
const hero = new HeroModel('id'); // <-- HeroModel은 클래스입니다

```

이제 `HeroModel` 클래스의 모든 인스턴스는 다음을 사용하지 않고도 이벤트를 게시할 수 있습니다.

메서드를 호출합니다.

또한 [이벤트버스](#)를 사용하여 수동으로 이벤트를 발생시킬 수도 있습니다:

```
this.eventBus.publish(new HeroKilledDragonEvent());
```

정보 힌트 [이벤트버스](#)는 인젝터블 클래스입니다.

각 이벤트에는 여러 이벤트 핸들러가 있을 수 있습니다.

[@@파일명](#)(영웅을 죽인 용.핸들러) [@이벤트 핸들러](#)(영

웅을 죽인 용 이벤트)

```
export 클래스 HeroKilledDragonHandler 구현 IEventHandler<HeroKilledDragonEvent>
{
    생성자(비공개 저장소: HeroRepository) {}

    handle(event: HeroKilledDragonEvent) {
        // 비즈니스 로직
    }
}
```

정보 힌트 이벤트 핸들러를 사용하기 시작하면 기존 HTTP 웹 컨텍스트에서 벗어나게 된다는 점에 유의하세요.

-

- 명령 핸들러의 오류는 기본 제공 [예외 필터](#)로 여전히 포착할 수 있습니다.

이벤트 핸들러의 오류는 예외 필터로 포착할 수 없으므로 수동으로 처리해야 합니다. 간단한 [시도/잡기](#),

- 보상 이벤트를 트리거하여 Sagas를 사용하거나 다른 해결 방법을 선택해야 합니다.

[CommandHandlers](#)의 HTTP 응답은 여전히 클라이언트로 다시 전송할 수 있습니다.

이벤트 핸들러의 HTTP 응답은 불가능합니다. 클라이언트에 정보를 보내려면 [WebSocket](#), SSE 또는

다른 솔루션을 사용할 수 있습니다.

사가

사가는 이벤트를 수신하고 새로운 명령을 트리거할 수 있는 장기 실행 프로세스입니다. 일반적으로 애플리케이션에서 복잡한 워크플로를 관리하는 데 사용됩니다. 예를 들어, 사용자가 가입할 때 사가는 [UserRegisteredEvent](#)를 수신하고 사용자에게 환영 이메일을 보낼 수 있습니다.

사가는 매우 강력한 기능입니다. 하나의 사가는 1...* 이벤트를 수신할 수 있습니다. [RxJS](#) 라이브러리를 사용하면 이벤트 스트림을 필터링, 매팅, 포크, 병합하여 정교한 워크플로를 만들 수 있습니다. 각 사가는 명령 인스턴스를 생성하

는 Observable을 반환합니다. 그런 다음 이 명령은 CommandBus 의해 비동기적으로 전송됩니다.

HeroKilledDragonEvent를 듣고 영웅을 처치하는 사가를 만들어 보겠습니다.

DropAncientItemCommand 명령입니다.

```
@@파일명(heroes-game.saga)
```

```
@Injectable()
```

```

내보내기 클래스 히어로즈게임사가 { @Saga()
  dragonKilled = (events$: Observable<any>): Observable< ICommand> => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}

@@스위치
@Injectable()
내보내기 클래스 히어로즈게임사가 { @Saga()
  dragonKilled = (events$) => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}

```

정보 힌트 ofType 연산자와 @Saga() 데코레이터는 @nestjs/cqrs에서 내보냅니다.

패키지입니다.

사가() 데코레이터는 메서드를 사가로 표시합니다. events\$ 인수는 모든 이벤트의 관찰 가능한 스트림입니다.

ofType 연산자는 지정된 이벤트 유형에 따라 스트림을 필터링합니다. map 연산자는 이벤트를 새 명령 인스턴스에 매핑합니다.

이 예제에서는 HeroKilledDragonEvent를 DropAncientItemCommand 명령에 매핑합니다. 그러면 CommandBus에 의해 DropAncientItemCommand 명령이 자동 발송됩니다.

설정

마무리하자면, 모든 명령 핸들러, 이벤트 핸들러, 사가를

히어로즈게임모듈:

```
@@파일명(heroes-game.module)
```

```
export const CommandHandlers = [KillDragonHandler,  
DropAncientItemHandler];  
export const EventHandlers = [HeroKilledDragonHandler,  
HeroFoundItemHandler];
```

```
모듈({
```

```
    수입: [CqrsModule],
```

```
    컨트롤러: [히어로즈게임컨트롤러], 공급자: [
```

```
        히어로즈게임서비스, 히어로즈게임
```

```
        사가,
```

```
        ...명령 핸들러,
```

```

    ...이벤트 핸들러, 영웅 저장소,
]
})

내보내기 클래스 히어로즈게임모듈 {}

```

처리되지 않은 예외

이벤트 핸들러는 비동기 방식으로 실행됩니다. 즉, 애플리케이션이 일관되지 않은 상태가 되는 것을 방지하기 위해 항상 모든 예외를 처리해야 합니다. 그러나 예외가 처리되지 않으면 이벤트버스는 `UnhandledExceptionInfo` 객체를 생성하고 이를 `UnhandledExceptionBus` 스트림으로 푸시합니다. 이 스트림은 처리되지 않은 예외를 처리하는 데 사용할 수 있는 `Observable`입니다.

```

private destroy$ = new Subject<void>();

constructor(private unhandledExceptionsBus: UnhandledExceptionBus) {
    this.unhandledExceptionsBus
        .pipe(takeUntil(this.destroy$))
        .subscribe((exceptionInfo) => {
            // 여기서 예외 처리
            // 예: 외부 서비스로 전송, 프로세스 종료 또는 새 이벤트 게시
        });
}

onModuleDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
}

```

예외를 필터링하려면 다음과 같이 `ofType` 연산자를 사용할 수 있습니다:

```

this.unhandledExceptionsBus.pipe(takeUntil(this.destroy$),
    UnhandledExceptionBus.ofType(TransactionNotAllowedException)).subscribe((e
xceptionInfo) => {
    // 여기서 예외 처리
});

```

여기서 `TransactionNotAllowedException`은 필터링하려는 예외입니다.

`UnhandledExceptionInfo` 객체에는 다음과 같은 프로퍼티가 포함되어 있습니다:

```
내보내기 인터페이스 UnhandledExceptionInfo<Cause = IEvent | ICommand,  
Exception = any> {  
    /**
```

```

 * 던져진 예외입니다.
 */
예외: 예외;
/***
 * 예외의 원인(이벤트 또는 명령 참조).
 */
원인: 원인;
}

```

모든 이벤트 구독하기

`CommandBus`, `QueryBus`, `EventBus`는 모두 `Observable`입니다. 즉, 전체 스트림을 구독하고 예를 들어 모든 이벤트를 처리할 수 있습니다. 예를 들어 모든 이벤트를 콘솔에 기록하거나 이벤트 저장소에 저장할 수 있습니다.

```

private destroy$ = new Subject<void>();

constructor(private eventBus: EventBus) {
    this.eventBus
        .pipe(takeUntil(this.destroy$))
        .subscribe((event) => {
            // 데이터베이스에 이벤트 저장
        });
}

onModuleDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
}

```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

문서

Compodoc은 Angular 애플리케이션을 위한 문서화 도구입니다. Nest와 Angular는 유사한 프로젝트 및 코드 구조를 공유하므로 Compodoc은 Nest 애플리케이션에서도 작동합니다.

설정

기존 Nest 프로젝트 내에서 Compodoc을 설정하는 방법은 매우 간단합니다. 먼저 OS 터미널에서 다음 명령으로 개발 종속성을 추가합니다:

```
$ npm i -D @compodoc/compodoc
```

세대

다음 명령을 사용하여 프로젝트 문서를 생성합니다([npx를 지원하려면 npm 6이 필요합니다](#)). 더 많은 옵션은 [공식 문서를 참조하세요](#).

```
$ npx @compodoc/compodoc -p tsconfig.json -s
```

브라우저를 열고 <http://localhost:8080> 으로 이동합니다. 초기 Nest CLI 프로젝트가 표시됩니다:



기여하기

[여기에서](#) Compodoc 프로젝트에 참여하고 기여할 수 있습니다.

프리즈마

프리즈마는 Node.js 및 TypeScript용 오픈소스 ORM입니다. 일반 SQL을 작성하거나 SQL 쿼리 빌더(예: [knex.js](#)) 또는 ORM(예: [TypeORM](#) 및 [Sequelize](#))과 같은 다른 데이터베이스 액세스 도구를 사용하는 대신에 사용됩니다. 프리즈마는 현재 PostgreSQL, MySQL, SQL Server, SQLite, MongoDB, CockroachDB([미리보기](#))를 지원합니다.

Prisma는 일반 JavaScript와 함께 사용할 수 있지만 TypeScript를 수용하며 TypeScript 에코시스템의 다른 ORM이 보장하는 수준을 뛰어넘는 유형 안전성을 제공합니다. Prisma와 TypeORM의 유형 안전 보장에 대한 심층적인 비교는 [여기에서](#) 확인할 수 있습니다.

정보 참고 Prisma의 작동 방식에 대한 간략한 개요를 확인하려면 [빠른 시작을](#) 따르거나 [문서에서 소개를](#) 읽어보세요. [prisma-examples](#) 리포지토리에 바로 실행할 수 있는 REST 및 GraphQL 예제도 있습니다.

시작하기

이 레시피에서는 NestJS와 Prisma를 처음부터 시작하는 방법을 배웁니다. 데이터베이스에서 데이터를 읽고 쓸 수 있는 REST API를 사용하여 샘플 NestJS 애플리케이션을 빌드할 것입니다.

이 가이드에서는 데이터베이스 서버를 설정하는 데 드는 오버헤드를 줄이기 위해 [SQLite](#) 데이터베이스를 사용합니다. PostgreSQL 또는 MySQL을 사용하더라도 이 가이드를 따를 수 있으며, 적절한 위치에서 이러한 데이터베이스 사용에 대한 추가 지침을 확인할 수 있습니다.

정보 참고 기존 프로젝트가 이미 있고 Prisma로 마이그레이션을 고려하고 있다면 [기존 프로젝트에 Prisma 추가 가이드를](#) 따르세요. TypeORM에서 마이그레이션하는 경우 [TypeORM에서 Prisma로 마이그레이션하기 가이드를](#) 참조하세요.

NestJS 프로젝트 생성

시작하려면 NestJS CLI를 설치하고 다음 명령을 사용하여 앱 스켈레톤을 생성합니다:

```
$ npm install -g @nestjs/cli  
등지 새 헬로 프리즘
```

이 명령으로 생성된 프로젝트 파일에 대해 자세히 알아보려면 [첫 단계](#) 페이지를 참조하세요. 이제 `npm start`를 실행하여 애플리케이션을 시작할 수 있다는 점도 참고하세요. <http://localhost:3000/>에서 실행되는 REST API는 현재 `src/app.controller.ts`에 구현된 단일 경로를 제공합니다. 이 가이드에서는 사용자 및 글에 대한 데이터를 저장하고 검색하는 추가 경로를 구현할 것입니다.

Prisma 설정

프로젝트에 개발 종속 요소로 Prisma CLI를 설치하여 시작하세요:

```
$ cd hello-prisma  
$ npm 설치 프리즈마 --save-dev
```

다음 단계에서는 [프리즈마 CLI를](#) 활용하겠습니다. 모범 사례로, 접두사 앞에 `npx`를 붙여 로컬에서 CLI를 호출하는 것이 좋습니다:

```
엔엑스피 프리즈마
```

▶ 실을 사용하는 경우 확장

Yarn을 사용하는 경우 다음과 같이 Prisma CLI를 설치할 수 있습니다:

```
yarn 추가 프리즈마 --dev
```

설치가 완료되면 앞에 `yarn`을 붙여 호출할 수 있습니다:

```
원사 프리즈마
```

이제 Prisma CLI의 `init` 명령을 사용하여 초기 Prisma 설정을 생성합니다:

```
npx 프리즈마 초기화
```

이 명령은 다음 내용을 포함하는 새 `prisma` 디렉터리를 만듭니다:

- `schema.prisma`: 데이터베이스 연결을 지정하고 데이터베이스 스키마를 포함합니다.
- `.env`: 일반적으로 데이터베이스 자격 증명을 환경 변수 그룹에 저장하는 데 사용되는 `dotenv` 파일입니다.

데이터베이스 연결 설정

데이터베이스 연결은 `schema.prisma` 파일의 `데이터 소스` 블록에서 구성됩니다. 기본적으로 `포스트그레스큐엘`로 설정되어 있지만 이 가이드에서는 SQLite 데이터베이스를 사용하므로 `데이터 소스` 블록의 `공급자` 필드를 `sqlite`로 조정해야 합니다:

```
데이터 소스 db { 공급자 =
  "sqlite"
  url= env( "DATABASE_URL" )
}
```

```
제너레이터 클라이언트 {
  공급자 = "prisma-client-js"
}
```

이제 `.env`를 열고 `DATABASE_URL` 환경 변수를 조정하여 다음과 같이 표시되도록 합니다:

```
DATABASE_URL="file:./dev.db"
```

구성한 ConfigModule이 있는지 확인하세요. 그렇지 않으면 `.env`에서 `DATABASE_URL` 변수가 선택되지 않습니다.

SQLite 데이터베이스는 단순한 파일이므로 서버가 필요하지 않습니다. 따라서 호스트와 포트로 연결 URL을 구성하는 대신 로컬 파일(이 경우 `dev.db`)을 가리키기만 하면 됩니다. 이 파일은 다음 단계에서 생성됩니다.

▶ PostgreSQL 또는 MySQL을 사용하는 경우 확장하기

PostgreSQL 및 MySQL을 사용하는 경우 데이터베이스 서버를 가리키도록 연결 URL을 구성해야 합니다. 필요한 연결 URL 형식에 대한 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

PostgreSQL

PostgreSQL을 사용하는 경우 다음과 같이 `schema.prisma` 및 `.env` 파일을 조정해야 합니다:

`schema.prisma`

```
데이터 소스 DB {
    공급자 = "postgresql"
    url= env("DATABASE_URL")
}

제너레이터 클라이언트 {
    공급자 = "prisma-client-js"
}
```

`.env`

```
DATABASE_URL="postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=SCHEMA"
```

모든 대문자로 된 자리 표시자의 철자를 데이터베이스 자격 증명으로 바꿉니다. `SCHEMA` 자리 표시자에 무엇을 입력해야 할지 잘 모르겠다면 기본값인 `public`을 입력하는 것이 가장 좋습니다:

```
DATABASE_URL="postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=public"
```

PostgreSQL 데이터베이스를 설정하는 방법을 배우고 싶으시다면, [Heroku에서 무료 PostgreSQL 데이터베이스 설정](#) 가이드를 참조하세요.

MySQL

MySQL을 사용하는 경우 다음과 같이 `schema.prisma` 및 `.env` 파일을 조정해야 합니다:

schema.prisma

```
데이터 소스 db {
    공급자
    = "mysql"
    url= env("DATABASE_URL")
}

제너레이터 클라이언트 {
    공급자 = "prisma-client-js"
}
```

.env

```
DATABASE_URL="mysql://USER:PASSWORD@HOST:PORT/DATABASE"
```

모든 대문자로 표기된 자리 표시자를 데이터베이스 자격 증명으로 바꿉니다. Prisma 마이그레이션으로 두 개의 데이터베이스 테이블 만들기

이 섹션에서는 [Prisma 마이그레이션을](#) 사용하여 데이터베이스에 두 개의 새 테이블을 생성합니다. 프리즈마 마이그레이션

은 프리즈마 스키마에서 선언적 데이터 모델 정의를 위한 SQL 마이그레이션 파일을 생성합니다. 이러한 마이그레이션 파일은 완전히 사용자 정의할 수 있으므로 기본 데이터베이스의 추가 기능을 구성하거나 시딩과 같은 추가 명령을 포함할 수 있습니다.

다음 두 모델을 `schema.prisma` 파일에 추가합니다:

```
모델 사용자 {
    id      Int      기본값(자동증가()) id
    이메일  문자열
    이름    문자열? 고유
} 게시물 게시물[]

모델 포스트 {
    id          Int  문   기본값(자동증가()) id
    title       자열
    콜렉션    부울인자? @default(false)
    작성자     사용자   관계(필드: [authorId], 참조: [id])
} authorId ? Int?
```

Prisma 모델이 준비되면 SQL 마이그레이션 파일을 생성하고 데이터베이스에 대해 실행할 수 있습니다. 터미널에서 다음 명령을 실행합니다:



```
$ npx prisma 마이그레이션 개발 --이름 초기화
```

이 `prisma 마이그레이션 개발` 명령은 SQL 파일을 생성하고 데이터베이스에 대해 직접 실행합니다. 이 경우 기존 `prisma` 디렉터리에 다음과 같은 마이그레이션 파일이 생성되었습니다:

```
트리 프리즘 프리즘
└── dev.db
└── 마이그레이션
    └── 20201207100915_init
        └── migration.sql
└── schema.prisma
```

▶ 확장하여 생성된 SQL 문을 확인합니다.

다음 표는 SQLite 데이터베이스에서 생성되었습니다:

```
-- CreateTable CREATE
TABLE "User" (
    "id" INTEGER NOT NULL PRIMARY KEY 자동 인크루먼트,
    "email" TEXT NOT NULL,
    "이름" 텍스트
);

-- CreateTable CREATE
TABLE "Post" (
    "id" INTEGER NOT NULL PRIMARY KEY 자동인크립션, "title"
TEXT NOT NULL,
    "콘텐츠" 텍스트,
    "published" BOOLEAN DEFAULT false,
    "authorId" INTEGER,

    FOREIGN KEY("authorId") REFERENCES "User"("id") ON DELETE SET NULL ON
UPDATE CASCADE
);

-- CreateIndex
"User"("이메일")에 고유 인덱스 "User.email_unique"를 만듭니다;
```

프리즈마 클라이언트 설치 및 생성

Prisma 클라이언트는 Prisma 모델 정의에서 생성되는 유형 안전 데이터베이스 클라이언트입니다. 이러한 접근 방

식 덕분에 Prisma 클라이언트는 모델에 맞게/특별히 맞춤화된 CRUD 작업을 노출할 수 있습니다.

프로젝트에 프리즈마 클라이언트를 설치하려면 터미널에서 다음 명령을 실행하세요:



```
npm 설치 @prisma/client
```

설치하는 동안 Prisma는 자동으로 `prisma 생성` 명령을 호출합니다. 앞으로는 Prisma 모델을 변경할 때마다 이 명령을 실행하여 생성된 Prisma 클라이언트를 업데이트해야 합니다.

정보 `prisma 생성` 명령은 프리즈마 스키마를 읽고 생성된 프리즈마 클라이언트 라이브러리를 `node_modules/@prisma/client` 내부에 업데이트합니다.

NestJS 서비스에서 프리즈마 클라이언트 사용

이제 프리즈마 클라이언트로 데이터베이스 쿼리를 전송할 수 있습니다. 프리즈마 클라이언트로 쿼리를 작성하는 방법에 대해 자세히 알아보려면 [API 설명서](#)를 확인하세요.

NestJS 애플리케이션을 설정할 때 서비스 내에서 데이터베이스 쿼리를 위한 Prisma 클라이언트 API를 추상화할 수 있습니다. 시작하려면 `PrismaClient` 인스턴스화 및 데이터베이스 연결을 처리하는 새 `PrismaService`를 생성하면 됩니다.

`src` 디렉터리 내에 `prisma.service.ts`라는 새 파일을 생성하고 다음 코드를 추가합니다:

```
'@nestjs/common'에서 { Injectable, OnModuleInit }를 임포트하고,
'@prisma/client'에서 { PrismaClient }를 임포트합니다;

@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
  async onModuleInit() {
    await this.$connect();
  }
}
```

정보 참고 `onModuleInit`은 선택 사항이며, 이를 생략하면 Prisma는 데이터베이스에 처음 호출할 때 느리게 연결됩니다.

다음으로 Prisma 스키마에서 `사용자` 및 `게시물` 모델에 대한 데이터베이스 호출을 수행하는 데 사용할 수 있는 서비스를 작성할 수 있습니다.

여전히 `src` 디렉터리 안에 `user.service.ts`라는 새 파일을 만들고 다음 코드를 추가합니다:

'@nestjs/common'에서 { Injectable }을 가져오고,
'./prisma.service'에서 { PrismaService }를 가져오고,
'@prisma/client'에서 { 사용자, 프리즈마 }를 가져옵니다;

```
@Injectable()  
사용자 서비스 클래스 내보내기 {  
  constructor(private prisma: PrismaService) {}  
  
  async user(
```

```
사용자 위치 고유 입력: Prisma.UserWhereUniqueInput,
): Promise<사용자 | null> {
  return this.prisma.user.findUnique({
    where: userWhereUniqueInput,
  });
}

async users(params: {
  skip?: 숫자; take?:
  숫자;
  cursor?: Prisma.UserWhereUniqueInput;
  where?: Prisma.UserWhereInput;
  orderBy?: Prisma.UserOrderByWithRelationInput;
}): 약속<사용자[]> {
  const { skip, take, cursor, where, orderBy } = params;
  return this.prisma.user.findMany({
    건너뛰기,
    취하다, 커
    서, 어디,
    주문기준,
  });
}

async createUser(데이터: Prisma.UserCreateInput): Promise<User> {
  return this.prisma.user.create({
    데이터,
  });
}

async updateUser(params: {
  where: Prisma.UserWhereUniqueInput;
  데이터: Prisma.UserUpdateInput;
}): 약속<사용자> {
  const { where, data } = params;
  return this.prisma.user.update({
    데이터,
    위치,
  });
}

async deleteUser(where: Prisma.UserWhereUniqueInput): Promise<User> {
  return this.prisma.user.delete({
    어디에,
  });
}
```

프리즈마 클라이언트에서 생성된 유형을 사용하여 서비스에 노출되는 메소드가 올바르게 입력되었는지 확인합니다. 따라서 모델을 입력하고 추가 인터페이스 또는 DTO 파일을 생성하는 번거로움을 줄일 수 있습니다.

이제 포스트 모델에 대해서도 동일한 작업을 수행합니다.

여전히 src 디렉터리 안에 post.service.ts라는 새 파일을 만들고 다음 코드를 추가합니다:

```
'@nestjs/common'에서 { Injectable } 가져오기;
'./prisma.service'에서 { PrismaService } 가져오기;
'@prisma/client'에서 { Post, Prisma } 가져오기;

@Injectable()
내보내기 클래스 PostService {
  constructor(private prisma: PrismaService) {}

  비동기 포스트(
    postWhereUniqueInput: Prisma.PostWhereUniqueInput,
  ): 약속<포스트 | 널> {
    return this.prisma.post.findUnique({
      where: postWhereUniqueInput,
    });
  }

  async posts(params: {
    skip?: 숫자; take?:
    숫자;
    cursor?: Prisma.PostWhereUniqueInput;
    where?: Prisma.PostWhereInput;
    orderBy?: Prisma.PostOrderByWithRelationInput;
  }): Promise<Post[]> {
    const { skip, take, cursor, where, orderBy } = params;
    return this.prisma.post.findMany({
      건너뛰기,
      취하다, 커
      서, 어디,
      주문기준,
    });
  }

  async createPost(데이터: Prisma.PostCreateInput): Promise<Post> {
    return this.prisma.post.create({
      데이터,
    });
  }

  async updatePost(params: {
    where: Prisma.PostWhereUniqueInput;
```

```
데이터: Prisma.PostUpdateInput;
}): 약속<포스트> {
  const { 데이터, where } = 매개변수;
  return this.prisma.post.update({
    데이터,
    위치,
  });
}
```

```
async deletePost(where: Prisma.PostWhereUniqueInput): Promise<Post> {
  return this.prisma.post.delete({
    어디에,
  });
}
```

현재 [사용자 서비스](#) 및 [포스트 서비스](#)는 프리즈마 클라이언트에서 사용할 수 있는 CRUD 쿼리를 래핑합니다. 실제 애플리케이션에서 서비스는 애플리케이션에 비즈니스 로직을 추가하는 장소이기도 합니다. 예를 들어 [사용자 서비스](#) 내에 사용자의 비밀번호 업데이트를 담당하는 [updatePassword](#)라는 메서드가 있을 수 있습니다.

기본 앱 컨트롤러에서 REST API 경로 구현하기

마지막으로 이전 섹션에서 만든 서비스를 사용하여 앱의 다양한 경로를 구현합니다. 이 가이드에서는 모든 경로를 이미 존재하는 [AppController](#) 클래스에 넣겠습니다.

[app.controller.ts](#) 파일의 내용을 다음 코드로 바꿉니다:

```
가져 오기 {
  Controller,
  Get,
  매개변수,
  포스트,
  본문, 넣
  기, 삭제
}

}를 '@nestjs/common'에서 가져옵니다;
'./user.service'에서 { UserService }를 가져오고,
'./post.service'에서 { PostService }를 가져옵니다;
'@prisma/client'에서 { 사용자 사용자모델, 포스트 포스트모델 }을 가져옵니다;
```

컨트롤러()

```
내보내기 클래스 AppController {

  생성자(
    비공개 읽기 전용 사용자 서비스: UserService, 비공
    개 읽기 전용 postService: 포스트서비스,
  ) {}

  @Get('post/:id')
  async getPostById(@Param('id') id: 문자열): Promise<PostModel> {
    return this.postService.post({ id: Number(id) });
  }

  @Get('feed')
  비동기 getPublishedPosts(): Promise<PostModel[]> {
    return this.postService.posts({
      where: { 게시됨: true },
    });
  }
}
```

```
@Get('filtered-posts/:searchString')
async getFilteredPosts(
  Param('searchString') searchString: 문자열,
): Promise<PostModel[]> {
  return this.postService.posts({
    where: {
      또는: [
        {
          title: { 포함: searchString },
        },
        {
          콘텐츠: { 포함: 검색 문자열 },
        },
      ],
    },
  });
}

@Post('post')
async createDraft(
  Body() postData: { 제목: 문자열; 내용?: 문자열; 작성자 이메일: 문자열 },
): 약속<포스트모델> {
  const { title, content, authorEmail } = postData;
  return this.postService.createPost({
    제목, 콘텐츠, 작
    성자: {
      연결합니다: { 이메일: 작성자 이메일 },
    },
  });
}

@Post('user')
async signupUser(
  @Body() 사용자데이터: { 이름?: 문자열; 이메일: 문자열 },
): 약속<사용자 모델> {
  this.userService.createUser(userData)를 반환합니다;
}

@Put('publish/:id')
async publishPost(@Param('id') id: 문자열): Promise<PostModel> {
  return this.postService.updatePost({
    where: { id: Number(id) },
    data: { published: true },
  });
}

삭제('post/:id')
```

```
async deletePost(@Param('id') id: 문자열): Promise<PostModel> {
    return this.postService.deletePost({ id: Number(id) });
}
```

이 컨트롤러는 다음 경로를 구현합니다:

GET

- `/post/:id`: 아이디로 단일 게시물 가져오기
- `/posts`: 게시물 모든 글 가져오기
- `/filter-posts/:검색_문자열`: 제목 또는 콘텐츠로 글 필터링

POST

- `/post`: 새 글 작성
 - 제목: 문자열(필수): 글의 제목
 - 콘텐츠입니다: 문자열(선택 사항): 글의 콘텐츠
 - 작성자 이메일: 문자열(필수): 글을 작성한 사용자의 이메일입니다.
- `/user`: 새 사용자 만들기

본문:

- 이메일: 문자열(필수): 사용자의 이메일 주소
- 이름: 문자열(선택 사항): 사용자의 이름

PUT

- `/publish/:id`: 해당 아이디로 글 게시

삭제

- `/post/:id`: 해당 아이디로 글 삭제

요약

이 레시피에서는 Prisma를 NestJS와 함께 사용하여 REST API를 구현하는 방법을 배웠습니다. API의 경로를 구현하는 컨트롤러는 `PrismaService`를 호출하고, 이 컨트롤러는 다시 Prisma 클라이언트를 사용하여 들어오는 요청의 데이터 요구를 충족하기 위해 데이터베이스에 쿼리를 보냅니다.

Prisma에서 NestJS를 사용하는 방법에 대해 자세히 알아보려면 다음 리소스를 확인하세요:

- ◆ [NestJS 및 프리즈마](#)
- ◆ [바로 실행 가능한 REST 및 GraphQL용 예제 프로젝트](#)
- ◆ [프로덕션 지원 스타터 키트](#)
- ◆ [비디오: Prisma로 NestJS를 사용하여 데이터베이스에 액세스하기\(5분\)](#) 작성자: 마크 스템머요한(Marc Stammerjohann)

정적 제공

SPA(단일 페이지 애플리케이션)와 같은 정적 콘텐츠를 제공하기 위해 `ServeStaticModule`을 사용할 수 있습니다. 를 사용하여 `@nestjs/serve-static` 패키지를 생성합

니다. 설치

먼저 필요한 패키지를 설치해야 합니다:

```
npm install --save @nestjs/serve-static
```

부트스트랩

설치 프로세스가 완료되면 `ServeStaticModule`을 루트 `AppModule`로 가져올 수 있습니다. 를 생성하고 구성 객체를 `forRoot()` 메서드에 전달하여 구성합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./app.controller'에서 { AppController }를 임포트하고,
'./app.service'에서 { AppService }를 임포트합니다;
'@nestjs/serve-static'에서 { ServeStaticModule }을 가져오고,
'path'에서 { join }을 가져옵니다;

모듈({ import: [
  ServeStaticModule.forRoot({
    rootPath: join(__dirname, '..', 'client'),
  }),
],
컨트롤러: [AppController], 공급자: [
  앱서비스],
})
내보내기 클래스 AppModule {}
```

이 상태로 정적 웹사이트를 빌드하고 루트 경로로 지정된 위치에 콘텐츠를 배치합니다.

속성을 설정

합니다. 구성

ServeStaticModule은 다양한 옵션으로 구성하여 동작을 사용자 정의할 수 있습니다. 설정할 수 있습니다. 경로를 사용하여 정적 앱을 렌더링하고, 제외 경로를 지정하고, 캐시 제어 응답 헤더 설정을 활성화 또는 비활성화 할 수 있습니다. 전체 옵션 목록은 [여기에서 확인하세요](#).

경고 정적 앱의 기본 렌더경로는 *(모든 경로)이며, 모듈은 응답으로 "index.html" 파일을 전송합니다. 이를 통해 SPA에 대한 클라이언트 측 라우팅을 생성할 수 있습니다. 컨트롤러에 지정된 경로는 서버로 폴백됩니다. 이 동작을 다른 옵션과 결합하여 `serveRoot`, `renderPath`를 설정하여 변경할 수 있습니다.

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

둥지 사령관

독립형 애플리케이션 문서를 확장하면 일반적인 Nest 애플리케이션과 유사한 구조로 명령줄 애플리케이션을 작성할 수 있는 [nest-commander](#) 패키지도 있습니다.

정보 nest-commander는 타사 패키지이며 NestJS 핵심 팀 전체가 관리하지 않습니다. 라이브러리와 관련된 문제가 발견되면 [해당 리포지토리에](#) 보고해 주세요.

설치

다른 패키지와 마찬가지로 사용하려면 먼저 설치해야 합니다.

```
$ npm i nest-commander
```

명령 파일

[nest-commander](#)을 사용하면 클래스의 경우 [@Command\(\)](#) 데코레이터를, 해당 클래스의 메서드의 경우 [@Option\(\)](#) 데코레이터를 통해 [데코레이터](#)를 사용하여 새 명령줄 애플리케이션을 쉽게 작성할 수 있습니다. 모든 명령 파일은 [CommandRunner](#) 추상 클래스를 구현해야 하며 [@Command\(\)](#) 데코레이터로 데코레이션해야 합니다.

모든 명령은 Nest에서 [@Injectable\(\)](#)로 간주되므로 일반적인 의존성 주입은 여전히 예상대로 작동합니다. 주목해야 할 것은 각 명령이 구현해야 하는 [CommandRunner](#) 추상 클래스뿐입니다. [CommandRunner](#) 추상 클래스는 모든 명령에 [Promise<void>](#)를 반환하고 매개변수 [string\[\]](#), [Record<string, any>](#)를 받는 [run](#) 메서드를 갖도록 합니다. [실행](#) 명령은 모든 로직을 시작할 수 있는 곳이며, 옵션 플래그와 일치하지 않는 매개변수를 받아 배열로 전달하므로 여러 매개변수로 작업하려는 경우에 대비해 사용할 수 있습니다. 옵션의 경우 [Record<string, any>](#), 이러한 프로퍼티의 이름은 [@Option\(\)](#) 데코레이터에 지정된 이름 프로퍼티와 일치하고, 그 값은 옵션 핸들러의 반환과 일치합니다. 더 나은 유형 안전성을 원한다면 옵션에 대한 인터페이스도 만들 수 있습니다.

명령 실행

NestJS 애플리케이션에서 [NestFactory](#)를 사용하여 서버를 생성하고 [listen](#)을 사용하여 서버를 실행하는 것과

유사하게, `nest-commander` 패키지는 서버를 실행하기 위해 사용하기 쉬운 API를 노출합니다.

`CommandFactory`를 가져와서 정적 메서드 `run`을 사용하고 애플리케이션의 루트 모듈에 전달하세요. 이것은 아마도 아래와 같이 보일 것입니다.

```
'nest-commander'에서 { CommandFactory }를 임포트하고,  
'./app.module'에서 { AppModule }을 임포트합니다;
```

```
비동기 함수 부트스트랩() {  
    await CommandFactory.run(AppModule);  
}
```

```
부트스트랩();
```

기본적으로 `CommandFactory`를 사용할 때 Nest의 로거는 비활성화되어 있습니다. 하지만 `실행` 함수의 두 번째 인수로 로거를 제공할 수 있습니다. 사용자 정의 NestJS 로거를 제공하거나 유지하려는 로그 수준의 배열을 제공할 수 있습니다. Nest의 오류 로그만 인쇄하려는 경우 여기에 `['error']`를 제공하는 것이 유용할 수 있습니다.

```
'nest-commander'에서 { CommandFactory }를 임포트하고,  
'./app.module'에서 { AppModule }을 임포트합니다;  
import { LogService } './log.service';  
  
비동기 함수 부트스트랩() {  
    await CommandFactory.run(AppModule, new LogService());  
  
    // 또는 Nest의 경고와 오류만 출력하려면  
    CommandFactory.run(AppModule, ['경고', '오류'])을 기다립니다;  
}  
  
부트스트랩();
```

그게 다입니다. 내부적으로 `CommandFactory`는 `NestFactory`를 호출하고 필요할 때 `app.close()`를 호출하므로 메모리 누수에 대해 걱정할 필요가 없습니다. 오류 처리를 추가해야 하는 경우, `실행` 명령을 래핑하는 `try/catch`를 사용하거나 `.catch()` 메서드를 `부트스트랩()` 호출에 연결할 수 있습니다.

테스트

아주 쉽게 테스트할 수 없다면 아주 멋진 명령줄 스크립트를 작성하는 것이 무슨 소용이 있을까요? 다행히도 `nest-commander`에는 NestJS 에코시스템에 완벽하게 맞는 몇 가지 유ти리티를 사용할 수 있으며, 모든 네슬링이 집처럼 편안하게 사용할 수 있습니다. 테스트 모드에서 명령을 빌드할 때 `CommandFactory`를 사용하는 대신 `CommandTestFactory`를 사용하고 메타데이터를 전달할 수 있는데, 이는 `@nestjs/testing`의 `Test.createTestingModule()` 작동하는 방식과 매우 유사합니다. 실제로 내부적으로 이 패키지를 사용합니다. 또한 `compile()`을 호출하기 전에 `overrideProvider` 메서드를 체인에 연결할 수 있으므로 테스트에서 바로 DI 조각을 교체할 수 있습니다.

모든 것을 종합하기

다음 클래스는 하위 명령을 기본으로 받거나 직접 호출할 수 있는 CLI 명령과 동일하며, `-n`, `-s` 및 `-b`(긴 플래그와

함께)가 모두 지원되고 각 옵션에 대한 사용자 정의 구문 분석기가 있습니다. 커맨더와 마찬가지로 `--help` 플래그도 지원됩니다.

```
'nest-commander'에서 { Command, CommandRunner, Option } 가져오기;
```

```
'./log.service'에서 { LogService } 가져오기;
```

```
인터페이스 BasicCommandOptions { 문자열
```

```
? : 문자열;
```

```
부울?: 부울; 숫자?: 숫자
```

```
;
```

```
}

Command({ name: 'basic', description: '매개변수 파싱' }) export
class BasicCommand extends CommandRunner {
  constructor(private readonly logService: LogService) {
    super()
  }

비동기 실행(
  passedParam: 문자열[], options?
  기본 명령 옵션,
): 약속<무효> {
  if (options?.boolean !== undefined && options?.boolean !== null) {
    this.runWithBoolean(passedParam, options.boolean);
  } else if (options?.number) {
    this.runWithNumber(passedParam, options.number);
  } else if (options?.string) {
    this.runWithString(passedParam, options.string);
  } else {
    this.runWithNone(passedParam);
  }
}

@Option({
  플래그: '-n, --number [숫자]', 설명: '기본 숫자 구문 분석기',
})
parseNumber(val: 문자열): number {
  return Number(val);
}

@Option({
  플래그: '-s, --string [문자열]', 설명: '문자열 반환',
})
parseString(val: 문자열): 문자열 {
  return val;
}

@Option({
  플래그: '-b, --boolean [부울]', 설명: '부울 구문 분석기',
})
parseBoolean(val: 문자열): boolean {
  return JSON.parse(val);
}
```

```
runWithString(매개변수: 문자열[], 옵션: 문자열): void { this.logService.log({  
    매개변수, 문자열: 옵션  
});  
  
runWithNumber(매개변수: 문자열[], 옵션: 숫자): void { this.logService.log({  
    매개변수, 숫자: 옵션  
});  
}
```

```
runWithBoolean(param: string[], option: boolean): void {
  this.logService.log({ param, boolean: option });
}

runWithNone(매개변수: 문자열[]): void {
  this.logService.log({ 매개변수 });
}
}
```

명령 클래스가 모듈에 추가되었는지 확인합니다.

```
모듈({
  공급자: [LogService, BasicCommand],
})
내보내기 클래스 AppModule {}
```

이제 main.ts에서 CLI를 실행하려면 다음을 수행할 수 있습니다.

```
비동기 함수 부트스트랩() {
  await CommandFactory.run(AppModule);
}

부트스트랩();
```

이렇게 하면 명령줄 애플리케이션이 완성됩니다. 자세한 정

보

자세한 정보, 예제 및 API 설명서를 보려면 [nest-commander 문서 사이트](#)를 방문하세요.

비동기 로컬 스토리지

AsyncLocalStorage는 함수 매개변수로 명시적으로 전달하지 않고도 애플리케이션을 통해 로컬 상태를 전파할 수 있는 대체 방법을 제공하는 [Node.js API](#)(`async_hooks` API 기반)입니다. 다른 언어의 스레드 로컬 스토리지와 유사합니다.

비동기 로컬 저장소의 주요 아이디어는 일부 함수 호출을 `AsyncLocalStorage#run` 호출로 [래핑](#)할 수 있다는 것입니다. 래핑된 호출 내에서 호출되는 모든 코드는 각 호출 체인에 고유한 동일한 [저장소](#)에 액세스하게 됩니다.

NestJS의 맥락에서 이는 요청의 라이프사이클 내에서 요청의 나머지 코드를 래핑할 수 있는 위치를 찾을 수 있다면 해당 요청에만 표시되는 상태에 액세스하고 수정할 수 있다는 의미이며, 이는 요청 범위 제공자 및 일부 제한 사항에 대한 대안으로 사용될 수 있습니다.

또는 ALS를 사용하여 컨텍스트를 서비스 전체에 명시적으로 전달하지 않고 시스템의 일부(예: [트랜잭션 객체](#))에 대해서만 컨텍스트를 전파하여 격리 및 캡슐화를 강화할 수 있습니다.

사용자 지정 구현

NestJS 자체는 `AsyncLocalStorage`에 대한 내장 추상화를 제공하지 않으므로 전체 개념을 더 잘 이해하기 위해 가장 간단한 HTTP 사례에 대해 직접 구현하는 방법을 살펴 보겠습니다:

정보 정보 기성 전용 패키지에 대해서는 아래를 계속 읽어보세요.

- 먼저 공유 소스 파일에 `AsyncLocalStorage`의 새 인스턴스를 생성합니다. NestJS를 사용하고 있으므로 사용자 정의 공급자를 사용하여 모듈로 전환해 보겠습니다.

```
@@파일명(als.module) @Module({
  제공자: [
    {
      제공: AsyncLocalStorage,
      newValue: new AsyncLocalStorage(),
    },
  ],
  내보내기: [AsyncLocalStorage],
})  
내보내기 클래스 AlsModule {}
```

정보 힌트 `AsyncLocalStorage`는 [async_hook](#)에서 가져옵니다.

. 여기서는 HTTP에만 관심이 있으므로 미들웨어를 사용하여 다음 함수를 `AsyncLocalStorage#run`으로 래핑해 보겠습니다. 미들웨어는 요청이 가장 먼저 닿는 곳이기 때문에 모든 인핸서와 나머지 시스템에서 저장소를 사용할 수 있게 됩니다.

```
@@파일명(app.module) @Module({  
    임포트: [AlsModule] 공급자:  
        [CatService], 컨트롤러:  
        [CatController],  
})  
내보내기 클래스 AppModule은 NestModule을 구현합니다 { 생성자(  
    // 모듈 생성자에 AsyncStorage를 비공개 읽기 전용으로 삽입합니다  
    : AsyncStorage  
) {}  
  
configure(consumer: MiddlewareConsumer) {  
    // 미들웨어, 소비자 바인딩  
    .apply((req, res, next) => {  
        // 몇 가지 기본값으로 스토어 채우기  
        // 요청에 따라, const  
        store = {  
            userId: req.headers['X-user-id'],  
        };  
        // 그리고 "next" 함수를 콜백으로 전달합니다.  
        //를 스토어와 함께 "als.run" 메서드에 추가합니다. this.als.run(store, ()  
        => next());  
    })  
    // 모든 경로에 등록합니다 (Fastify의 경우 '(.*)' 사용).  
    .forRoutes('*');  
}  
}  
@@switch  
@Module({  
    임포트: [AlsModule] 공급자:  
        [CatService], 컨트롤러:  
        [CatController],  
})  
@Dependencies(AsyncLocalStorage)  
export 클래스 AppModule {  
    constructor(as) {  
        // 모듈 생성자, this.als = als에 AsyncStorage를 삽입합니다.  
    }  
  
    configure(consumer) {  
        // 미들웨어, 소비자 바인딩  
        .apply((req, res, next) => {  
            // 몇 가지 기본값으로 스토어 채우기  
            // 요청에 따라, const
```

```
store = {
  userId: req.headers['X-user-id'],
};

// 그리고 "next" 함수를 콜백으로 전달합니다.

//를 스토어와 함께 "als.run" 메서드에 추가합니다.
```

```

        this.als.run(store, () => next());
    })
    // 모든 경로에 등록합니다 (Fastify의 경우 '(.*)' 사용).
    .forRoutes('*');
}
}

```

. 이제 요청의 라이프사이클 내 어디서나 로컬 스토어 인스턴스에 액세스할 수 있습니다.

```

@@파일명(cat.service)
@Injectable()
내보내기 클래스 CatService { 생성

자(
    // 제공된 ALS 인스턴스를 삽입할 수 있습니다:
    AsyncLocalStorage, 비공개 읽기 전용 catRepository:
    CatRepository,
) {}

getCatForUser() {
    // "getStore" 메소드는 항상
    // 주어진 요청과 연관된 저장소 인스턴스입니다. const userId =
    this.als.getStore()["userId"] as number; return
    this.catRepository.getForUser(userId)를 반환합니다;
}

@@스위치
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
    constructor(als, catRepository) {
        // 제공된 ALS 인스턴스를 삽입할 수 있습니다.
        this.catRepository = 고양이 저장소
    }

    getCatForUser() {
        // "getStore" 메소드는 항상
        // 주어진 요청과 연관된 저장소 인스턴스입니다. const userId =
        this.als.getStore()["userId"] as number; return
        this.catRepository.getForUser(userId)를 반환합니다;
    }
}

```

. 그게 다입니다. 이제 전체 요청을 주입할 필요 없이 요청 관련 상태를 공유할 수 있는 방법이 생겼습니다.

요청 객체입니다.

경고 경고 이 기술은 많은 사용 사례에 유용하지만 본질적으로 코드 흐름을 난독화하므로(암시적 컨텍스트 생성) 책임감 있게 사용해야 하며, 특히 컨텍스트에 맞는 '[신 객체](#)'를 만들지 않도록 주의하세요.

NestJS CLS

[nestjs-cls](#) 패키지는 일반 [AsyncLocalStorage](#)를 사용할 때보다 몇 가지 DX 개선 사항을 제공합니다(CLS는 연속 로컬 스토리지라는 용어의 약어입니다). 이 패키지는 구현을 강력한 타이핑 지원뿐만 아니라 다양한 전송(HTTP뿐만 아니라)에 대해 [저장소](#)를 초기화하는 다양한 방법을 제공하는 [ClsModule](#)로 추상화합니다.

그런 다음 삽입 가능한 [ClsService](#)를 사용하여 스토어에 액세스하거나 [프록시 공급자](#)를 사용하여 비즈니스 로직에서 완전히 추상화할 수 있습니다.

정보 nestjs-cls는 타사 패키지이며 NestJS 코어 팀에서 관리하지 않습니다. 라이브러리와 관련된 문제가 발견되면 [해당 리포지토리](#)에 보고해 주세요.

설치

[nestjs](#) 라이브러리에 대한 피어 종속성을 제외하고는 기본 제공 Node.js API만 사용합니다. 다른 패키지로 설치하세요.

NPM 및 NESTJS-CLS

사용법

위에서 설명한 것과 유사한 기능을 다음과 같이 [nestjs-cls](#)를 사용하여 구현할 수 있습니다:

- . 루트 모듈에서 [ClsModule](#)을 가져옵니다.

```
@@파일명(app.module) @Module({
  수입: [
    // ClsModule 등록, ClsModule.forRoot({
      미들웨어: {
        // 자동으로 마운트
        // 모든 경로에 대한 ClsMiddleware 마운트:
        true,
        // 설정 방법을 사용하여
        // 기본 저장소 값을 제공합니다. 설정:
        (cls, req) => {
          cls.set('userId', req.headers['x-user-id']);
        },
      },
    }),
  ],
  제공자: [CatService], 컨트롤러:
  [CatController],
})
내보내기 클래스 AppModule {}
```

. 그런 다음 `ClsService`를 사용하여 저장소 값에 액세스할 수 있습니다.

```

@@파일명(cat.service)
@Injectable()
내보내기 클래스 CatService { 생성
  자(
    // 제공된 ClsService 인스턴스, 비공개 읽기 전용 cls를 삽입할
    수 있습니다: ClsService,
    비공개 읽기 전용 고양이 저장소: 고양이 저장소,
  ) {}

  getCatForUser() {
    // "get" 메서드를 사용하여 저장된 값을 검색합니다. const userId =
    this.cls.get('userId');
    this.catRepository.getForUser(userId)를 반환합니다;
  }
}

@@스위치
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
  constructor(cls, catRepository) {
    // 제공된 ClsService 인스턴스, this.cls = cls를 삽입할 수
    있습니다.
    this.catRepository = 고양이 저장소
  }

  getCatForUser() {
    // "get" 메서드를 사용하여 저장된 값을 검색합니다. const userId =
    this.cls.get('userId');
    this.catRepository.getForUser(userId)를 반환합니다;
  }
}

```

. `ClsService`에서 관리하는 저장소 값을 강력하게 입력하려면(또한 문자열 키의 자동 제안을 받으려면), 선택적 유형 매개변수 `ClsService<MyClsStore>`를 삽입할 때 사용할 수 있습니다.

내보내기 인터페이스 MyClsStore는 ClsStore를 확장합니다

```
{
  userId: 숫자;
}
```

정보 힌트 패키지가 자동으로 요청 ID를 생성하도록 하고 나중에 `cls.getId()`를 사용하여 액세스하거나

^{9.17} `cls.get(CLSS_REQ)`를 사용하여 전체 요청 객체를 가져올 수도 있습니다.

테스트

ClService는 또 다른 인젝터를 프로바이더이므로 단위 테스트에서 완전히 모의 테스트할 수 있습니다.

그러나 특정 통합 테스트에서는 여전히 실제 `ClsService` 구현을 사용하고 싶을 수 있습니다. 이 경우 컨텍스트 인식 코드 조각을 `ClsService#run` 또는 `ClsService#runWith` 호출로 래핑해야 합니다.

```
describe('CatService', () => {
  let service: CatService
  let cls: ClsService
  const mockCatRepository = createMock<CatRepository>()

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      // 대부분의 테스트 모듈을 평소와 같이 설정합니다: [
      CatService,
      {
        제공: CatRepository 사용값:
        mockCatRepository
      }
    ],
    수입: [
      // 다음만 제공하는 정적 버전의 ClsModule을 가져옵니다.
      //를 호출하지만 어떤 방식으로도 저장소를 설정하지 않습니다. ClsModule
    ],
    }).compile()

    service = module.get(CatService)

    // 나중에 사용할 수 있도록 ClsService도 검색합니다. cls
    = module.get(ClsService)
  })

  describe('getCatForUser', () => {
    it('사용자 아이디를 기반으로 고양이를 검색합니다', async () => {
      const expectedUserId = 42
      mockCatRepository.getForUser.mockImplementationOnce(
        (id) => ({ userId: id })
      )

      // 테스트 호출을 `runWith` 메서드로 감싸기
      // 수작업으로 만든 저장소 값을 전달할 수 있습니다. const cat =
      await cls.runWith(
        { userId: expectedUserId },
        () => service.getCatForUser()
      )

      기대(cat.userId).toEqual(expectedUserId)
    })
  })
})
```

```
    } )  
}
```

자세한 정보

전체 API 문서와 더 많은 코드 예제를 보려면 [NestJS CLS GitHub 페이지](#)를 방문하세요.

오토목

Automock은 단위 테스트를 위한 독립형 라이브러리입니다. 내부적으로 TypeScript Reflection API([반영 메타데이터](#))를 사용하여 모의 객체를 생성하는 Automock은 클래스 외부 종속성을 자동으로 모의하여 테스트 개발을 간소화합니다.

정보 Automock은 타사 패키지이며 NestJS 코어 팀에서 관리하지 않습니다. 라이브러리와 관련된 문 제가 발견되면 해당 리포지토리에 보고해 주세요.

소개

의존성 주입(DI) 컨테이너는 Nest 모듈 시스템의 필수 구성 요소입니다. 이 컨테이너는 테스트와 애플리케이션 실행 중에 모두 활용됩니다. 단위 테스트는 DI 컨테이너 내의 공급자/서비스를 완전히 재정의해야 한다는 점에서 통합 테스트와 같은 다른 유형의 테스트와 다릅니다. 소위 "단위"라고 하는 외부 클래스 종속성(공급자)은 완전히 격리되어야 합니다. 즉, DI 컨테이너 내의 모든 종속성을 모의 객체로 대체해야 합니다. 결과적으로 대상 모듈을 로드하고 그 안에 있는 프로바이더를 교체하는 것은 그 자체로 반복되는 프로세스입니다. Automock은 모든 클래스 외부 프로바이더를 자동으로 모킹하여 테스트 중인 유닛을 완전히 격리함으로써 이 문제를 해결합니다.

설치

```
$ npm i -D @automock/jest
```

오토목은 추가 설정이 필요하지 않습니다.

정보 Jest는 현재 Automock에서 지원하는 유일한 테스트 프레임워크입니다. Sinon은 곧 출시될 예정입니다.

예

생성자 매개변수 세 개를 받는 다음 cats 서비스를 생각해 보세요:

```
@@파일명(cats.service)
'@nestjs/core'에서 { Injectable }을 임포트합니다;

@Injectable()
내보내기 클래스 CatsService { 생

    성자(
        개인 logger: Logger,
        개인 httpService: HttpService, 비공개
        catsDal: CatsDal,
    ) {}

    async getAllCats() {
        const cats = await
this.httpService.get('http://localhost:3000/api/cats');
```

```
    this.logger.log('성공적으로 모든 고양이를 가져왔습니다');

    this.catsDal.saveCats(cats);
}
}
```

이 서비스에는 다음 단위 테스트에 예제로 사용하는 메서드인 `getAllCats`라는 공용 메서드가 하나 포함되어 있습니다:

```
@@파일명(cats.service.spec)
'@automock/jest'에서 { TestBed }를 임포트하고,
'./cats.service'에서 { CatsService }를 임포트합니다;

describe('CatsService 단위 사양', () => {
  underTest: CatsService;
  logger: jest.Mocked<Logger>;
  let httpService: jest.Mocked<HttpService>;
  let catsDal: jest.Mocked<CatsDal>;

  beforeAll(() => {
    const { unit, unitRef } = TestBed.create(CatsService)
      .mock(HttpService)
      .using({ get: jest.fn() })
      .mock(Logger)
      .using({ log: jest.fn() })
      .mock(CatsDal)
      .using({ saveCats: jest.fn() })
      .compile();

    underTest = unit;

    logger = unitRef.get(Logger);
    httpService = unitRef.get(HttpService);
    catsDal = unitRef.get(CatsDal);
  });

  describe('모든 고양이를 구할 때', () => {
    test('then
meet some expectations', async () => {
      httpService.get.mockResolvedValueOnce([{ id: 1, name: 'Catty' }]);
      await catsService.getAllCats();

      expect(logger.log).toBeCalled();
      expect(catsDal).toBeCalledWith([{ id: 1, name: 'Catty' }]);
    });
  });
});
```

정보 jest.Mocked 유ти리티 유형은 Jest 모의 함수의 유형 정의로 래핑된 소스 유형을 반환합니다. ([참조](#))

단위 및 `unitRef` 정보

다음 코드를 살펴보겠습니다:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();
```

`.compile()`을 호출하면 두 개의 프로퍼티, `unit`과 `unitRef`가 있는 객체가 반환됩니다.

단위는 테스트 중인 단위로, 테스트 중인 클래스의 실제 인스턴스입니다.

`unitRef`은 테스트된 클래스의 모의 종속성이 작은 컨테이너에 저장되는 "단위 참조"입니다. 컨테이너의 `.get()` 메서드는 모든 메서드가 자동으로 스텁된(`jest.fn()` 사용) 모킹된 의존성을 반환합니다:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();
let httpServiceMock: jest.Mocked<HttpService> = unitRef.get(HttpService);
```

정보 정보 `.get()` 메서드는 문자열 또는 실제 클래스(`Type`)를 인수로 받을 수 있습니다. 이는 기본적으로 프로바이더가 테스트 중인 클래스에 주입되는 방식에 따라 달라집니다.

다양한 제공업체와 협력

프로바이더는 Nest에서 가장 중요한 요소 중 하나입니다. 서비스, 저장소, 팩토리, 헬퍼 등 많은 기본 Nest 클래스를 프로바이더로 생각할 수 있습니다. 프로바이더의 주요 기능은 `Injectable` 의존성의 형태를 취하는 것입니다.

매개변수 하나를 취하는 다음 `CatsService`를 고려해 보겠습니다. 이 매개변수는 다음 `Logger`의 인스턴스입니다. 인터페이스:

```
export interface Logger {
  log(message: 문자열): void;
}

export class CatsService {
  constructor(private logger: Logger) {}
}
```

TypeScript의 리플렉션 API는 아직 인터페이스 리플렉션을 지원하지 않습니다. Nest는 문자열 기반 인젝션 토큰으로 이 문제를 해결합니다([사용자 정의 공급자](#) 참조):

```
export const MyLoggerProvider = {  
  provide: 'MY_LOGGER_TOKEN',  
  useValue: { ... },  
}  
}
```

```
내보내기 클래스 CatsService {  
    생성자(@Inject('MY_LOGGER_TOKEN') 비공개 읽기 전용 로거: Logger)  
}  
}
```

Automock은 이러한 관행을 따르며 `unitRef.get()` 메서드에서 실제 클래스를 제공하는 대신 문자열 기반 토큰을 제공할 수 있습니다:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();  
  
let loggerMock: jest.Mocked<Logger> = unitRef.get('MY_LOGGER_TOKEN');
```

자세한 정보

자세한 내용은 [Automock GitHub 리포지토리](#)를 참조하세요.

서비스

서비스 컴퓨팅은 클라우드 제공업체가 고객을 대신하여 서버를 관리하면서 온디맨드 방식으로 머신 리소스를 할당하는 클라우드 컴퓨팅 실행 모델입니다. 앱을 사용하지 않을 때는 앱에 할당된 컴퓨팅 리소스가 없습니다. 가격은 애플리케이션([소스](#))이 소비하는 실제 리소스 양을 기준으로 책정됩니다.

서비스 아키텍처를 사용하면 애플리케이션 코드의 개별 기능에만 집중할 수 있습니다. AWS Lambda, Google Cloud Functions, Microsoft Azure Functions와 같은 서비스가 모든 물리적 하드웨어, 가상 머신 운영 체제, 웹 서버 소프트웨어 관리를 처리합니다.

정보 힌트 이 장에서는 서비스 기능의 장단점을 다루거나 클라우드 제공업체의 세부 사항을 자세히 다루지 않습니다.

콜드 스타트

콜드 스타트란 오랜만에 코드를 실행하는 것을 말합니다. 사용하는 클라우드 제공업체에 따라 코드를 다운로드하고 런타임을 부트스트랩하는 것부터 최종적으로 코드를 실행하는 것까지 여러 가지 작업이 포함될 수 있습니다. 이 프로세스는 언어, 애플리케이션에 필요한 패키지 수 등 여러 요인에 따라 상당한 지연 시간을 추가합니다.

콜드 스타트는 중요하며, 우리가 통제할 수 없는 상황도 있지만 가능한 한 짧게 만들기 위해 우리가 할 수 있는 일은 많습니다.

Nest는 복잡한 엔터프라이즈 애플리케이션에 사용하도록 설계된 완전한 프레임워크라고 생각할 수 있지만, 훨씬 "간단한" 애플리케이션(또는 스크립트)에도 적합합니다. 예를 들어, [독립형 애플리케이션](#) 기능을 사용하면 간단한 워커, CRON 작업, CLI 또는 서비스 기능에서 Nest의 DI 시스템을 활용할 수 있습니다.

벤치마크

서비스 함수의 맥락에서 Nest 또는 기타 잘 알려진 라이브러리(예: [express](#))를 사용하는 데 드는 비용을 더 잘 이해하기 위해 Node 런타임이 다음 스크립트를 실행하는 데 얼마나 많은 시간이 필요한지 비교해 보겠습니다:

```
// #1 Express  
'express'에서 *를 express로 가져옵니다;  
  
비동기 함수 부트스트랩() { const  
    app = express();  
    app.get('/', (req, res) => res.send('Hello world!'));  
    await new Promise<void>((resolve) => app.listen(3000, resolve));  
}  
부트스트랩();  
  
// #2 Nest (@nestjs/platform-express 사용)  
'@nestjs/core'에서 { NestFactory } 임포트;  
'./app.module'에서 { AppModule } 임포트;
```

```

비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule, { logger: ['error'] });
  await app.listen(3000);
}

부트스트랩();

// #3 Nest를 독립형 애플리케이션(HTTP 서버 없음)으로 사용하려면
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'./app.module'에서 { AppModule }을 가져오고,
'./app.service'에서 { AppService }를 가져옵니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.createApplicationContext(AppModule, {
    logger: ['error'],
  });
  console.log(app.get(AppService).getHello());
}

부트스트랩();

// #4 원시 Node.js 스크립트 비동

```

```

기 함수 bootstrap() {
  console.log('안녕하세요!');
}

부트스트랩();

```

이 모든 스크립트에는 `tsc`(TypeScript) 컴파일러를 사용했기 때문에 코드가 번들되지 않은 상태로 유지됩니다(
웹팩은 사용되지 않음).

Express	0.0079초(7.9ms)
---------	----------------

Nest(@nestjs/platform-express 포함)	0.1974초
-----------------------------------	---------

(197.4ms) Nest(독립 실행형 애플리케이션)	0.1117초
-------------------------------	---------

(111.7ms)	
-----------	--

원시 Node.js 스크립트	0.0071초(7.1ms)
-----------------	----------------

정보 노트 기계: MacBook Pro Mid 2014, 2.5GHz 쿼드코어 인텔 코어 i7, 16GB 1600MHz DDR3, SSD.

이제 모든 벤치마크를 반복하되 이번에는 웹팩(Nest CLI가 설치되어 있는 경우 `nest build --webpack`)을 실행하

면 됩니다)을 사용하여 애플리케이션을 단일 실행 가능한 JavaScript 파일로 번들링해 보겠습니다. 하지만 Nest CLI와 함께 제공되는 기본 웹팩 구성은 사용하는 대신 다음과 같이 모든 종속 요소([node_modules](#))를 함께 번들로 묶어 보겠습니다:

```
module.exports = (옵션, 웹팩) => { const
  lazyImports = [
    '@nestjs/microservices/microservices-module',
    '@nestjs/websockets/socket-module',
  ];
}

반환 {
```

```

...옵션,
외부: [], 플러그인: [
  ...options.plugins,
  새로운 웹팩.무시 플러그인({
    checkResource(resource) {
      if (lazyImports.includes(resource)) {
        try {
          require.resolve(resource);
        } catch (err) {
          return true;
        }
      }
    }
  },
  ],
],
};

}
;

```

정보 힌트 Nest CLI가 이 구성을 사용하도록 지시하려면 프로젝트의 루트 디렉터리에 새
이 구성 [webpack.config.js](#) 파일을 만드세요.
이 구성을 통해 다음과 같은 혐의를 얻었습니다: Express

0.0068초

(6.8ms)

Nest([@nestjs/platform-express](#) 포함) 0.0815초

(81.5ms) Nest(독립 실행형 애플리케이션) 0.0319초

(31.9ms)

원시 Node.js 스크립트 0.0066초(6.6ms)

정보 노트 기계: MacBook Pro Mid 2014, 2.5GHz 쿼드코어 인텔 코어 i7, 16GB 1600MHz DDR3, SSD.

정보 힌트 추가 코드 최소화 및 최적화 기법(웹팩 플러그인 등 사용)을 적용하여 더욱 최적화할 수 있습니다.

보시다시피 컴파일 방식(그리고 코드 번들링 여부)은 매우 중요하며 전체 시작 시간에 큰 영향을 미칩니다. 웹팩을 사용하면 독립형 Nest 애플리케이션(하나의 모듈, 컨트롤러, 서비스가 포함된 스타터 프로젝트)의 부트스트랩 시간을 평균 32ms까지 단축할 수 있으며, 일반 HTTP, 익스프레스 기반 NestJS 앱의 경우 81.5ms까지 단축할

수 있습니다.

예를 들어 10개의 리소스(\$ nest g 리소스 스키마 = 10개의 모듈, 10개의 컨트롤러, 10개의 서비스, 20개의 DTO 클래스, 50개의 HTTP 엔드포인트 + AppModule을 통해 생성됨)가 있는 더 복잡한 Nest 애플리케이션의 경우, MacBook Pro Mid 2014, 2.5GHz 쿼드 코어 인텔 코어 i7, 16GB 1600MHz DDR3, SSD에서 전체 시작은 약 0.1298초(129.8ms)입니다. 모놀리식 애플리케이션을 서비스 기능으로 실행하는 것은 일반적으로 큰 의미가 없으므로 이 벤치마크는 애플리케이션이 성장함에 따라 부트스트랩 시간이 어떻게 증가할 수 있는지를 보여주는 예시라고 생각하면 됩니다.

런타임 최적화

지금까지 컴파일 시간 최적화에 대해 살펴보았습니다. 이는 애플리케이션에서 프로바이더를 정의하고 네스트 모듈을 로드하는 방식과 무관하지 않으며, 애플리케이션이 커질수록 필수적인 역할을 합니다.

예를 들어 데이터베이스 연결이 [비동기 공급자로](#) 정의되어 있다고 가정해 보겠습니다. 비동기 공급자는 하나 이상의 비동기 작업이 완료될 때까지 애플리케이션 시작을 지연하도록 설계되었습니다. 즉, 서비스 함수가 데이터베이스에 연결하는 데 평균 2초가 걸리는 경우(부트스트랩에서), 엔드포인트는 (콜드 스타트이고 애플리케이션이 아직 실행 중이 아닌 경우) 응답을 다시 보내는 데 최소 2초(연결이 설정될 때까지 기다려야 하므로)가 더 필요합니다.

보시다시피, 부트스트랩 시간이 중요한 서비스 환경에서는 프로바이더를 구성하는 방식이 다소 달라집니다. 또 다른 좋은 예는 특정 시나리오에서만 캐싱을 위해 Redis를 사용하는 경우입니다. 이 경우, 특정 함수 호출에 필요하지 않더라도 부트스트랩 시간이 느려질 수 있으므로 Redis 연결을 비동기 공급자로 정의해서는 안 됩니다.

또한 [이 장에서](#) 설명한 대로 [LazyModuleLoader](#) 클래스를 사용하여 전체 모듈을 지연 로드할 수도 있습니다. 여기에서도 캐싱이 좋은 예입니다. 애플리케이션에 내부적으로 Redis에 연결하고 또한 [CacheService](#)를 내보내 Redis 스토리지와 상호 작용하는 [CacheModule](#)이 있다고 가정해 보겠습니다. 모든 잠재적 함수 호출에 필요하지 않은 경우, 온디맨드 방식으로 느리게 로드하면 됩니다. 이렇게 하면 캐싱이 필요하지 않은 모든 호출에 대해 시작 시간(콜드 스타트 발생 시)을 단축할 수 있습니다.

```
if (request.method === RequestMethod[RequestMethod.GET]) {  
    const { CacheModule } = await import('./cache.module');  
    const moduleRef = await this.lazyModuleLoader.load(() => CacheModule);  
  
    const { CacheService } = await import('./cache.service');  
    const cacheService = moduleRef.get(CacheService);  
  
    cacheService.get(ENDPOINT_KEY)를 반환합니다;  
}
```

또 다른 좋은 예는 특정 조건(예: 입력 인수)에 따라 다른 작업을 수행할 수 있는 웹훅 또는 워커입니다. 이러한 경우 라우트 핸들러 내부에 특정 함수 호출에 적합한 모듈을 느리게 로드하는 조건을 지정하고 다른 모든 모듈을 느리게 로드할 수 있습니다.

```
if (workerType === WorkerType.A) {  
    const { WorkerAModule } = await import('./worker-a.module');  
    const moduleRef = await this.lazyModuleLoader.load(() => WorkerAModule);  
    // ...  
} else if (workerType === WorkerType.B) {  
    const { WorkerBModule } = await import('./worker-b.module');  
    const moduleRef = await this.lazyModuleLoader.load(() => WorkerBModule);  
    // ...  
}
```

통합 예시

애플리케이션의 엔트리 파일(일반적으로 `main.ts` 파일)의 모양은 여러 요인에 따라 달라지므로 모든 시나리오에 적합한 단일 템플릿은 없습니다. 예를 들어, 서비스 기능을 가동하는 데 필요한 초기화 파일은 클라우드 제공업체(AWS, Azure, GCP 등)에 따라 다릅니다. 또한 여러 경로/엔드포인트로 일반적인 HTTP 애플리케이션을 실행할지, 단일 경로만 제공할지(또는 코드의 특정 부분만 실행할지) 여부에 따라 애플리케이션 코드의 모양이 달라집니다(예를 들어, 함수별 엔드포인트 접근 방식의 경우 HTTP 서버 부팅, 미들웨어 설정 등 대신 `NestFactory.createApplicationContext`를 사용할 수 있습니다.).

예시를 보여드리기 위해 Nest(@nestjs/platform-express)를 사용하여 전체 기능을 갖춘 HTTP 라우터 전체를 스피너업(서버리스) 프레임워크(이 경우 AWS Lambda를 대상으로 함)와 통합해 보겠습니다. 앞서 언급했듯이 선택한 클라우드 제공업체와 기타 여러 요인에 따라 코드가 달라집니다.

먼저 필요한 패키지를 설치해 보겠습니다:

```
$ npm i @vendia/serverless-express aws-lambda
$ npm i -D @타입스/aws-lambda 서버리스-오프라인
```

정보 힌트 개발 주기를 단축하기 위해 AWS Lambda 및 API 게이트웨이를 에뮬레이트하는 `서버리스-오프라인` 플러그인을 설치합니다.

설치 프로세스가 완료되면 서비스 프레임워크를 구성하기 위해 `serverless.yml` 파일을 생성해 보겠습니다:

서비스: 서비스 예제 플러그인:

- 서비스 오프라인

공급자: 이름:

AWS

런타임: nodejs14.x

기능: 메인:

핸들러: dist/main.handler 이벤트:

- http:

메서드: 모든 경

로: /

- http:

메서드를 사용합니다:

모든 경로: '{프록시

+}'

정보 힌트 서비스 프레임워크에 대해 자세히 알아보려면 [공식 문서를 참조하세요](#).

이제 `main.ts` 파일로 이동하여 필요한 상용구로 부트스트랩 코드를 업데이트할 수 있습니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@vendia/serverless-express'에서 서비스익스프레스 임포트; 'aws-
lambda'에서 { 콜백, 컨텍스트, 핸들러 } 임포트; './app.module'에서 {
앱모듈 } 임포트;
```

렛 서버: 핸들러;

```
비동기 함수 부트스트랩(): Promise<Handler> {
  const app
    = await NestFactory.create(AppModule);
  await app.init();

  const expressApp = app.getHttpAdapter().getInstance();
  return serverlessExpress({ app: expressApp });
}
```

const 핸들러를 내보냅니다: 핸들러 = async (이벤트:

any,

컨텍스트: 컨텍스트, 콜백:

콜백,

) => {

정보 힌트 여러 서비스 함수를 생성하고 이를 간에 곱통 모듈을 공유하려면 [CLI 모노레포 모드를 사용](#)

하는 경우 좋습니다, 콜백);

경고 경고 `@nestjs/swagger` 패키지를 사용하는 경우 서비스 기능의 맥락에서 제대로 작동하도록 하

기 위해 몇 가지 추가 단계가 필요합니다. 자세한 내용은 이 [스레드](#)를 확인하세요.

그런 다음 `tsconfig.json` 파일을 열고 `esModuleInterop` 옵션을 활성화하여 `@vendia/serverless-express` 패키지가 제대로 로드되도록 합니다.

```
{
  "컴파일러옵션": {
    ...
    "esModuleInterop": true
  }
}
```

이제 애플리케이션을 빌드하고(네스트 빌드 또는 tsc 사용) 서버리스 CLI를 사용하여 람다 함수를 로컬에서 시작할 수 있습니다:

```
$ npm 실행 빌드  
npx 서비스 오프라인
```

애플리케이션이 실행 중이면 브라우저를 열고 [http://localhost:3000/dev/\[ANY_ROUTE\]](http://localhost:3000/dev/[ANY_ROUTE])로 이동합니다(여기서 [ANY_ROUTE]는 애플리케이션에 등록된 모든 엔드포인트).

위 섹션에서는 웹팩을 사용하여 앱을 번들링하면 전체 부트스트랩 시간에 상당한 영향을 미칠 수 있음을 보여주었습니다. 하지만 이 예제에서 작동하도록 하려면 `webpack.config.js` 파일에 몇 가지 추가 구성은 추가해야 합니다. 일반적으로 `핸들러` 함수가 선택되도록 하려면 출력 `libraryTarget` 속성을 `commonjs2`로 변경해야 합니다.

```
반환 {  
  ...옵션,  
  외부: [], 출력: {  
    ...options.output,  
    libraryTarget: 'commonjs2',  
  },  
  // ... 나머지 구성  
};
```

이제 `$ nest build --webpack`을 사용하여 함수의 코드를 컴파일할 수 있습니다(그런 다음 `npx 서비스 오프라인으로 테스트합니다`).

또한 프로덕션 빌드를 축소할 때 클래스명을 그대로 유지하기 위해 `terser-webpack-plugin` 패키지를 설치하고 해당 구성은 재정의하는 것이 좋습니다(빌드 프로세스가 느려지므로 필수는 아님). 그렇게 하지 않으면 애플리케이션 내에서 `클래스 유효성 검사기`를 사용할 때 잘못된 동작이 발생할 수 있습니다.

```
const TerserPlugin = require('terser-webpack-plugin');

return {
  ...옵션,
  외부: [],
  최적화: {
    최소화합니다: [
      새로운
      TerserPlugin({
        terserOptions: {
          keep_classnames: true,
        },
      }),
    ],
  },
  출력합니다: {
    ...options.output,
    libraryTarget: 'commonjs2',
  }
}
```

```

},
// ... 나머지 구성
};

```

독립 실행형 애플리케이션 기능 사용

또는 함수를 매우 가볍게 유지하고 HTTP 관련 기능(라우팅뿐만 아니라 가드, 인터셉터, 파이프 등)이 필요하지 않은 경우, 다음과 같이 전체 HTTP 서버를 실행하는 대신 `NestFactory.createApplicationContext`(앞서 언급한 대로)를 사용할 수 있습니다(그리고 내부적으로 표현) :

@@파일명(메인)

```
'@nestjs/common'에서 { HttpStatus }를 임포트하고
, '@nestjs/core'에서 { NestFactory }를 임포트합니다;
'aws-lambda'에서 { Callback, Context, Handler }를 가져오고,
'./app.module'에서 { AppModule }을 가져옵니다;
'./app.service'에서 { AppService }를 가져옵니다;
```

`const` 핸들러를 내보냅니다: 핸들러 = 비동기 (이벤트:

```
any,
컨텍스트: 컨텍스트, 콜백:
콜백,
) => {
  const appContext = await
NestFactory.createApplicationContext(AppModule);
  const appService = appContext.get(AppService);
```

보조 힌트 {
 정보 힌트 `NestFactory.createApplicationContext`는 컨트롤러 메서드를 인핸서(가드, 인터셉터 등)
 body: appService.getHello(),
 로 래핑하지 않으라는 점에 유의하세요. 이를 위해서는 `NestFactory.create` 메서드를 사용해야
 };
 합니다.

또한 이벤트 객체를 처리하고 입력 값과 비즈니스 로직에 따라 해당 값을 반환할 수 있는 이벤트 서비스 공급자에게 이벤트 객체를 전달할 수도 있습니다.

```
const 핸들러를 내보냅니다: 핸들러 = 비동기 ( 이벤트:  
any,  
컨텍스트: 컨텍스트, 콜백:  
콜백,  
) => {  
  const appContext = await  
NestFactory.createApplicationContext(AppModule);  
  const eventsService = appContext.get(EventsService);
```

```
반환 이벤트서비스.프로세스(이벤트);  
};
```

HTTP 어댑터

Nest 애플리케이션 컨텍스트 내에서 또는 외부에서 기본 **HTTP** 서버에 액세스해야 하는 경우가 있습니다.

모든 네이티브(플랫폼별) **HTTP** 서버/라이브러리(예: Express 및 Fastify) 인스턴스는 어댑터로 래핑됩니다. 어댑터는 애플리케이션 컨텍스트에서 검색할 수 있고 다른 제공업체에 삽입할 수 있는 전역적으로 사용 가능한 제공업체로 등록됩니다.

외부 애플리케이션 컨텍스트 전략

애플리케이션 컨텍스트 외부에서 `HttpAdapter`에 대한 참조를 가져 오려면
메서드를 [호출합니다](#).

@@파일명()

```
const app = await NestFactory.create(AppModule);
const httpAdapter = app.getHttpAdapter();
```

상황에 맞는 전략

애플리케이션 컨텍스트 내에서 `HttpAdapterHost`에 대한 참조를 얻으려면 다른 기존 공급자와 동일한 기술(예: 생성자 주입 사용)을 사용하여 주입합니다.

@@파일명()

```
내보내기 클래스 CatsService {
  constructor(private adapterHost: HttpAdapterHost) {}
}

@@스위치 @디펜던시(HttpAdapterHost) 내보

내기 클래스 CatsService {
  constructor(adapterHost: HttpAdapterHost) {
    this.adapterHost = adapterHost;
  }
}
```

정보 힌트 `HttpAdapterHost`는 [@nestjs/core](#) 패키지에서 가져옵니다.

`HttpAdapterHost`는 실제 `HttpAdapter`가 아닙니다. 실제 `HttpAdapter` 인스턴스를 가져오려면
`httpAdapter` 프로퍼티에 액세스하면 됩니다.

```
const adapterHost = app.get(HttpAdapterHost);
const httpAdapter = adapterHost.httpAdapter;
```

`httpAdapter`는 기본 프레임워크에서 사용하는 HTTP 어댑터의 실제 인스턴스입니다. `ExpressAdapter` 또는 `FastifyAdapter`의 인스턴스입니다(두 클래스 모두 `AbstractHttpAdapter`를 확장합니다).

어댑터 객체는 **HTTP** 서버와 상호 작용할 수 있는 몇 가지 유용한 메서드를 노출합니다. 그러나 라이브러리 인스턴스(예: Express 인스턴스)에 직접 액세스하려면 `getInstance()` 메서드를 호출하세요.

```
const 인스턴스 = httpAdapter.getInstance();
```

글로벌 접두사

HTTP 애플리케이션에 등록된 모든 경로에 접두사를 설정하려면 `INestApplication` 인스턴스의 `setGlobalPrefix()` 메서드를 사용합니다.

```
const app = await NestFactory.create(AppModule);
app.setGlobalPrefix('v1');
```

다음 구문을 사용하여 글로벌 접두사에서 경로를 제외할 수 있습니다:

```
app.setGlobalPrefix('v1', {
  제외: [{ 경로: 'health', 메서드: RequestMethod.GET }],
});
```

또는 경로를 문자열로 지정할 수도 있습니다(모든 요청 메서드에 적용됨):

```
app.setGlobalPrefix('v1', { 제외: ['cats' ] });
```

정보 힌트 경로 속성은 경로 정규식 패키지를 사용하여 와일드카드 매개변수를 지원합니다. 참고:
와일드카드 별표 *를 사용할 수 없습니다. 대신 매개변수(예: `(.*)`, `:splat*`)를 사용해야 합니다.

생체

원시 요청 본문에 액세스하는 가장 일반적인 사용 사례 중 하나는 웹훅 서명 검증을 수행하는 것입니다. 일반적으로 웹훅 서명 유효성 검사를 수행하려면 직렬화되지 않은 요청 본문이 HMAC 해시를 계산하는 데 필요합니다.

경고 경고 이 기능은 내장된 글로벌 본문 구문 분석기 미들웨어가 활성화된 경우에만 사용할 수 있습니다.

즉, 앱을 만들 때 `bodyParser: false`를 전달하지 않아야 합니다.

Express와 함께 사용

먼저 Nest Express 애플리케이션을 만들 때 이 옵션을 활성화합니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/platform-express'에서 { NestExpressApplication } 유형 가져오기;
'./app.module'에서 { AppModule } 가져오기;

// "부트스트랩" 함수에서
const app = await NestFactory.create<NestExpressApplication>(AppModule, {
  rawBody: true,
});
await app.listen(3000);
```

컨트롤러에서 원시 요청 본문에 액세스하려면 요청에 원시 요청 필드를 노출하기 위한 편의 인터페이스 `RawBodyRequest`가 제공됩니다(인터페이스 `RawBodyRequest` 유형 사용):

```
'@nestjs/common'에서 { Controller, Post, RawBodyRequest, Req }를 임포트하고,
'express'에서 { Request }를 임포트합니다;

@Controller('cats') 클
래스 CatsController {
  @Post()
  create(@Req() req: RawBodyRequest<Request>) {
    const raw = req.rawBody; // `버퍼`를 반환합니다.
  }
}
```

다른 구문 분석기 등록하기

기본적으로 `json` 및 `urlencoded` 구문 분석기만 등록되어 있습니다. 다른 구문 분석기를 즉시 등록하려면 명시적으로 등록해야 합니다.

예를 들어 `テキスト` 구문 분석기를 등록하려면 다음 코드를 사용할 수 있습니다:

```
app.useBodyParser('text');
```

경고 `NestFactory.create` 호출에 올바른 애플리케이션 유형을 제공하고 있는지 확인합니다.

Express 애플리케이션의 경우 올바른 유형은 `NestExpressApplication`입니다. 그렇지 않으면

`.useBodyParser` 메서드를 찾을 수 없습니다.

본문 구문 분석기 크기 제한

애플리케이션에서 Express의 기본 100KB보다 큰 본문을 구문 분석해야 하는 경우 다음을 사용하세요:

```
app.useBodyParser('json', { limit: '10MB' });
```

`.useBodyParser` 메서드는 애플리케이션 옵션에서 전달되는 `rawBody` 옵션을 존중합니다. Fastify와 함께

사용

먼저 Nest Fastify 애플리케이션을 생성할 때 옵션을 활성화합니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다.  
FastifyAdapter,  
NestFastifyApplication,  
}를 '@nestjs/platform-fastify'에서 가져옵니다;  
'./app.module'에서 { AppModule }을 가져옵니다;  
  
// "부트스트랩" 함수에서  
const app = await NestFactory.create<NestFastifyApplication>(AppModule,  
    새로운 FastifyAdapter(),  
    {  
        rawBody: true,  
    },  
>;  
await app.listen(3000);
```

컨트롤러에서 원시 요청 본문에 액세스하려면 요청에 원시 요청 필드를 노출하기 위한 편의 인터페이스

`RawBodyRequest`가 제공됩니다(인터페이스 `RawBodyRequest` 유형 사용):

'@nestjs/common'에서 { Controller, Post, RawBodyRequest, Req }를 임포트하고,
'fastify'에서 { FastifyRequest }를 임포트합니다;

```
@Controller('cats') 클  
래스 CatsController {  
    @Post()  
    create(@Req() req: RawBodyRequest<FastifyRequest>) {  
        const raw = req.rawBody; // `버퍼`를 반환합니다.  
    }  
}
```

다른 구문 분석기 등록하기

기본적으로 `application/json` 및 `application/x-www-form-urlencoded` 구문 분석기만 등록되어 있습니다. 다른 구문 분석기를 즉시 등록하려면 명시적으로 등록해야 합니다.

예를 들어 `텍스트/일반` 구문 분석기를 등록하려면 다음 코드를 사용할 수 있습니다:

```
app.useBodyParser('text/plain');
```

경고 `NestFactory.create` 호출에 올바른 애플리케이션 유형을 제공하고 있는지 확인합니다. Fastify 애플리케이션의 경우 올바른 유형은 `NestFastifyApplication`입니다. 그렇지 않으면

`.useBodyParser` 메서드를 찾을 수 없습니다.

본문 구문 분석기 크기 제한

애플리케이션에서 Fastify의 기본 1MiB보다 큰 본문을 구문 분석해야 하는 경우 다음을 사용하세요:

```
const bodyLimit = 10_485_760; // 10MiB
app.useBodyParser('application/json', { bodyLimit });
```

`.useBodyParser` 메서드는 애플리케이션 옵션에서 전달된 `rawBody` 옵션을 존중합니다.

하이브리드 애플리케이션

하이브리드 애플리케이션은 HTTP 요청을 수신할 뿐만 아니라 연결된 마이크로서비스를 사용하는 애플리케이션입니다. `INestApplication` 인스턴스는 `connectMicroservice()` 메서드를 통해 `INestMicroservice` 인스턴스와 연결할 수 있습니다.

```
const app = await NestFactory.create(AppModule);
const microservice = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
});

await app.startAllMicroservices();
await app.listen(3001);
```

여러 마이크로서비스 인스턴스를 연결하려면 각 마이크로서비스에 대해 `connectMicroservice()`를 호출합니다:

```
const app = await NestFactory.create(AppModule);
// 마이크로서비스 #1
const microserviceTcp = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.TCP,
  옵션: { port:
    3001,
  },
});
// 마이크로서비스 #2
const microserviceRedis = app.connectMicroservice<MicroserviceOptions>({
  transport: Transport.REDIS,
  옵션: {
    호스트: 'localhost',
    포트: 6379,
  },
});
await app.startAllMicroservices();
await app.listen(3001);
```

여러 마이크로서비스가 있는 하이브리드 애플리케이션에서 `@MessagePattern()`을 하나의 전송 전략(예: `MQTT`)에만 바인딩하려면, 모든 기본 제공 전송 전략이 정의된 열거형인 `Transport` 유형의 두 번째 인수를 전달하면 됩니다.

```
@@파일명()  
메시지 패턴('time.us.*', Transport.NATS)  
getDate(@Payload() 데이터: 숫자[], @Ctx() context: NatsContext) {  
    console.log(`Subject: ${context.getSubject()}`); // 예: "time.us.east"  
    return new Date().toLocaleTimeString(...);  
}
```

```

메시지 패턴({ cmd: 'time.us' }, Transport.TCP)
getTCPDate(@Payload() 데이터: 숫자[]) {
    새 Date().toLocaleTimeString(...)을 반환합니다;
}
@@switch
@Bind(Payload(), Ctx())
@MessagePattern('time.us.*', Transport.NATS)
getDate(data, context) {
    console.log(`Subject: ${context.getSubject()}`); // 예: "time.us.east"
    return new Date().toLocaleTimeString(...);
}
@Bind(Payload(), Ctx())
메시지 패턴({ cmd: 'time.us' }, Transport.TCP) getDate(데이터, 컨텍스트) {
    새 Date().toLocaleTimeString(...)을 반환합니다;
}

```

정보 힌트 @Payload(), @Ctx(), Transport 및 NatsContext는 다음에서 가져옵니다.
@nestjs/microservices.

구성 공유

기본적으로 하이브리드 애플리케이션은 기본(HTTP 기반) 애플리케이션에 대해 구성된 전역 파이프, 인터셉터, 가드 및 필터를 상속하지 않습니다. 주 애플리케이션에서 이러한 구성 속성을 상속하려면 다음과 같이 connectMicroservice() 호출의 두 번째 인수(선택적 옵션 개체)에 inheritAppConfig 속성을 설정하세요:

```

const microservice = app.connectMicroservice<MicroserviceOptions>(
{
    전송: Transport.TCP,
},
{ inheritAppConfig: true },
);

```

HTTPS

HTTPS 프로토콜을 사용하는 애플리케이션을 만들려면 `NestFactory` 클래스의 `create()` 메서드에 전달된 옵션 객체에서 `httpsOptions` 속성을 설정합니다:

```
const httpsOptions = {
  키: fs.readFileSync('./secrets/private-key.pem'),
  인증: fs.readFileSync('./secrets/public-certificate.pem'),
};

const app = await NestFactory.create(AppModule, {
  httpsOptions,
});

await app.listen(3000);
```

`FastifyAdapter`를 사용하는 경우 다음과 같이 애플리케이션을 생성합니다:

```
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  새로운 FastifyAdapter({ https: httpsOptions }),
);
```

여러 대의 동시 서버

다음 레시피는 여러 포트(예: 비-HTTPS 포트와 HTTPS 포트)에서 동시에 수신 대기하는 Nest 애플리케이션을 인스턴스화하는 방법을 보여줍니다.

```
const httpsOptions = {
  키: fs.readFileSync('./secrets/private-key.pem'),
  인증: fs.readFileSync('./secrets/public-certificate.pem'),
};

const server = express();
const app = await NestFactory.create(
  AppModule,
  새로운 ExpressAdapter(서버),
);
await app.init();

const httpServer = http.createServer(server).listen(3000);
const httpsServer = https.createServer(httpsOptions, server).listen(443);
```

`http.createServer` / `https.createServer`를 직접 호출했기 때문에 NestJS는 `app.close` / `on` 종료 신호를

호출할 때 이를 닫지 않습니다. 이 작업을 직접 수행해야 합니다:



```
@Injectable()
내보내기 클래스 ShutdownObserver는 OnApplicationShutdown을 구현합니다 {
  private httpServers: http.Server[] = [];

  public addHttpServer(server: http.Server): void {
    this.httpServers.push(server);
  }

  공용 비동기 onApplicationShutdown(): Promise<void> { await
    Promise.all(
      this.httpServers.map((server) => new
        Promise((resolve, reject) => {
          server.close((error) => {
            if (error) {
              reject(error);
            } else {
              resolve(null);
            }
          });
        })
      ),
    );
  }
}

const shutdownObserver = app.get(ShutdownObserver);
shutdownObserver.addHttpServer(httpServer);
shutdownObserver.addHttpServer(httpsServer);
```

정보 힌트 ExpressAdapter는 [@nestjs/platform-express](#) 패키지에서 가져옵니다. 이 패키지의 [http](#) 및 [https](#) 패키지는 네이티브 Node.js 패키지입니다.

경고 이 레시피는 [GraphQL](#) 구독에서는 작동하지 않습니다.

요청 수명 주기

Nest 애플리케이션은 요청 수명 주기라고 하는 시퀀스에 따라 요청을 처리하고 응답을 생성합니다. 미들웨어, 파이프, 가드, 인터셉터를 사용하면 특히 글로벌, 컨트롤러 수준, 경로 수준 구성 요소가 작용하기 때문에 요청 수명 주기 동안 특정 코드가 실행되는 위치를 추적하기가 어려울 수 있습니다. 일반적으로 요청은 미들웨어를 통해 가드, 인터셉터, 파이프로 이동한 다음 응답이 생성될 때 반환 경로에서 다시 인터셉터로 이동합니다.

미들웨어

미들웨어는 특정 순서로 실행됩니다. 먼저 Nest는 전역으로 바인딩된 미들웨어(예: `app.use`로 바인딩된 미들웨어)를 실행한 다음 경로에 따라 결정되는 **모듈 바인딩된** 미들웨어를 실행합니다. 미들웨어는 바인딩된 순서대로 순차적으로 실행되며, 이는 Express의 미들웨어가 작동하는 방식과 유사합니다. 여러 모듈에 바인딩된 미들웨어의 경우 루트 모듈에 바인딩된 미들웨어가 먼저 실행되고, 그 다음 모듈이 `import` 배열에 추가되는 순서대로 미들웨어가 실행됩니다.

경비병

가드 실행은 글로벌 가드부터 시작하여 컨트롤러 가드, 라우팅 가드 순으로 진행됩니다. 미들웨어와 마찬가지로 가드는 바인딩된 순서대로 실행됩니다. 예를 들어

```
UseGuards(Guard1, Guard2)
@Controller('cats')
내보내기 클래스 CatsController {
  constructor(private catsService: CatsService) {}

  사용가드(Guard3)
  @Get()
  getCats(): Cats[] {
    this.catsService.getCats()를 반환합니다;
  }
}
```

가드1은 가드2보다 먼저 실행되며, 둘 다 가드3보다 먼저 실행됩니다.

정보 힌트 전역 바인딩과 컨트롤러 또는 로컬 바인딩에 대해 말할 때 가드(또는 다른 컴포넌트)가 바인딩되는 위치에 차이가 있습니다. `app.useGlobalGuard()`를 사용하거나 모듈을 통해 컴포넌트를 제공하는 경우 전역적으로 바인딩됩니다. 그렇지 않으면 데코레이터가 컨트롤러 클래스 앞에 오는 경우 컨트롤러에 바인딩되고, 데코레이터가 경로 선언을 진행하는 경우 경로에 바인딩됩니다.

인터셉터

인터셉터는 대부분 가드와 동일한 패턴을 따르지만, 한 가지 예외가 있습니다. 인터셉터가 [RxJS Observables](#)를 반환할 때, 관찰 가능 항목은 선입선출 방식으로 해결됩니다. 따라서 인바운드 요청은 표준 글로벌, 컨트롤러, 라우트 수준 확인을 거치지만, 요청의 응답 측(즉, 컨트롤러 메서드 핸들러에서 반환된 후)은 라우트에서 컨트롤러로, 글로벌로 확인됩니다. 또한,

파이프, 컨트롤러 또는 서비스에서 발생하는 모든 오류는 인터셉터의 `catchError` 연산자에서 읽을 수 있습니다

파이프

파이프는 표준 글로벌-컨트롤러-경로 바인딩 순서를 따르며, `@UsePipes()` 매개변수와 관련하여 동일한 선입선출 방식을 따릅니다. 그러나 경로 매개변수 수준에서는 여러 개의 파이프가 실행 중인 경우 마지막 매개변수에서 첫 번째 매개변수까지 순서대로 실행됩니다. 이는 경로 수준 및 컨트롤러 수준 파이프에도 적용됩니다. 예를 들어 다음과 같은 컨트롤러가 있다고 가정해 보겠습니다:

```
사용파이프(일반검증파이프) @컨트롤러('cats')
내보내기 클래스 CatsController {
    constructor(private catsService: CatsService) {}

    사용파이프(경로특정파이프) @패치('/:id')
    업데이트캣(
        Body() 본문: UpdateCatDTO, @Param()
        매개변수: UpdateCatParams, @Query()
        쿼리: UpdateCatQuery,
    ) {
        이.catsService.updateCat(body, params, query)을 반환합니다;
    }
}
```

로 설정하면 `쿼리`, `매개변수`, `본문` 객체에 대해 `GeneralValidationPipe`가 실행된 후 동일한 순서를 따르는 `RouteSpecificPipe`로 이동합니다. 매개변수별 파이프가 있는 경우 컨트롤러 및 경로 수준 파이프 다음에 매개변수별 파이프가 실행됩니다(다시 말해서 마지막 매개변수부터 첫 번째 매개변수까지).

필터

필터는 전역 먼저 해결하지 않는 유일한 구성 요소입니다. 대신 필터는 가능한 가장 낮은 수준부터 해결하므로 실행은 경로 바인딩 필터에서 시작하여 컨트롤러 수준, 그리고 마지막으로 전역 필터로 진행됩니다. 예외는 필터 간에 전달될 수 없으며, 경로 수준 필터가 예외를 포착하면 컨트롤러 또는 전역 수준 필터는 동일한 예외를 포착할 수 없습니다. 이와 같은 효과를 얻을 수 있는 유일한 방법은 필터 간에 상속을 사용하는 것입니다.

정보 힌트 필터는 요청 프로세스 중에 잡히지 않은 예외가 발생하는 경우에만 실행됩니다. [시도/잡기로](#) 잡힌 예외와 같이 잡힌 예외는 예외 필터가 실행되도록 트리거되지 않습니다. 잡히지 않은 예외가 발생하면 나머지 라이프사이클은 무시되고 요청은 바로 필터로 건너뛰게 됩니다.

요약

일반적으로 요청 라이프사이클은 다음과 같습니다:

- . 수신 요청

. 미들웨어

- 2.1. 글로벌 바인딩 미들웨어
- 2.2. 모듈 바인딩 미들웨어

. Guards

- 3.1 글로벌 가드
- 3.2 컨트롤러 가드
- 3.3 경로 가드

. 인터셉터(사전 컨트롤러)

- 4.1 글로벌 인터셉터
- 4.2 컨트롤러 인터셉터
- 4.3 경로 인터셉터

. 파이

- 프 5.1 글로벌 파이프
- 5.2 컨트롤러 파이프
 - 5.3 파이프 라우팅
 - 5.4 매개변수 파이프 라우팅

. 컨트롤러(메서드 핸들러)

. 서비스(있는 경우)

. 인터셉터(요청 후)

- 8.1 경로 인터셉터
- 8.2 컨트롤러 인터셉터
- 8.3 글로벌 인터셉터

. 예외 필터

- 9.1 경로
- 9.2 컨트롤러
- 9.3 글로벌

. 서버 응답

일반적인 오류

NestJS로 개발하는 동안 프레임워크를 익히면서 다양한 오류가 발생할 수 있습니다. "종속성을 해결할 수 없음"

오류

정보 힌트 "종속성을 해결할 수 없음" 오류를 쉽게 해결하는 데 도움이 되는 [NestJS 개발자 도구를 확인하세요.](#)

아마도 가장 일반적인 오류 메시지는 Nest가 공급자의 종속성을 해결할 수 없다는 것입니다. 오류 메시지 는 일반적으로 다음과 같이 표시됩니다:

Nest가 <제공자>(?)의 종속성을 해결할 수 없습니다. 인덱스 [<인덱스>]의 인수 <unknown_token>을 사용할 수 있는지 확인하세요.
<모듈> 컨텍스트.

잠재적인 솔루션:

- <모듈>이 유효한 NestJS 모듈인가요?
- <unknown_token>이 공급자라면 현재 <모듈>의 일부인가요?
- <unknown_token>이 별도의 @Module에서 내보낸 경우 해당 모듈을 <module> 내에서 가져올 수 있나요?

모듈({

 임포트합니다: [/* <unknown_token>이 포함된 모듈 */]입니다.

})

오류의 가장 일반적인 원인은 모듈의 공급자 배열에 <공급자>가 없는 경우입니다. 공급자가 실제로 공급자 배열에 있고 [표준 NestJS 공급자 관행을](#) 따르고 있는지 확인하세요.

몇 가지 일반적인 문제가 있습니다. 하나는 import 배열에 프로바이더를 넣는 경우입니다. 이 경우 오류에 <module>이 있어야 할 위치에 공급자 이름이 표시됩니다.

개발 중에 이 오류가 발생하면 오류 메시지에 언급된 모듈을 살펴보고 해당 모듈의 공급자를 살펴보세요. 공급자 배열의 각 공급자에 대해 모듈이 모든 종속성에 액세스할 수 있는지 확인하세요. 종종 프로바이더가 "기능 모듈"과 "루트 모듈"에 중복되는 경우가 있는데, 이는 Nest가 프로바이더를 두 번 인스턴스화하려고 시도한다는 의미입니다. 대부분의 경우 <공급자>가 복제되는 대신 "루트 모듈"의 가져오기 배열에 추가되어야 합니다.

위의 `<unknown_token>` 문자열 종속성인 경우 순환 파일 가져오기가 있을 수 있습니다. 이는 생성자에서 프로바이더가 서로 종속되는 대신 두 파일이 서로를 임포트하는 것을 의미하기 때문에 아래의 [순환 종속성](#)과는 다릅니다. 일반적인 경우는 모듈 파일에서 토큰을 선언하고 프로바이더를 임포트하고, 프로바이더는 모듈 파일에서 토큰 상수를 임포트하는 경우입니다. 배럴 파일을 사용하는 경우, 배럴 임포트가 이러한 순환 임포트를 생성하지 않도록 하세요.

위의 `<unknown_token>` 문자열 `Object`인 경우 적절한 공급자 토큰이 없는 유형/인터페이스를 사용하여 주입하고 있다는 의미입니다. 이 문제를 해결하려면 클래스 참조를 가져오고 있는지 확인하거나 `@Inject()` 데코레이터와 함께 사용자 지정 토큰을 사용하세요. [사용자 정의 공급자](#) 페이지를 읽어보세요.

또한 NestJS에서는 자체 주입이 허용되지 않으므로 공급자를 자체적으로 주입하지 않았는지 확인해야 합니다.

이 경우 `<unknown_token>`은 `<provider>`와 같을 가능성이 높습니다.

모노레포 설정인 경우 위와 동일한 오류가 발생할 수 있지만 코어 공급자의 경우

`ModuleRef`를 `<알 수 없는 토큰>`으로 저장합니다:

Nest가 `<제공자>(?)`의 종속성을 해결할 수 없습니다.

인덱스 [`<인덱스>`]의 `ModuleRef` 인수가 `<모듈>` 컨텍스트에서 사용 가능한지 확인하세요.

...

프로젝트에서 다음과 같이 `@nestjs/core` 패키지의 두 노드 모듈을 로드할 때 이런 문제가 발생할 수 있습니다

```
.
├── package.json
└── 앱
    └── API
        └── node_modules
            └── @nestjs/bull
                └── node_modules
                    └── @nestjs/core
└── node_modules
    ├── (기타 패키지)
    └── @nestjs/core
```

솔루션:

- Yarn 워크스페이스의 경우, `nohoist` 기능을 사용하여 `@nestjs/core` 패키지를 들어 올리지 않도록 하세요.
- pnpm 워크스페이스의 경우, 다른 모듈의 피어 종속성 및 "의존성 메타"에 `@nestjs/core`를 설정 합니다: `{} ' {}' {}"다른 모듈 이름": {} ' {}' {}"injected": true{}' {}}를 모듈` 을 임포트한 앱 `package.json`에 추가합니다. 참조: [의존성 메타인젝트](#)

"순환 종속성" 오류

애플리케이션에서 [순환 종속성을](#) 피하기 어려운 경우가 있습니다. Nest가 이러한 문제를 해결할 수 있도록 몇 가지 단계를 수행해야 합니다. 순환 종속성으로 인해 발생하는 오류는 다음과 같습니다:

Nest가 <모듈> 인스턴스를 만들 수 없습니다.

<모듈> "가져오기" 배열의 인덱스 [<index>]에 있는 모듈이 정의되지 않았습니다.

잠재적 원인:

- 모듈 간의 순환 종속성. 이를 방지하려면 `forwardRef()`를 사용하세요. 자세히 보기:

<https://docs.nestjs.com/fundamentals/circular-dependency>

- 인덱스 [<index>]의 모듈이 "정의되지 않음" 유형입니다. 가져오기 확인

문과 모듈의 유형에 따라 다릅니다.

범위 [`<모듈_임포트_체인>`]]

```
# 예제 체인 AppModule -> FooModule
```

모듈 파일에서 상수를 내보내고 서비스 파일에서 가져오는 것과 같이 두 공급자가 서로 의존하거나 상수에 대해 서로 의존하는 타입스크립트 파일에서 순환 종속성이 발생할 수 있습니다. 후자의 경우 상수를 위한 별도의 파일을 생성하는 것이 좋습니다. 전자의 경우 순환 종속성에 대한 가이드를 따르고 모듈과 공급자가 모두 `forwardRef`로 표시되어 있는지 확인하세요.

종속성 오류 디버깅

종속성이 올바른지 수동으로 확인하는 것 외에도 Nest 8.1.0부터는 `NEST_DEBUG` 환경 변수를 진실로 해석하는 문자열로 설정하여 Nest가 애플리케이션의 모든 종속성을 해석하는 동안 추가 로깅 정보를 얻을 수 있습니다.



위 이미지에서 노란색 문자열은 주입되는 종속성의 호스트 클래스, 파란색 문자열은 주입된 종속성의 이름 또는 주입 토큰, 보라색 문자열은 종속성을 검색 중인 모듈입니다. 이 문자열을 사용하면 일반적으로 종속성 주입 문제가 발생하는 이유와 종속성 해결 과정을 추적할 수 있습니다.

"파일 변경 감지됨"이 끝없이 반복됨

TypeScript 버전 4.9 이상을 사용하는 Windows 사용자는 이 문제가 발생할 수 있습니다. 이 문제는 감시 모드에서 애플리케이션을 실행하려고 할 때 발생합니다(예: `npm 실행 start:dev`에서 로그 메시지가 끝없이 반복되는 것을 볼 수 있습니다):

```
XX:XX:XX AM - 파일 변경이 감지되었습니다. 증분 컴파일 시작 중... XX:XX:XX AM - 0 오류 발견. 파일 변경을 감시 중입니다.
```

NestJS CLI를 사용하여 감시 모드에서 애플리케이션을 시작할 때 `tsc -- watch`을 호출하여 수행되며, TypeScript 버전 4.9부터 파일 변경을 감지하는 새로운 전략이 사용되어 이 문제의 원인이 될 수 있습니다. 이 문제를 해결하려면 다음과 같이 `"compilerOptions"` 옵션 뒤에 `tsconfig.json` 파일에 설정을 추가해야 합니다:

```
"watchOptions": {  
    "watchFile": "고정 폴링 간격"  
}
```

이 옵션은 일부 컴퓨터에서 문제를 일으킬 수 있는 파일 시스템 이벤트(새로운 기본 방법) 대신 폴링 방법을 사용하여 파일 변경 사항을 확인하도록 TypeScript에 지시합니다. ["watchFile"](#) 옵션에 대한 자세한 내용은 [TypeScript 문서에서](#) 확인할 수 있습니다.

개요

정보 힌트 이 장에서는 Nest 프레임워크와의 Nest Devtools 통합에 대해 설명합니다. 개발자 도구 애플리케이션을 찾고 계신다면 [개발자](#) 도구 웹사이트를 방문하세요.

로컬 애플리케이션 디버깅을 시작하려면 다음과 같이 `main.ts` 파일을 열고 애플리케이션 옵션 객체에서 `스냅샷` 속성을 `true`로 설정하세요:

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule, { 스냅샷: true });
  await app.listen(3000);
}
```

이렇게 하면 프레임워크가 네스트 개발자 도구가 애플리케이션의 그래프를 시각화할 수 있도록 필요한 메타데이터를 수집하도록 지시합니다. 다음으로 필요한 의존성을 설치하겠습니다:

```
$ npm i @nestjs/devtools-integration
```

경고 경고 애플리케이션에서 `@nestjs/graphql` 패키지를 사용하는 경우 최신 버전(`npm i @nestjs/graphql@11`)을 설치해야 합니다.

이 종속성을 설정했으면 `app.module.ts` 파일을 열고 방금 설치한 `DevtoolsModule`을 임포트해 보겠습니다:

```
모듈({ import:
  [
    DevtoolsModule.register({
      http: process.env.NODE_ENV !== 'production',
    }),
    컨트롤러: [AppController], 공급
    자: [앱서비스],
  }
)
내보내기 클래스 AppModule {}
```

경고 경고 여기서 `NODE_ENV` 환경 변수를 확인하는 이유는 프로덕션 환경에서 이 모듈을 절대 사용해서는 안 되기 때문입니다!

개발자 도구 `모듈`을 가져오고 애플리케이션이 실행되면(`npm 실행 시작: dev`) [개발자](#) 도구 URL로 이동하여 검사된 그래프를 볼 수 있어야 합니다.



정보 힌트 위의 스크린샷에서 볼 수 있듯이 모든 모듈은 `InternalCoreModule`에 연결됩니다. `InternalCoreModule`은 항상 루트 모듈로 임포트되는 글로벌 모듈입니다. 글로벌 노드로 등록되어 있기 때문에 Nest는 모든 모듈과 `InternalCoreModule` 노드 사이에 자동으로 에지를 생성합니다. 이제 그래프에서 전역 모듈을 숨기려면 사이드바의 "전역 모듈 숨기기" 체크박스를 사용하면 됩니다.

위에서 볼 수 있듯이 `DevtoolsModule`을 사용하면 애플리케이션이 포트 8000에 있는 추가 HTTP 서버를 노출하여 Devtools 애플리케이션이 앱을 인트로스펙티브하는 데 사용할 수 있습니다.

모든 것이 예상대로 작동하는지 다시 확인하려면 그래프 보기 를 "클래스"로 변경합니다. 다음 화면이 표시됩니다:



특정 노드에 초점을 맞추려면 사각형을 클릭하면 그래프에 '포커스' 버튼이 있는 팝업 창이 표시됩니다. 사이드바에 있는 검색창을 사용하여 특정 노드를 찾을 수도 있습니다.

정보 힌트 검사 버튼을 클릭하면 애플리케이션이 특정 노드가 선택된 `/debug` 페이지로 이동합니다.



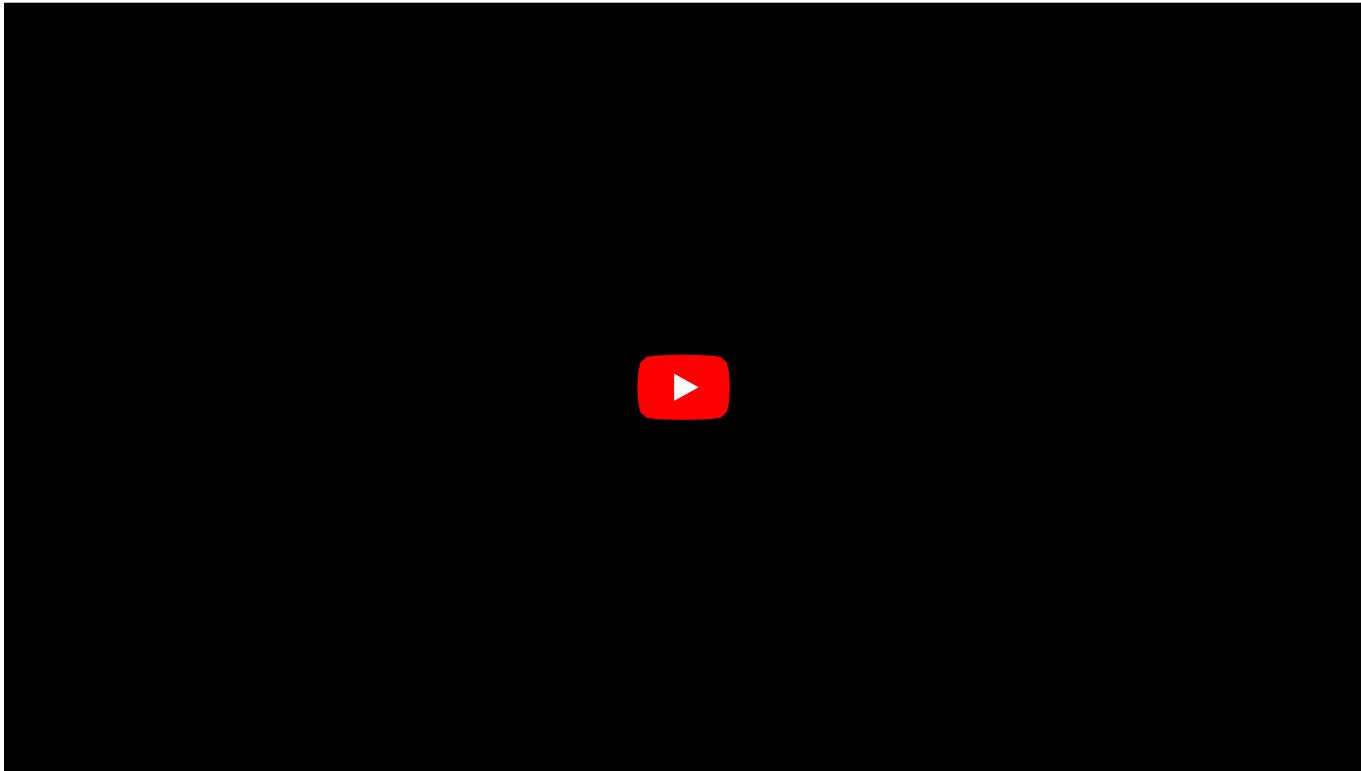
정보 힌트 그래프를 이미지로 내보내려면 그래프 오른쪽 모서리에 있는 PNG로 내보내기 버튼을 클릭합니다.

사이드바(왼쪽)에 있는 양식 컨트롤을 사용하여 예를 들어 특정 애플리케이션 하위 트리를 시각화하기 위해 가장자리 근접성을 제어할 수 있습니다:



이 기능은 팀에 새로운 개발자가 들어왔을 때 애플리케이션이 어떻게 구성되어 있는지 보여주고 싶을 때 특히 유용할 수 있습니다. 또한 이 기능을 사용하여 특정 모듈(예: [TasksModule](#))과 모든 종속성을 시각화할 수 있으므로 대규모 애플리케이션을 더 작은 모듈(예: 개별 마이크로 서비스)로 세분화할 때 유용하게 사용할 수 있습니다.

이 동영상을 통해 그래프 탐색기 기능이 실제로 작동하는 모습을 확인할 수 있습니다:



"종속성을 해결할 수 없음" 오류 조사 중

정보 참고 이 기능은 `@nestjs/core >= v9.3.10`에서 지원됩니다.

아마도 가장 흔히 볼 수 있는 오류 메시지는 Nest가 공급자의 종속성을 해결할 수 없다는 것입니다. Nest 개발자 도구를 사용하면 문제를 쉽게 식별하고 해결 방법을 배울 수 있습니다.

먼저 `main.ts` 파일을 열고 다음과 같이 [부트스트랩\(\)](#) 호출을 업데이트합니다:

```
bootstrap().catch((err) => {
  fs.writeFileSync('graph.json', PartialGraphHost.toString() ?? '');
  process.exit(1);
});
```

또한 `abortOnErrorHandler`를 `false`로 설정해야 합니다:

```
const app = await NestFactory.create(AppModule, { 스냅샷:
  true,
  abortOnErrorHandler: false, // <--- THIS
});
```

이제 애플리케이션이 "종속성을 해결할 수 없음" 오류로 인해 부트스트랩에 실패할 때마다 루트 디렉터리에서 그래프(부분 그래프를 나타내는) 파일인 `graph.json`을 찾을 수 있습니다. 그런 다음 이 파일을 개발자 도구로 끌어다 놓으면 됩니다(현재 모드를 "대화형"에서 "미리보기"로 전환해야 함):



업로드에 성공하면 다음과 같은 그래프와 대화창이 표시됩니다:



보시다시피 강조 표시된 `작업` 모듈이 우리가 살펴봐야 할 모듈입니다. 또한 대화 창에서 이 문제를 해결하는 방법에 대한 몇 가지 지침을 이미 볼 수 있습니다.

대신 '클래스' 보기로 전환하면 다음과 같은 내용이 표시됩니다:



이 그래프는 `태스크` 서비스에 주입하려는 `진단` 서비스를 `태스크 모듈의` 컨텍스트에서 찾을 수 없음을 보여줍니다.

도풀에진단 모듈을 가져와서 이 문제를 해결해야 할 것입니다! 경로 탐색기

경로 탐색기 페이지로 이동하면 등록된 모든 진입점을 볼 수 있습니다:



정보 힌트 이 페이지에는 HTTP 경로뿐만 아니라 다른 모든 진입점(예: 웹소켓, gRPC, GraphQL 리졸버 등)도 표시됩니다.

엔트리포인트는 호스트 컨트롤러에 따라 그룹화됩니다. 검색창을 사용하여 특정 엔트리포인트를 찾을 수도 있습니다.

특정 진입 지점을 클릭하면 흐름 그래프가 표시됩니다. 이 그래프는 진입점(예: 이 경로에 연결된 가드, 인터셉터, 파이프 등)의 실행 흐름을 보여줍니다. 이 그래프는 특정 경로에 대한 요청/응답 주기를 이해하거나 특정 가드/인터셉터/파이프가 실행되지 않는 이유를 해결할 때 특히 유용합니다.

샌드박스

자바스크립트 코드를 즉석에서 실행하고 애플리케이션과 실시간으로 상호작용하려면 샌드박스 페이지로 이동하세요:



플레이그라운드를 사용하면 API 엔드포인트를 실시간으로 테스트하고 디버깅할 수 있으므로 개발자는 HTTP 클라이언트 등을 사용하지 않고도 문제를 빠르게 식별하고 해결할 수 있습니다. 또한 인증 계층을 우회할 수 있으므로 더 이상 추가 로그인 단계나 테스트 목적의 특수 사용자 계정이 필요하지 않습니다. 이벤트 기반 애플리케이션의 경우, 플레이그라운드에서 직접 이벤트를 트리거하고 애플리케이션이 이에 어떻게 반응하는지 확인할 수도 있습니다.

로그아웃되는 모든 항목은 플레이그라운드의 콘솔로 간소화되어 무슨 일이 일어나고 있는지 쉽게 확인할 수 있습니다.

애플리케이션을 다시 빌드하고 서버를 다시 시작할 필요 없이 코드를 즉시 실행하고 결과를 바로 확인할 수 있습니다.



정보 힌트 객체 배열을 예쁘게 표시하려면 `console.table()`(또는 그냥 `table()`) 함수를 사용하세요.

이 동영상은 동일한 내용입니다. 프레임워크마다 차이는 있지만 기본적인 목적으로 차이가 없습니다.



부트스트랩 성능 분석기

모든 클래스 노드(컨트롤러, 공급자, 인핸서 등) 목록과 해당 인스턴스화 시간을 확인하려면 부트스트랩 성능 페이지로 이동하세요:



이 페이지는 애플리케이션의 부트스트랩 프로세스에서 가장 느린 부분을 식별하려는 경우(예: 서비스 환경에서 중요한 애플리케이션의 시작 시간을 최적화하려는 경우)에 특히 유용합니다.

감사

자동 생성된 감사(애플리케이션이 직렬화된 그래프를 분석하는 동안 발생한 오류/경고/힌트)를 확인하려면 감사 페이지로 이동합니다:



정보 힌트 위의 스크린샷에는 사용 가능한 모든 감사 규칙이 표시되어 있지 않습니다.

이 페이지는 애플리케이션의 잠재적인 문제를 파악할 때 유용합니다.

정적 파일 미리보기

직렬화된 그래프를 파일에 저장하려면 다음 코드를 사용합니다:

```
await app.listen(3000); // OR await app.init()  
fs.writeFileSync('./graph.json', app.get(SerializedGraph).toString());
```

정보 힌트 SerializedGraph는 [@nestjs/core](#) 패키지에서 내보냅니다.

그런 다음 이 파일을 끌어다 놓거나 업로드할 수 있습니다:



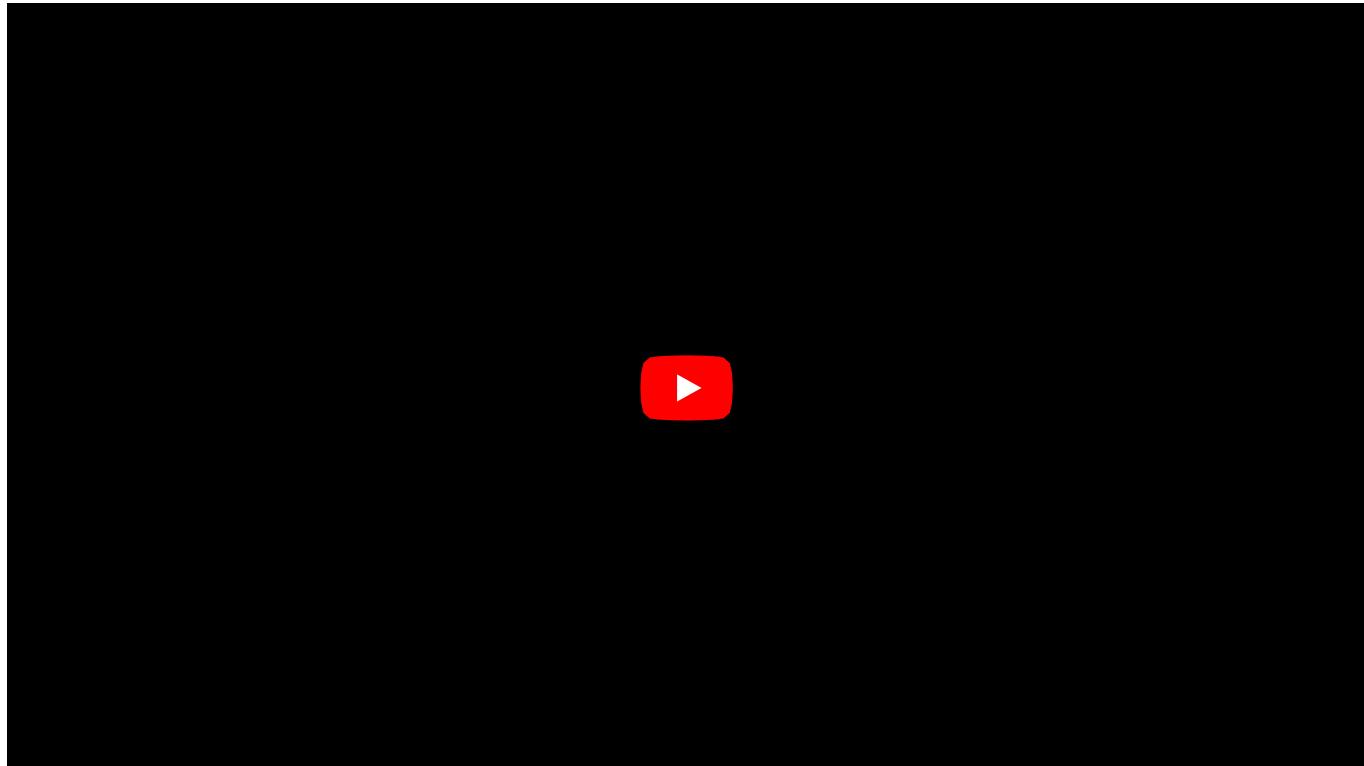
이 기능은 다른 사람(예: 동료)과 그래프를 공유하거나 오프라인에서 분석하고 싶을 때 유용합니다.

CI/CD 통합

정보 힌트 이 장에서는 Nest 프레임워크와의 Nest Devtools 통합에 대해 설명합니다. 개발자 도구 애플리케이션을 찾고 계신다면 [개발자](#) 도구 웹사이트를 방문하세요.

CI/CD 통합은 [Enterprise](#) 요금제를 사용하는 사용자에게 제공됩니다.

이 동영상을 통해 CI/CD 통합이 도움이 되는 이유와 방법을 알아보세요:



그래프 게시

먼저 애플리케이션 부트스트랩 파일([main.ts](#))을 다음과 같이 [GraphPublisher](#) 클래스(@nestjs/devtools-integration에서 내보낸 - 자세한 내용은 이전 장 참조)를 사용하도록 구성해 보겠습니다:

```
비동기 함수 부트스트랩() {
  const shouldPublishGraph = process.env.PUBLISH_GRAPH === "true";

  const app = await NestFactory.create(AppModule, { 스냅
    샷: true,
    미리보기: shouldPublishGraph,
  });

  if (shouldPublishGraph) {
    await app.init();

    const publishOptions = { ... } // 참고: 이 옵션 객체는 사용 중인 CI/CD 제공업체에 따라 달라집니다.
    const graphPublisher = new GraphPublisher(app);
    await graphPublisher.publish(publishOptions);

    await app.close();
  } else {
    await app.listen(3000);
  }
}
```

보시다시피, 여기서는 [그레프](#) 퍼블리셔를 사용하여 직렬화된 그래프를 중앙 집중식 레지스트리에 게시하고 있습니다. [PUBLISH_GRAPH](#)는 그래프를 게시할지(CI/CD 워크 플로), 아니면 일반 애플리케이션 부트스트랩에 게시할지 여부를 제어할 수 있는 사용자 정의 환경 변수입니다. 또한 여기서 [미리보기](#) 속성을 [true](#)로 설정했습니다. 이 플래

그를 활성화하면 애플리케이션이 미리보기 모드에서 부트스트랩되며, 이는 기본적으로 애플리케이션의 모든 컨트롤러, 인핸서 및 프로바이더의 생성자(및 수명 주기 후크)가 실행되지 않음을 의미합니다. 참고 - 필수는 아니지만 이 경우 CI/CD 파이프라인에서 애플리케이션을 실행할 때 데이터베이스 등에 연결할 필요가 없으므로 작업이 더 간단해집니다.

제시 옵션 개체는 사용 중인 CI/CD 제공업체에 따라 달라집니다. 가장 많이 사용되는 CI/CD 제공업체에 대한 지침은 이후 섹션에서 자세히 설명합니다.

그래프가 성공적으로 게시되면 워크플로 보기에 다음과 같은 출력이 표시됩니다:



그래프가 게시될 때마다 프로젝트의 해당 페이지에 새 항목이 표시됩니다:



보고서

중앙 집중식 레지스트리에 이미 해당 스냅샷이 저장되어 있는 경우 Devtools는 모든 빌드에 대해 보고서를 생성합니다. 예를 들어 그래프가 이미 게시된 [마스터](#) 브랜치에 대해 PR을 만들면 애플리케이션에서 차이점을 감지하고 보고서를 생성할 수 있습니다. 그렇지 않으면 보고서가 생성되지 않습니다.

보고서를 보려면 프로젝트의 해당 페이지로 이동합니다(조직 참조).



이 기능은 코드 검토 중에 눈에 띄지 않았을 수 있는 변경 사항을 식별하는 데 특히 유용합니다. 예를 들어, 누군가 깊게 중첩된 프로바이더의 범위를 변경했다고 가정해 보겠습니다. 이러한 변경 사항은 검토자가 즉시 알아차리지 못할 수도 있지만 Devtools를 사용하면 이러한 변경 사항을 쉽게 발견하고 의도적인 변경인지 확인할 수 있습니다. 또는 특정 엔드포인트에서 가드를 제거하면 보고서에서 해당 엔드포인트가 영향을 받는 것으로 표시됩니다. 해당 경로에 대한 통합 또는 e2e 테스트가 없다면 더 이상 보호되지 않는다는 사실을 알아차리지 못할 수 있으며, 알아차렸을 때는 이미 너무 늦을 수 있습니다.

마찬가지로 대규모 코드베이스에서 작업할 때 모듈을 전역으로 수정하면 그래프에 추가된 에지 수를 확인할 수 있으며, 이는 대부분의 경우 원기 잘못하고 있다는 신호입니다.

빌드 미리 보기

게시된 모든 그래프에 대해 시간을 거슬러 올라가서 미리보기 버튼을 클릭하여 이전 그래프의 모습을 미리 볼 수 있습니다. 또한 보고서가 생성된 경우 그래프에서 차이점이 강조 표시된 것을 볼 수 있습니다:

- 녹색 노드는 추가된 요소를 나타냅니다.
- 밝은 흰색 노드는 업데이트된 요소를 나타냅니다
- 빨간색 노드는 삭제된 요소를 나타냅니다.

아래 스크린샷을 참조하세요:



시간을 거슬러 올라가는 기능을 사용하면 현재 그래프와 이전 그래프를 비교하여 문제를 조사하고 해결할 수 있습니다. 설정 방법에 따라 모든 플리퀘스트(또는 모든 커밋)에 해당하는 스냅샷이 레지스트리에 저장되므로 시간을 거슬러 올라가서 변경된 내용을 쉽게 확인할 수 있습니다. Nest가 애플리케이션 그래프를 구성하는 방식을 이해하고 이를 시각화할 수 있는 기능을 갖춘 Devtools를 Git이라고 생각하세요.

통합: Github 액션

먼저 프로젝트의 [.github/workflows](#) 디렉터리에 새 Github 워크플로우를 생성하고, 예를 들어 [publish-graph.yml](#)이라고 부르겠습니다. 이 파일 안에 다음 정의를 사용하겠습니다:

이름: 개발자 도구

켜짐:

푸시합니다:

브랜치:

- 마스터 풀

_요청:

브랜치:

- '*'

작업:

게시합니다:

```
if: github.actor != 'dependabot[bot]'  
name: 그래프 게시
```

실행 중: 우분투 최신 단계:

- 용도: 액션/체크아웃@v3
- 사용: 동작/설정 노드@v3와 함께:

노드-버전: '16' 캐시:

'npm'

- 이름: 설치 종속성 실행: npm ci

- 이름: 설정 환경(PR)

```
if: {{ '${{' }} github.event_name == 'pull_request' {{ '}}' }} 셸:  
bash  
실행합니다: |  
echo "COMMIT_SHA={{ '${{' }} github.event.pull_request.head.sha {{ '}}' }}" >> ${github_env}  
- 이름: 설정 환경(푸시)
```

```

if: {{ '${{' }} github.event_name == 'push' {{ '}}' }}
셸: bash
실행합니다:
echo "COMMIT_SHA=\${GITHUB_SHA}" >> \${GITHUB_ENV}
- 이름: 게시
실행합니다: PUBLISH_GRAPH=true npm 실행 시
작 환경:
devtools_api_key: 변경_이_것을_귀하의_api_키로_변경합니다.
리포지토리_이름: {{ '${{' }} github.event.repository.name {{ '}}' }}
BRANCH_NAME: {{ '${{' }} github.head_ref || github.ref_name {{ '}}' }}
TARGET_SHA: {{ '${{' }} github.event.pull_request.base.sha {{ '}}' }}

```

이상적으로 `DEVTOOLS_API_KEY` 환경 변수는 Github 시크릿에서 검색해야 하며, 자세한 내용은 [여기](#)를 참조하세요.

이 워크플로는 **마스터 브랜치** 대상으로 하는 각 폴 리퀘스트마다 실행되거나 **마스터 브랜치에** 직접 커밋이 있는 경우 실행됩니다. 이 구성은 프로젝트에 필요한 모든 것에 맞춰 자유롭게 조정할 수 있습니다. 여기서 중요한 것은 [그래프 퍼블리셔](#) 클래스를 실행하는 데 필요한 환경 변수를 제공해야 한다는 것입니다.

하지만 이 워크플로를 사용하기 전에 업데이트해야 하는 변수가 하나 있는데, 바로 `DEVTOOLS_API_KEY`입니다. 이 페이지에서 프로젝트 전용 API 키를 생성할 수 있습니다.

마지막으로, 다시 `main.ts` 파일로 이동하여 이전에 비워둔 `publishOptions` 객체를 업데이트해 보겠습니다.

```

const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY, 리포지토리
  : process.env.REPOSITORY_NAME, 소유자:
  process.env.GITHUB_REPOSITORY_OWNER, sha:
  process.env.COMMIT_SHA,
  대상: process.env.TARGET_SHA,
  트리거: process.env.GITHUB_BASE_REF ? 'pull' : 'push', 브랜치:
  process.env.BRANCH_NAME,
};

```

최상의 개발자 환경을 위해 "Github 앱 통합" 버튼을 클릭하여 프로젝트의 Github 애플리케이션을 통합하세요(아래 스크린샷 참조). 참고 - 필수는 아닙니다.



이 통합 기능을 사용하면 폴 리퀘스트에서 바로 미리보기/리포트 생성 프로세스의 상태를 확인할 수 있습니다:



통합: Gitlab 파이프라인

먼저 프로젝트의 루트 디렉터리에 새 Gitlab CI 구성 파일을 생성하고 이름을 `.gitlab-ci.yml`로 지정합니다. 이 파일 안에 다음 정의를 사용하겠습니다:

```

const publishOptions = {
  apiKey: process.env.DEVTOOLS_API_KEY, 리포지토리
  : process.env.REPOSITORY_NAME, 소유자:
  process.env.GITHUB_REPOSITORY_OWNER, sha:
  process.env.COMMIT_SHA,
  대상: process.env.TARGET_SHA,
  트리거: process.env.GITHUB_BASE_REF ? 'pull' : 'push', 브랜치:
  process.env.BRANCH_NAME,
};

```

정보 힌트 이상적으로는 `DEVTOOLS_API_KEY` 환경 변수를 시크릿에서 검색해야 합니다.

이 워크플로는 **마스터 브랜치** 대상으로 하는 각 폴 리퀘스트마다 실행되거나 **마스터 브랜치에** 직접 커밋이 있는 경우 실행됩니다. 이 구성은 프로젝트에 필요한 모든 것에 맞춰 자유롭게 조정할 수 있습니다. 여기서 중요한 것은 [그래프 퍼블리셔](#) 클래스가 실행되는 데 필요한 환경 변수를 제공해야 한다는 것입니다.

하지만 이 워크플로우를 사용하기 전에 업데이트해야 하는 변수(이 워크플로우 정의에서)가 하나 있는데, 바로 `DEVTOOLS_API_KEY`입니다. 이 페이지에서 프로젝트 전용 API 키를 생성할 수 있습니다.

마지막으로, 다시 `main.ts` 파일로 이동하여 이전에 비워둔 `publishOptions` 객체를 업데이트해 보겠습니다.

이미지: 노드:16단

계:

- 빌드 캐시:

키를 누릅니다:

파일을 만듭니다:

- 패키지-잠금.json 경로:
- node_modules/

워크플로: 규칙:

- if: CI_PIPELINE_SOURCE == "merge_request_event"
 - when: 항상
- if: CI_COMMIT_BRANCH == "마스터" && \$CI_PIPELINE_SOURCE == "푸시" 언제
 - : 항상
- 언제: 절대

설치_종속성: 단계: 빌드

스크립트:

- npm CI

게시_그래프: 단계

: 빌드 요구 사

항:

- 설치_종속성 스크립트:

npm 실행 시작 변수:

```
PUBLISH_GRAPH: 'true'  
devtools_api_key: 'change_this_to_our_api_key'
```

기타 CI/CD 도구

Nest Devtools CI/CD 통합은 원하는 모든 CI/CD 도구(예: [Bitbucket Pipelines](#), [CircleCI](#) 등)와 함께 사용할 수 있으므로 여기에서 설명한 제공업체에 국한되지 않습니다.

다음 게시 옵션 개체 구성을 살펴보고 특정 커밋/빌드/PR에 대한 그래프를 게시하는 데 필요한 정보를 이해하세요.

```
const publishOptions = {  
  apiKey: process.env.DEVTOOLS_API_KEY, 리포지토리  
  : process.env.CI_PROJECT_NAME, 소유자:  
  process.env.CI_PROJECT_ROOT_NAMESPACE, sha:  
  process.env.CI_COMMIT_SHA,  
  대상: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA,  
  트리거: process.env.CI_MERGE_REQUEST_DIFF_BASE_SHA ? 'pull' : 'push',  
  branch:  
    process.env.CI_COMMIT_BRANCH ??  
    process.env.CI_MERGE_REQUEST_SOURCE_BRANCH_NAME,  
};
```

이 정보의 대부분은 CI/CD 기본 제공 환경 변수를 통해 제공됩니다([CircleCI 기본 제공 환경 목록](#) 및 [Bitbucket 변수](#) 참조). 그래프 게시를 위한 파이프라인

구성의 경우, 다음 트리거를 사용하는 것이 좋습니다:

- 푸시 이벤트 - 현재 브랜치가 배포 환경(예: 마스터, 메인, 스테이징, 프로덕션 등)을 나타내는 경우에만 해당됩니다.
- 풀 리퀘스트 이벤트 - 항상 또는 대상 브랜치가 배포 환경을 나타내는 경우(위 참조)

TypeScript 및 GraphQL의 강력한 기능 활용하기

GraphQL은 API를 위한 강력한 쿼리 언어이자 기존 데이터로 이러한 쿼리를 수행하기 위한 런타임입니다. 이는 REST API에서 일반적으로 발견되는 많은 문제를 해결하는 우아한 접근 방식입니다. 배경 지식이 필요하다면 GraphQL과 REST를 [비교한](#) 이 글을 읽어보시기 바랍니다. GraphQL과 [TypeScript](#)를 결합하면 GraphQL 쿼리의 유형 안전성을 개선하여 엔드투엔드 타이핑을 개발할 수 있습니다.

이 장에서는 GraphQL에 대한 기본적인 이해를 가정하고, 기본 제공 [@nestjs/graphql](#) 모듈로 작업하는 방법을 중점적으로 설명합니다. [GraphQL](#) 모듈은 [Apollo](#) 서버([@nestjs/apollo](#) 드라이버 사용)와 [Mercurius](#)([@nestjs/mercurius](#) 사용)를 사용하도록 구성할 수 있습니다. 이러한 검증된 GraphQL 패키지에 대한 공식 통합을 제공하여 Nest에서 GraphQL을 사용하는 간단한 방법을 제공합니다(더 많은 통합은 [여기](#)에서 참조하세요).

전용 드라이버를 직접 빌드할 수도 있습니다(자세한 내용은 [여기](#)를 참조하세요). 설

치

필요한 패키지를 설치하는 것으로 시작하세요:

```
# Express 및 Apollo의 경우(기본값)
$ npm i @nestjs/graphql @nestjs/apollo @apollo/server graphql

# Fastify 및 Apollo의 경우
# npm i @nestjs/graphql @nestjs/apollo @apollo/server @as-
integrations/fastify graphql

# Fastify 및 Mercurius용
# npm i @nestjs/graphql @nestjs/mercurius graphql mercurius
```

경고 경고 [@nestjs/graphql@>=9](#) 및 [@nestjs/apollo^10](#) 패키지는 Apollo v3와 호환되지만(자세한 내용은 Apollo Server 3 [マイグ레이션 가이드](#) 참조), [@nestjs/graphql@^8](#)은 Apollo v2(예: [apollo-server-express@2.x.x](#) 패키지)만 지원합니다.

개요

Nest는 코드 우선 방식과 스키마 우선 방식의 두 가지 GraphQL 애플리케이션 구축 방법을 제공합니다. 자신에게 가장 적합한 방법을 선택해야 합니다. 이 GraphQL 섹션의 대부분의 장은 코드 우선 방식을 채택하는 경우 따라야 할 부분과 스키마 우선 방식을 채택하는 경우 사용할 부분으로 나뉩니다.

코드 우선 접근 방식에서는 데코레이터와 TypeScript 클래스를 사용하여 해당 GraphQL 스키마를 생성합니다. 이 접근 방식은 TypeScript로만 작업하고 언어 구문 간의 컨텍스트 전환을 피하려는 경우에 유용합니다.

스키마 우선 접근 방식에서 진실의 소스는 GraphQL SDL(스키마 정의 언어) 파일입니다. SDL은 서로 다른 플랫폼 간에 스키마 파일을 공유할 수 있는 언어에 구애받지 않는 방법입니다. Nest는 다음을 자동으로 생성합니다.

클래스 또는 인터페이스를 사용하는 타입스크립트 정의는 GraphQL 스키마를 기반으로 하여 중복 상용구 코드를 작성할 필요성을 줄여줍니다.

GraphQL 및 TypeScript 시작하기

정보 힌트 다음 챕터에서는 [@nestjs/apollo](#) 패키지를 통합할 것입니다. 대신 [머큐리우스](#) 패키지를 사용 하려면 [이 섹션으로](#) 이동하세요.

패키지가 설치되면 [GraphQLModule](#)을 임포트하고 [forRoot\(\)](#)를 사용하여 구성할 수 있습니다.

정적 메서드입니다.

@@파일명()

```
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;  
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트합니다;
```

모듈({ import:

```
[  
    GraphQLModule.forRoot<ApolloDriverConfig>({  
        driver: ApolloDriver,  
    }),  
,  
)
```

정보 힌트 [클래스 ApolloModule](#)에 대해서는 [머큐리우스](#) 드라이버와

[MercuriusDriverConfig](#)을 대신 사용하세요. 둘 다 [@nestjs/mercurius](#) 패키지에서 내보냅니다.

[forRoot\(\)](#) 메서드는 옵션 객체를 인자로 받습니다. 이러한 옵션은 기본 드라이버 인스턴스로 전달됩니다(사용 가능한 설정에 대한 자세한 내용은 [Apollo](#) 및 [Mercurius](#)를 참조하세요). 예를 들어 [플레이그라운드](#)를 비활성화하고 [디버그](#) 모드(Apollo의 경우)를 끄려면 다음 옵션을 전달합니다:

@@파일명()

'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트합니다;

모듈({ import:

```
[  
  GraphQLModule.forRoot<ApolloDriverConfig>({  
    driver: ApolloDriver,  
    놀이터: 거짓,  
  }),  
,  
})
```

내보내기 클래스 AppModule {}

이 경우 이러한 옵션은 ApolloServer 생성자에게 전달됩니다.

GraphQL 놀이터

플레이그라운드는 그래픽 인터랙티브 브라우저 내 GraphQL IDE로, 기본적으로 GraphQL 서버 자체와 동일한 URL에서 사용할 수 있습니다. 플레이그라운드에 액세스하려면 기본 GraphQL 서버가 구성 및 실행 중이어야 합니다. 지금 바로 확인하려면 [여기에서 작동하는 예제를 설치 및 빌드할 수 있습니다](#). 또는 이 코드 샘플을 따라 하는 경우 [리졸버 챕터의 단계](#)를 완료한 후 플레이그라운드에 액세스할 수 있습니다.

애플리케이션이 백그라운드에서 실행 중인 상태에서 웹 브라우저를 열고 `http://localhost:3000/graphql`(호스트와 포트는 구성에 따라 다를 수 있음)로 이동하면 됩니다. 그러면 아래와 같이 GraphQL 플레이그라운드가 표시됩니다.

경고 참고 `@nestjs/mercurius` 통합은 기본 제공 GraphQL Playground 통합과 함께 제공되지 않습니다. 대신 [GraphQL을 사용할 수 있습니다](#)(`graphiql: true` 설정).

여러 엔드포인트

`nestjs/graphql` 모듈의 또 다른 유용한 기능은 한 번에 여러 엔드포인트를 제공할 수 있다는 점입니다. 이를 통해 어떤 모듈을 어떤 엔드포인트에 포함할지 결정할 수 있습니다. 기본적으로 `GraphQL`은 전체 앱에서 리졸버를 검색합니다. 이 검색을 모듈의 하위 집합으로만 제한하려면 `include` 속성을 사용하세요.

```
GraphQLModule.forRoot({  
  포함: [CatsModule],  
}) ,
```

경고 경고 단일 애플리케이션에서 여러 GraphQL 엔드포인트와 함께 `@apollo/server`와 `@as-integrations/fastify` 패키지를 사용하는 경우, `GraphQLModule` 구성에서 `disableHealthCheck` 설정을 활성화해야 합니다.

코드 우선

코드 우선 접근 방식에서는 데코레이터와 TypeScript 클래스를 사용하여 해당 GraphQL 스키마를 생성합니다.

코드 우선 접근 방식을 사용하려면 먼저 옵션 개체에 `autoSchemaFile` 속성을 추가합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),  
}),
```

자동 [스키마 파일](#) 속성 값은 자동으로 생성된 스키마가 생성될 경로입니다. 또는 스키마를 메모리에서 즉석에서 생성할 수도 있습니다. 이 기능을 사용하려면 `autoSchemaFile` 속성을 `true`로 설정합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  autoSchemaFile: true,  
}),
```

기본적으로 생성된 스키마의 유형은 포함된 모듈에 정의된 순서대로 정렬됩니다. 스키마를 사전순으로 정렬하려면 `sortSchema` 속성을 `true`로 설정하세요:

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  autoSchemaFile: join(process.cwd(), 'src/schema.gql'),  
  sortSchema: true,  
}),
```

예

완전히 작동하는 코드 우선 샘플은 [여기에서](#) 확인할

수 있습니다. 스키마 우선

스키마 우선 접근 방식을 사용하려면 먼저 옵션 개체에 `typePaths` 속성을 추가합니다. 이 속성은 `typePaths` 속성은 `GraphQLModule`이 작성할 GraphQL SDL 스키마 정의 파일을 어디에서 찾아야 하는지를 나타냅니다. 이러한 파일은 메모리에 결합되므로 스키마를 여러 파일로 분할하여 해당 리졸버 근처에서 찾을 수 있습니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  typePaths: ['./**/*.graphql'],  
}),
```

일반적으로 GraphQL SDL 유형에 해당하는 TypeScript 정의(클래스 및 인터페이스)도 있어야 합니다. 해당 TypeScript 정의를 수작업으로 생성하는 것은 중복되고 지루한 작업입니다. SDL 내에서 변경할 때마다 TypeScript 정의도 조정해야 하므로 신뢰할 수 있는 단일 소스가 없습니다. 이 문제를 해결하기 위해 `@nestjs/graphql` 패키지는 추상 구문 트리(AST)에서 TypeScript 정의를 자동으로 생성할 수 있습니다. 이 기능을 사용하려면 `GraphQLModule`을 구성할 때 정의 옵션 속성을 추가하세요.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'], 정의:
  {
    경로: join(process.cwd(), 'src/graphql.ts'),
  },
}) ,
```

정의 객체의 경로 속성은 생성된 TypeScript 출력을 저장할 위치를 나타냅니다. 기본적으로 생성된 모든 TypeScript 유형은 인터페이스로 생성됩니다. 대신 클래스를 생성하려면 `출력As` 속성에 '`class`' 값을 지정합니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  typePaths: ['./**/*.graphql'], 정의:
  {
    경로: join(process.cwd(), 'src/graphql.ts'),
    outputAs: 'class',
  },
}),
```

위의 접근 방식은 애플리케이션이 시작될 때마다 TypeScript 정의를 동적으로 생성합니다. 또는 필요에 따라 생성하는 간단한 스크립트를 작성하는 것이 더 바람직할 수 있습니다. 예를 들어 다음 스크립트를 `generate-typings.ts`로 생성한다고 가정해 보겠습니다:

```
'@nestjs/graphql'에서 { GraphQLDefinitionsFactory }를 가져오고,
'path'에서 { join }를 가져옵니다;

const definitionsFactory = new GraphQLDefinitionsFactory();
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  경로: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
});
```

이제 이 스크립트를 필요에 따라 실행할 수 있습니다:

ts-node 생성 타이핑

정보 힌트 스크립트를 미리 컴파일한 후(예: `tsc`를 사용하여) `node`를 사용하여 실행할 수 있습니다.

스크립트에 대해 감시 모드를 사용하려면(`.graphql` 파일이 변경될 때마다 자동으로 타이핑을 생성하면) `generate()` 메서드에 `watch` 옵션을 전달합니다.

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  경로: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
  watch: true,
});
```

모든 객체 유형에 대해 추가 _____ **타입명** 필드를 자동으로 생성하려면, 모든 객체 유형에 대해 **emitTypenameField** 옵션을 사용합니다.

```
definitionsFactory.generate({
  // ...,
  emitTypenameField: true,
});
```

인수가 없는 일반 필드로 리졸버(쿼리, 돌연변이, 구독)를 생성하려면

skipResolverArgs 옵션.

```
definitionsFactory.generate({
  // ...,
  skipResolverArgs: true,
});
```

아폴로 샌드박스

로컬 개발용 GraphQL IDE로 **graphql-playground** 대신 [Apollo 샌드박스](#)를 사용하려면 다음 구성을 사용하세요:

```
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트하고
, '@nestjs/common'에서 { Module }을 임포트합니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;
'@apollo/server/plugin/landingPage/default'에서 {
ApolloServerPluginLandingPageLocalDefault }를 임포트합니다;

모듈({ import:
[
  GraphQLModule.forRoot<ApolloDriverConfig>({
    driver: ApolloDriver,
    놀이터: 거짓,
    플러그인: [ApolloServerPluginLandingPageLocalDefault()],
  }),
],
})

내보내기 클래스 AppModule {}
```

예

완전히 작동하는 스키마 첫 번째 샘플은 [여기에서](#) 확인할 수 있습니다

다. 생성된 스키마에 액세스하기

일부 상황(예: 엔드투엔드 테스트)에서는 생성된 스키마 객체에 대한 참조를 얻고 싶을 수 있습니다. 그러면 엔드투엔드 테스트에서 HTTP 리스너를 사용하지 않고 `graphql` 객체를 사용하여 쿼리를 실행할 수 있습니다.

코드 우선 또는 스키마 우선 접근 방식 중 하나를 사용하여 생성된 스키마에 액세스할 수 있습니다.

`GraphQLSchemaHost` 클래스:

```
const { schema } = app.get(GraphQLSchemaHost);
```

정보 힌트 애플리케이션이 초기화된 후(`app.listen()` 또는 `app.init()` 메서드에 의해 `onModuleInit` 훅이 트리거된 후) `GraphQLSchemaHost#schema` 게터를 호출해야 합니다.

비동기 구성

모듈 옵션을 정적이 아닌 비동기적으로 전달해야 하는 경우, `forRootAsync()`

메서드를 사용합니다. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성을 처리하는 몇 가지 기술을 제공합니

다. 한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  useFactory: () => ({
    typePaths: ['./**/*.graphql'],
  }),
}) ,
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 [비동기화될](#) 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.
주입합니다.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({
  driver: ApolloDriver,
  임포트: [구성 모듈],
  useFactory: async (configService: ConfigService) => ({
    typePaths: configService.get<string>('GRAPHQL_TYPE_PATHS'),
  }),
  주입합니다: [구성 서비스],
}) ,
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 [GraphQLModule](#)을 구성할 수도 있습니다:

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  사용 클래스: GqlConfigService,  
}),
```

위의 구성은 `GraphQLModule` 내부에 `GqlConfigService`를 인스턴스화하여 이를 사용하여 옵션 객체를 생성합니다. 이 예제에서 `GqlConfigService`는 아래와 같이 `GqlOptionsFactory` 인터페이스를 구현해야 한다는 점에 유의하세요. `GraphQL모듈`은 제공된 클래스의 인스턴스화된 객체에서 `createGqlOptions()` 메서드를 호출합니다.

```
@Injectable()
GqlConfigService 클래스는 GqlOptionsFactory { createGqlOptions() 를
구현합니다: ApolloDriverConfig {
    반환 {
        typePaths: ['./**/*.graphql'],
    };
}
```

내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려는 경우

`GraphQLModule`에 `사용Existing` 구문을 사용합니다.

```
GraphQLModule.forRootAsync<ApolloDriverConfig>({ 임포트:
    [ConfigModule],
    사용Existing: ConfigService,
}),
```

머큐리우스 통합

아폴로를 사용하는 대신 Fastify 사용자(자세한 내용은 [여기](#)를 참조)는 `@nestjs/mercurius`를 사용할 수 있습니다.
드라이버.

@@파일명()

'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트하고,
'@nestjs/mercurius'에서 { MercuriusDriver,
MercuriusDriverConfig }를 임포트합니다;

모듈({ import:

```
[  
  GraphQLModule.forRoot<MercuriusDriverConfig>({  
    driver: MercuriusDriver,  
    graphiql: true,  
  }),  
,  
)
```

내보내기 클래스 AppModule {}

정보 힌트 애플리케이션이 실행 중이면 브라우저를 열고 다음 위치로 이동합니다.

<http://localhost:3000/graphiql>. GraphQL IDE가 표시되어야 합니다.

`forRoot()` 메서드는 옵션 객체를 인자로 받습니다. 이러한 옵션은 기본 드라이버 인스턴스로 전달됩니다. 사용 가능한 설정에 대한 자세한 내용은 [여기](#)를 참조하세요.

타사 통합

- [GraphQL 요가](#)

예제

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

리졸버

리졸버는 [GraphQL](#) 연산(쿼리, 변이 또는 구독)을 데이터로 변환하기 위한 지침을 제공합니다. 스키마에서 지정한 것과 동일한 형태의 데이터를 동기식으로 또는 해당 형태의 결과로 해결되는 프로미스로 반환합니다. 일반적으로 리졸버 맵은 수동으로 생성합니다. 반면에 [@nestjs/graphql](#) 패키지는 클래스에 주석을 다는 데 사용하는 데코레이터가 제공하는 메타데이터를 사용하여 리졸버 맵을 자동으로 생성합니다. 패키지 기능을 사용하여 GraphQL API를 생성하는 과정을 보여드리기 위해 간단한 작성자 API를 만들어 보겠습니다.

코드 우선

코드 우선 접근 방식에서는 GraphQL SDL을 직접 작성하여 GraphQL 스키마를 생성하는 일반적인 프로세스를 따르지 않습니다. 대신 TypeScript 데코레이터를 사용하여 TypeScript 클래스 정의에서 SDL을 생성합니다. [nestjs/graphql](#) 패키지는 데코레이터를 통해 정의된 메타데이터를 읽고 스키마를 자동으로 생성합니다.

객체 유형

GraphQL 스키마에 있는 대부분의 정의는 객체 유형입니다. 정의하는 각 객체 유형은 애플리케이션 클라이언트가 상호 작용해야 할 수 있는 도메인 객체를 나타내야 합니다. 예를 들어 샘플 API는 작성자와 해당 글의 목록을 가져올 수 있어야 하므로 이 기능을 지원하기 위해 [작성자](#) 유형과 [글](#) 유형을 정의해야 합니다.

스키마 우선 접근 방식을 사용한다면 다음과 같이 SDL로 스키마를 정의할 것입니다:

```
type Author {  
  id: Int!  
  이름: 성: 문자열입니다:  
  문자열 게시물:  
  [Post!]!  
}
```

이 경우 코드 우선 접근 방식을 사용하면 TypeScript 클래스를 사용하여 스키마를 정의하고 TypeScript 데코레이터를 사용하여 해당 클래스의 필드에 주석을 달 수 있습니다. 코드 우선 접근 방식에서 위의 SDL에 해당하는 것은 다음과 같습니다:

@@파일명(저자/모델/저자.모델)

'@nestjs/graphql'에서 { Field, Int, ObjectType } 가져오기;

'./post'에서 { Post } 가져오기;

객체 유형()

```
export class Author {
  @Field(type => Int)
  id: number;

  @Field({ nullable: true })
  firstName?: 문자열;
```

```

@Field({ nullable: true })
lastName?: 문자열;

필드(유형 => [게시물]) 게시물
: Post[];
}

```

정보 힌트 TypeScript의 메타데이터 반영 시스템에는 몇 가지 제한 사항이 있어 클래스가 어떤 속성으로 구성되어 있는지 확인하거나 주어진 속성이 선택 사항인지 필수 사항인지 인식하는 것이 불가능합니다. 이러한 제한 때문에 스키마 정의 클래스에서 `@Field()` 데코레이터를 명시적으로 사용하여 각 필드의 GraphQL 유형 및 선택 가능성에 대한 메타데이터를 제공하거나 [CLI 플러그인](#)을 사용하여 이러한 메타데이터를 생성해야 합니다.

다른 클래스와 마찬가지로 `Author` 객체 유형은 필드 모음으로 구성되며 각 필드는 유형을 선언합니다. 필드의 유형은 [GraphQL 유형](#)에 해당합니다. 필드의 GraphQL 유형은 다른 객체 유형 또는 스칼라 유형일 수 있습니다. GraphQL 스칼라 유형은 단일 값으로 해석되는 기본 유형(예: `ID`, `문자열`, `부울` 또는 `Int`)입니다.

정보 힌트 GraphQL의 기본 제공 스칼라 유형 외에도 사용자 정의 스칼라 유형을 정의할 수 있습니다([자세히 읽기](#)).

위의 `Author` 객체 유형 정의는 Nest가 위에 표시된 SDL을 생성하도록 합니다:

```

type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 게시물:
  [Post!]!
}

```

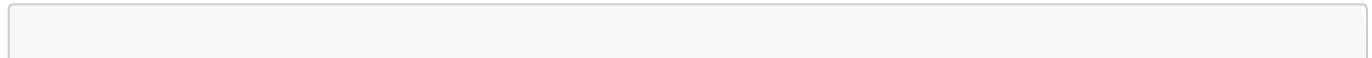
`필드()` 데코레이터는 선택적 타입 함수(예: `type => Int`)와 옵션 객체를 허용합니다.

타입 함수는 타입스크립트 타입 시스템과 GraphQL 타입 시스템 간에 모호한 부분이 있을 때 필요합니다. 구체적으로, `문자열` 및 `부울` 유형에는 필요하지 않지만 `숫자`(GraphQL `Int` 또는 `Float`에 매핑되어야 함)에는 필요합니다. 타입 함수는 단순히 원하는 GraphQL 타입을 반환해야 합니다(이 장의 다양한 예제에서 볼 수 있듯이).

옵션 객체는 다음 키/값 쌍 중 하나를 가질 수 있습니다:

- **nullable**: 필드의 널 가능 여부를 지정합니다(SDL에서 각 필드는 기본적으로 널 가능하지 않습니다);
부울
- **설명**: 필드 설명 설정용; **문자열**
- 사용 중단 **이유**: 필드를 사용 중단으로 표시하는 경우; **문자열**

예를 들어



```
@Field({ description: '책 제목', deprecationReason: 'v2 스키마에서 유용하지 않음' })
제목: 문자열;
```

정보 힌트 전체 객체 유형에 설명을 추가하거나 더 이상 사용하지 않을 수도 있습니다: 객체 유형({{ '{' }} 설명: '작성자 모델' {{ '}} }).

필드가 배열인 경우 아래와 같이 `Field()` 데코레이터의 유형 함수에서 배열 유형을 수동으로 표시해야 합니다:

```
필드(유형 => [게시물]) 게시물:
Post[];
```

정보 힌트 배열 괄호 표기법([])을 사용하면 배열의 깊이를 나타낼 수 있습니다. 예를 들어 [[Int]]를 사용하면 정수 행렬을 나타낼 수 있습니다.

배열의 항목(배열 자체가 아닌)이 널 가능함을 선언하려면 아래와 같이 널 가능 속성을 'items'로 설정합니다:

```
@Field(type => [Post], { nullable: 'items' })
posts: Post[];
```

정보 힌트 배열과 해당 항목이 모두 널러블인 경우, 대신 널러블을 'itemsAndList'로 설정하세요.

이제 작성자 객체 유형이 생성되었으므로 게시물 객체 유형을 정의해 보겠습니다.

@@파일명 (posts/models/post.model)

'@nestjs/graphql'에서 { Field, Int, ObjectType }을 가져옵니다;

객체 유형() 내보내기 클

래스 Post {

 @Field(유형 => Int)

 id: 숫자;

 @Field()

 title: 문자열;

 @Field(유형 => Int, { nullable: true })

 votes?: 숫자;

}

Post 객체 유형은 SDL에서 GraphQL 스키마의 다음 부분을 생성하게 됩니다:

```
유형 Post {
  id: Int!
  title: 문자열! 투표:
  Int
}
```

코드 우선 해결자

이 시점에서 데이터 그래프에 존재할 수 있는 객체(유형 정의)를 정의했지만 클라이언트는 아직 해당 객체와 상호 작용할 수 있는 방법이 없습니다. 이 문제를 해결하려면 리졸버 클래스를 만들어야 합니다. 코드 퍼스트 방법에서 리졸버 클래스는 리졸버 함수를 정의하고 쿼리 유형을 생성합니다. 이는 아래 예제를 진행하면서 명확해질 것입니다:

```
@@파일명(저자/저자.해결자) @Resolver(of =>
저자)

내보내기 클래스 AuthorsResolver { 생성자(
  비공개 작성자 서비스: AuthorsService, 비공개
  postsService: PostsService,
) {}

쿼리(반환값 => 작성자)
async author(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}

@ResolveField()
async posts(@Parent() author: Author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
}
```

정보 힌트 모든 데코레이터(예: `@Resolver`, `@ResolveField`, `@Args` 등)는 `nestjs/graphql` 패키지.

여러 리졸버 클래스를 정의할 수 있습니다. Nest는 런타임에 이를 결합합니다. 코드 구성에 대한 자세한 내용은 아래 [모듈](#) 섹션을 참조하세요.

경고 `AuthorsService` 및 `PostsService` 클래스 내부의 로직은 필요에 따라 단순하거나 정교하게 만들 수 있습니다. 이 예제의 요점은 리졸버를 구성하는 방법과 리졸버가 다른 공급자와 상호 작용하는 방법을 보여주기 위한 것입니다.

위의 예제에서는 하나의 쿼리 리졸버 함수와 하나의 필드 리졸버 함수를 정의하는 `AuthorsResolver`를 만들었습니다. 리졸버를 생성하려면 리졸버 함수를 메서드로 사용하는 클래스를 생성하고 `@Resolver()` 데코레이터를 사용하여 클래스에 주석을 달면 됩니다.

이 예제에서는 요청에 전송된 `ID`를 기반으로 작성자 개체를 가져오는 쿼리 처리기를 정의했습니다. 메서드가 쿼리 핸들러임을 지정하려면 `@Query()` 데코레이터를 사용합니다.

`Resolver()` 데코레이터에 전달되는 인수는 선택 사항이지만 그래프가 사소하지 않게 될 때 사용됩니다. 필드 리졸버 함수가 객체 그래프를 내려갈 때 사용하는 부모 객체를 제공하는 데 사용됩니다.

이 예제에서는 클래스에 필드 리졸버 함수(`Author` 객체 유형의 `게시물` 속성에 대한)가 포함되어 있으므로 이 클래스 내에 정의된 모든 필드 리졸버의 부모 유형(즉, 해당 `ObjectType` 클래스 이름)을 나타내는 값을 `@Resolver()` 데코레이터에 제공해야 합니다. 예제에서 알 수 있듯이 필드 해석기 함수를 작성할 때는 부모 객체(해석되는 필드가 멤버로 있는 객체)에 액세스해야 합니다. 이 예제에서는 작성자의 `ID`를 인수로 사용하는 서비스를 호출하는 필드 해석기로 작성자의 `게시물` 배열을 채웁니다. 따라서 `@Resolver()` 데코레이터에서 부모 객체를 식별해야 합니다. 필드 리졸버에서 해당 부모 객체에 대한 참조를 추출하려면 `@Parent()` 메서드 매개변수 데코레이터의 해당 사용에 유의하세요.

이 클래스와 다른 리졸버 클래스 모두에서 여러 개의 `@Query()` 리졸버 함수를 정의할 수 있으며, 이러한 함수는 리졸버 맵의 적절한 항목과 함께 생성된 SDL에서 단일 쿼리 유형 정의로 집계됩니다. 이를 통해 쿼리를 사용하는 모델 및 서비스에 가깝게 정의하고 모듈에서 잘 정리된 상태로 유지할 수 있습니다.

정보 힌트 네스트 CLI는 모든 상용구 코드를 자동으로 생성하는 제너레이터(회로도)를 제공하여 이 모든 작업을 피하고 개발자 환경을 훨씬 더 간단하게 만들 수 있도록 도와줍니다. 이 기능에 대한 자세한 내용은 [여기에서](#) 확인하세요.

쿼리 유형 이름

위의 예제에서 `@Query()` 데코레이터는 메서드 이름을 기반으로 GraphQL 스키마 쿼리 유형 이름을 생성합니다. 예를 들어 위의 예제에서 다음 구성을 고려해 보겠습니다:

```
쿼리(반환값 => 작성자)
async author(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

이렇게 하면 스키마에서 작성자 쿼리에 대한 다음 항목이 생성됩니다(쿼리 유형은 메서드 이름과 동일한 이름을

사용함):

```
유형 쿼리 {  
    author(id: Int!): 저자  
}
```

정보 힌트 GraphQL 쿼리에 대한 자세한 내용은 [여기](#)를 참조하세요.

일반적으로 이러한 이름을 분리하는 것을 선호합니다. 예를 들어 다음과 같은 이름을 사용하는 것을 선호합니다.

`getAuthor()` 메서드 대신 쿼리 핸들러 메서드를 사용하되 쿼리 유형 이름에는 여전히 `author`를 사용합니다. 동일한

는 필드 리졸버에 적용됩니다. 매팅 이름을 함수 호출의 인수로 전달하면 쉽게 이 작업을 수행할 수 있습니다.

쿼리() 및 `@ResolveField()` 데코레이터를 사용할 수 있습니다:

```
@@파일명(저자/저자.해결자) @Resolver(of =>
  저자)

내보내기 클래스 AuthorsResolver { 생성자(
  비공개 작성자 서비스: AuthorsService, 비공개
  postsService: PostsService,
) {}

쿼리(반환값 => 작성자, { 이름: '작성자' })
async getAuthor(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id)를 반환합니다;
}

ResolveField('posts', returns => [Post])
async getPosts(@Parent() author: Author) {
  const { id } = 작성자;
  return this.postsService.findAll({ authorId: id });
}
}
```

위의 `getAuthor` 핸들러 메서드는 SDL에서 GraphQL 스키마의 다음 부분을 생성하게 됩니다:

```
유형 쿼리 {
  author(id: Int!): 저자
}
```

쿼리 데코레이터 옵션

쿼리() 데코레이터의 옵션 객체(여기서 `{} ' {} '` 이름: 'author' `{} ' {} '`)

위)는 여러 키/값 쌍을 허용합니다:

- **이름:** 쿼리 이름; 문자열
- **설명:** GraphQL 스키마 문서를 생성하는 데 사용되는 설명(예: GraphQL 플레이그라운드); 문자열
- **deprecationReason:** 쿼리를 더 이상 사용되지 않는 것으로 표시하도록 쿼리 메타데이터를 설정합니다(예: GraphQL 플레이그라운드에서); 문자열
- **nullable:** 쿼리가 null 데이터 응답을 반환할 수 있는지 여부; 부울 또는 'items' 또는

'itemsAndList'('items' 및 'itemsAndList'에 대한 자세한 내용은 위 참조)

Args 데코레이터 옵션

메서드 핸들러에서 사용할 인수를 요청에서 추출하려면 `@Args()` 데코레이터를 사용합니다. 다음과 같이 작동합니다.

를 REST 경로 매개변수 인자 추출과 매우 유사한 방식으로 사용합니다.

일반적으로 `@Args()` 데코레이터는 간단하며, 위의 `getAuthor()` 메서드에서 볼 수 있듯이 객체 인수가 필요하지 않습니다. 예를 들어 식별자 유형이 문자열인 경우 다음과 같은 구조로 충분하며, 인바운드 GraphQL 요청에서 명명된 필드를 가져와 메서드 인수로 사용하기만 하면 됩니다.

```
@Args('id') id: 문자열
```

`getAuthor()`의 경우 `숫자` 유형이 사용되므로 문제가 발생합니다. `숫자` 타입스크립트 유형은 예상되는 GraphQL 표현에 대한 충분한 정보를 제공하지 않습니다(예: `Int` 대 `Float`). 따라서 명시적으로 타입 참조를 전달해야 합니다. 이를 위해 아래 그림과 같이 인수 옵션을 포함하는 두 번째 인수를 `Args()` 데코레이터에 전달합니다:

```
쿼리(반환값 => 작성자, { 이름: '작성자' })
async getAuthor(@Args('id', { type: () => Int }) id: number) {
  return this.authorsService.findOneById(id)를 반환합니다;
}
```

옵션 객체를 사용하면 다음과 같은 선택적 키 값 쌍을 지정할 수 있습니다:

- `type`: GraphQL 유형을 반환하는 함수
- `defaultValue`: 기본값; `any`
- `설명`: 설명 메타데이터, `문자열`
- `deprecationReason`: 필드를 사용 중단하고 그 이유를 설명하는 메타 데이터를 제공합니다.
- `nullable`: 필드가 `null` 가능한지 여부입니다.

쿼리 핸들러 메서드는 여러 인수를 받을 수 있습니다. 이름과 성을 기준으로 작성자를 가져오고 싶다고 가정해 보겠습니다. 이 경우 `@Args`를 두 번 호출하면 됩니다:

```
getAuthor(
  @Args('firstName', { nullable: true }) firstName?: 문자열,
  @Args('lastName', { defaultValue: '' }) lastName?: 문자열입니다,
) {}
```

전용 인수 클래스

인라인 `@Args()` 호출을 사용하면 위 예제와 같은 코드가 부풀어 오릅니다. 대신 다음과 같이 전용

`GetAuthorArgs` 인수 클래스를 생성하고 핸들러 메서드에서 액세스하면 됩니다:

```
@Args() args: GetAuthorArgs
```

아래와 같이 `@ArgsType()`을 사용하여 `GetAuthorArgs` 클래스를 생성합니다:

```

@@파일명(authors/dto/get-author.args) 'class-
validator'에서 { minLength }를 가져옵니다;
'@nestjs/graphql'에서 { Field, ArgsType }을 가져옵니다;

@ArgsType()
GetAuthorArgs 클래스 {
  @Field({ nullable: true })
  firstName?: 문자열;

  @Field({ defaultValue: '' })
  @MinLength(3)
  성: 문자열입니다,
}

```

정보 힌트 다시 한 번 말씀드리지만, TypeScript의 메타데이터 반영 시스템 제한으로 인해 `@Field` 데코레이터를 사용하여 유형 및 선택 사항을 수동으로 표시하거나 [CLI 플러그인](#)을 사용해야 합니다.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```

유형 쿼리 {
  저자(이름: 문자열, 성: 문자열 = ''): Author
}

```

정보 힌트 GetAuthorArgs와 같은 인자 클래스는
유효성 검사 파이프 ([자세히 읽기](#)).

클래스 상속

표준 TypeScript 클래스 상속을 사용하여 확장할 수 있는 일반 유틸리티 유형 기능(필드 및 필드 속성, 유효성 검사 등)이 있는 기본 클래스를 만들 수 있습니다. 예를 들어, 항상 표준 오프셋 및 제한 필드뿐만 아니라 유형별로 다른 인덱스 필드를 포함하는 페이지 매김 관련 인수 집합을 가질 수 있습니다. 아래와 같이 클래스 계층 구조를 설정할 수 있습니다.

베이스 `@ArgsType()` 클래스:

```
@ArgsType()
PaginationArgs 클래스 {
  @Field((type) => Int)
  offset: number = 0;

  @Field((type) => Int)
  limit: number = 10;
}
```

기본 `@ArgsType()` 클래스의 특정 하위 클래스를 입력합니다:

```

@ArgsType()
GetAuthorArgs 클래스 PaginationArgs 확장 { @Field({
  nullable: true })
  firstName?: 문자열;

  @Field({ defaultValue: '' })
  @MinLength(3)
  성: 문자열입니다;
}

```

`ObjectType()` 객체에서도 동일한 접근 방식을 사용할 수 있습니다. 베이스 클래스에서 제네릭 프로퍼티를 정의합니다:

```

객체 유형() 클래스
Character {
  @Field((유형) => Int)
  id: 숫자;

  @Field() 이름:
  문자열;
}

```

하위 클래스에 유형별 속성을 추가합니다:

```

객체 유형()
Warrior 클래스 확장자 Character {
  @Field()
  수준: 숫자;
}

```

리졸버와 함께 상속을 사용할 수도 있습니다. 상속과 TypeScript 제네릭을 결합하여 유형 안전을 보장할 수 있습니다. 예를 들어, 일반 `findAll` 쿼리가 있는 기본 클래스를 만들려면 다음과 같은 구성을 사용합니다:

```
함수 BaseResolver<T extends Type<unknown>>(<code>classRef: T</code>): any {
  @Resolver({ isAbstract: true })
  추상 클래스 BaseResolverHost {
    @Query((type) => [classRef], { name: `findAll${classRef.name}` })
    async findAll(): Promise<T[]> {
      반환 [];
    }
  }
  BaseResolverHost를 반환합니다;
}
```

다음 사항에 유의하세요:

- 명시적 반환 유형(위 중 **하나**)이 필요합니다. 그렇지 않으면 TypeScript에서 비공개 클래스 정의 사용에 대해 불만을 표시합니다. 권장: **아무 것도** 사용하지 말고 인터페이스를 정의하세요.
- 유형은 **@nestjs/common** 패키지에서 가져옵니다.
- **isAbstract: true** 속성은 이 클래스에 대해 SDL(스키마 정의 언어 문)을 생성하지 않아야 함을 나타냅니다. 다른 유형에 대해서도 이 속성을 설정하여 SDL 생성을 억제할 수 있습니다.

베이스 리졸버의 구체적인 하위 클래스를 생성하는 방법은 다음과 같습니다:

```
@Resolver((of) => Recipe)
export class RecipesResolver extends BaseResolver(Recipe) {
  constructor(private recipesService: RecipesService) {
    super();
  }
}
```

이 구성은 다음과 같은 SDL을 생성합니다:

```
유형 쿼리 {
  findAllRecipe: [Recipe!]!
}
```

제네릭

위에서 제네릭의 한 가지 용도를 살펴봤습니다. 이 강력한 타입스크립트 기능은 유용한 추상화를 만드는데 사용할 수 있습니다. 예를 들어 [이 문서를](#) 기반으로 커서 기반 페이지 매김을 구현한 샘플은 다음과 같습니다:

'@nestjs/graphql'에서 { Field, ObjectType, Int }를 가져오고,
'@nestjs/common'에서 { Type }을 가져옵니다;

인터페이스 IEdgeType<T> { 커

서: 문자열;
노드: T;
}

내보내기 인터페이스 IPaginatedType<T> { 가

장자리: IEdgeType<T>[];
노드를 반환합니다: T[];
totalCount: 숫자;
hasNextPage: 부울;
}

내보내기 함수 Paginated<T>(classRef: Type<T>): Type<IPaginatedType<T>> {
@ObjectType(` \${classRef.name}Edge`)

```

추상 클래스 EdgeType {
    @Field((type) => String)
    cursor: string;

    @Field((type) => classRef) 노
    드: T;
}

객체 유형({ isAbstract: true })

추상 클래스 PaginatedType 구현 IPaginatedType<T> { @Field((type) =>
[EdgeType], { nullable: true })
에지: EdgeType[];

@Field((type) => [classRef], { nullable: true }) 노
드: T[];

@Field((type) => Int)
totalCount: 숫자;

@Field()
hasNextPage: 부울;
}

PaginatedType을 Type<IPaginatedType<T>>로 반환합니다;
}

```

위의 기본 클래스가 정의되었으므로 이제 이 동작을 상속하는 특수 유형을 쉽게 만들 수 있습니다. 예를 들어

객체 유형()

PaginatedAuthor 클래스 Paginated(Author) 확장 {}

스키마 우선

이전 장에서 언급했듯이, 스키마 우선 접근 방식에서는 SDL에서 스키마 유형을 수동으로 정의하는 것으로 시작합니다([자세히 보기](#)). 다음 SDL 유형 정의를 고려하세요.

정보 힌트 이 장에서는 편의를 위해 모든 SDL을 한 위치(예: 하나의

파일)에 정의합니다. 이를 통해 각 유형은 그 자체로 정의되며, 다른 유형은 그에 대한 참조만으로 정의됩니다.

```

type Author {
    id: Int!
    이름: 성: 문자열입니다:
    문자열 글입니다: [포스트
]

```



```

}

유형 Post {
  id: Int!
  title: 문자열! 투표:
  Int
}

유형 쿼리 {
  author(id: Int!): 저자
}

```

스키마 우선 해결자

위의 스키마는 단일 쿼리인 `author(id: Int!)`를 노출합니다: `Author`.

정보 힌트 GraphQL 쿼리에 대한 자세한 내용은 [여기](#)를 참조하세요.

이제 작성자 쿼리를 해결하는 `AuthorsResolver` 클래스를 만들어 보겠습니다:

```

@@파일명(저자/저자.해결자) @Resolver('저자')
내보내기 클래스 AuthorsResolver { 생성자(
  비공개 작성자 서비스: AuthorsService, 비공개
  postsService: PostsService,
) {}

쿼리()
async author(@Args('id') id: number) {
  this.authorsService.findOneById(id)를 반환합니다;
}

@ResolveField()
async posts(@Parent() author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
}

```

정보 힌트 모든 데코레이터(예: `@Resolver`, `@ResolveField`, `@Args` 등)는

`nestjs/graphql` 패키지.

경고 `AuthorsService` 및 `PostsService` 클래스 내부의 로직은 필요에 따라 단순하거나 정교하게 만들 수 있습니다. 이 예제의 요점은 리졸버를 구성하는 방법과 리졸버가 다른 공급자와 상호 작용하는 방법을 보여주기 위한 것입니다.

`Resolver()` 데코레이터는 필수입니다. 이 데코레이터는 클래스 이름이 포함된 선택적 문자열 인수를 받습니다.
이 클래스 이름은 클래스에 `@ResolveField()` 데코레이터가 포함될 때마다 Nest에 해당 클래스가

장식된 메서드는 부모 유형(현재 예제에서는 `Author` 유형)과 연관되어 있습니다. 또는 클래스 상단에 `@Resolver()`를 설정하는 대신 각 메서드에 대해 이 작업을 수행할 수 있습니다:

```
Resolver('Author')
@ResolveField()
async posts(@Parent() author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
```

이 경우(메서드 레벨의 `@Resolver()` 데코레이터), 클래스 내에 여러 개의 `@ResolveField()` 데코레이터가 있는 경우 모든 데코레이터에 `@Resolver()`를 추가해야 합니다. 이는 추가 오버헤드가 발생하므로 모범 사례로 간주되지 않습니다.

정보 힌트 `@Resolver()`에 전달된 클래스 이름 인수는 쿼리(`@Query()` 데코레이터) 또는 돌연변이

(`@Mutation()` 데코레이터)에 영향을 주지 않습니다.

경고 메서드 수준에서 `@Resolver` 데코레이터를 사용하는 것은 코드 우선 접근 방식에서 지원되지 않습니다.

위의 예에서 `@Query()` 및 `@ResolveField()` 데코레이터는 메서드 이름에 따라 GraphQL 스키마 유형에 연결됩니다. 예를 들어 위의 예제에서 다음 구성을 고려해 보겠습니다:

```
쿼리()
async author(@Args('id') id: number) {
  this.authorsService.findOneById(id)를 반환합니다;
}
```

이렇게 하면 스키마에서 작성자 쿼리에 대한 다음 항목이 생성됩니다(쿼리 유형은 메서드 이름과 동일한 이름을 사용함):

```
유형 쿼리 {
  author(id: Int!): 저자
}
```

일반적으로는 리졸버 메서드에 `getAuthor()` 또는 `getPosts()`와 같은 이름을 사용하여 이를 분리하는 것을 선호

합니다. 아래 그림과 같이 데코레이터에 맵핑 이름을 인자로 전달하면 쉽게 분리할 수 있습니다:

```
@@파일명(저자/저자.해결자) @Resolver('저자')  
내보내기 클래스 AuthorsResolver { 생성자(  
    비공개 authorsService: AuthorsService,
```

```

비공개 게시 서비스: 게시 서비스,
) {}

@Query('author')
async getAuthor(@Args('id') id: number) {
  return this.authorsService.findOneById(id);
}

@ResolveField('posts')
async getPosts(@Parent() author) {
  const { id } = author;
  return this.postsService.findAll({ authorId: id });
}
}

```

정보 힌트 네스트 CLI는 모든 상용구 코드를 자동으로 생성하는 제너레이터(회로도)를 제공하여 이 모든 작업을 피하고 개발자 환경을 훨씬 더 간단하게 만들 수 있도록 도와줍니다. 이 기능에 대한 자세한 내용은

[여기에서](#) 확인하세요.

유형 생성

스키마 우선 접근 방식을 사용하고 유형 생성 기능을 활성화했다고 가정하면([이전](#) 장에 표시된 것처럼 `출력As: 'class'` 사용), 애플리케이션을 실행하면 다음 파일이 생성됩니다(`GraphQLModule.forRoot()` 메서드에서 지정 한 위치에). 예를 들어, `src/graphql.ts`에 있습니다:

`@@파일명`(그래프명) `내보내기`

```

@class 저자 {
  id: 숫자; 이름?: 문자
  열; 성?: 문자열;
  posts?: Post[];
}
export class Post {
  id: 숫자; title:
  문자열; votes?: 숫
  자;
}

```

`내보내기` 추상 클래스 `IQuery` {

```

추상 저자(ID: 숫자): 저자 | 약속<저자>;
}

```

인터페이스를 생성하는 기본 기법 대신 클래스를 생성하면 스키마 우선 접근 방식과 함께 선언적 유효성 검사 데코레이터를 사용할 수 있으며, 이는 매우 유용한 기법입니다([자세히](#) 읽기). 예를 들어, 아래와 같이 생성된 `CreatePostInput` 클래스에 [클래스 유효성](#) 검사 데코레이터를 추가하여 [제목](#) 필드에 최소 및 최대 문자열 길이를 적용할 수 있습니다:

'class-validator'에서 { minLength, maxLength }를 가져옵니다;

```
export 클래스 CreatePostInput {
    @MinLength(3)
    @MaxLength(50)
    title: 문자열;
}
```

경고 주의 입력(및 매개변수)의 자동 유효성 검사를 사용하려면 ValidationPipe를 사용하세요. 유효성 검사에 대한 자세한 내용은 [여기](#), 파이프에 대한 자세한 내용은 [여기](#) 참조하세요.

그러나 자동 생성된 파일에 데코레이터를 직접 추가하면 파일이 생성될 때마다 덮어쓰게 됩니다. 대신 별도의 파일을 생성하고 생성된 클래스를 확장하기만 하면 됩니다.

'class-validator'에서 { minLength, maxLength }를 가져오고,

'.../graphql.ts'에서 { Post }를 가져옵니다;

```
export 클래스 CreatePostInput extends Post {
    @MinLength(3)
    @MaxLength(50)
    title: 문자열;
}
```

GraphQL 인수 데코레이터

전용 데코레이터를 사용하여 표준 GraphQL 리졸버 인자에 액세스할 수 있습니다. 아래는 Nest 데코레이터와 이를 나타내는 일반 아폴로 매개변수를 비교한 것입니다.

루트() 및 @부모()

루트/부모 @컨텍스트(파라미터)

? : 문자열) context /context[파라미터] @정보(파라미터? :

문자열)

정보/정보[파라미터] @아르그(

파라미터? : 문자열)

args/ args[파라미터]

이러한 인수의 의미는 다음과 같습니다:

- **root**: 상위 필드에 있는 리졸버에서 반환된 결과를 포함하는 객체 또는 최상위 쿼리 필드의 경우 서버

구성에서 전달된 `rootValue`입니다.

- `컨텍스트`: 특정 쿼리의 모든 리졸버가 공유하는 객체로, 일반적으로 요청별 상태를 포함하는 데 사용됩니다.
- `정보`: 쿼리 실행 상태에 대한 정보를 포함하는 객체입니다.
- `args`: 쿼리에서 필드에 전달된 인수가 있는 객체입니다.

모듈

위의 단계를 완료하면, 리졸버 맵을 생성하는 데 필요한 모든 정보를 [GraphQLModule](#)에 선언적으로 지정했습니다. [그래프QL모듈은](#) 리플렉션을 사용하여 데코레이터를 통해 제공된 메타 데이터를 인트로스펙트하고 클래스를 올바른 리졸버 맵으로 자동 변환합니다.

처리해야 할 다른 유일한 작업은 리졸버 클래스([AuthorsResolver](#))를 제공하고(즉, 일부 모듈에 [공급자로](#) 나열) 모듈([AuthorsModule](#))을 어딘가에 가져와야 Nest가 이를 활용할 수 있습니다.

예를 들어 이 컨텍스트에 필요한 다른 서비스도 제공할 수 있는 [AuthorsModule](#)에서 이 작업을 수행할 수 있습니다. [AuthorsModule](#)을 루트 모듈이나 루트 모듈에서 임포트한 다른 모듈 등 어딘가에 임포트해야 합니다.

@@파일명 (authors/authors.module)

```
@Module({
  수입: [PostsModule],
  공급자: [AuthorsService, AuthorsResolver],
})
```

내보내기 [클래스](#) [AuthorsModule](#) {}

정보 힌트 소위 도메인 모델별로 코드를 구성하는 것이 도움이 됩니다(REST API에서 엔트리 포인트를 구성하는 방식과 유사). 이 접근 방식에서는 도메인 모델을 나타내는 Nest 모듈 내에 모델([ObjectType](#) 클래스), 리졸버 및 서비스를 함께 보관하세요. 이러한 모든 구성 요소를 모듈당 하나의 폴더에 보관하세요. 이렇게 하고 [Nest](#) CLI를 사용하여 각 요소를 생성하면 Nest가 이러한 모든 부분을 자동으로 연결(적절한 폴더에 파일 찾기, [공급자](#) 및 [가져오기](#) 배열에 항목 생성 등)해 줍니다.

돌연변이

GraphQL에 대한 대부분의 논의는 데이터 가져오기에 초점을 맞추고 있지만, 완전한 데이터 플랫폼에는 서버 측 데이터를 수정할 수 있는 방법도 필요합니다. REST에서는 모든 요청이 서버에 부작용을 일으킬 수 있지만, 모범 사례에 따르면 GET 요청에서 데이터를 수정하지 않는 것이 좋습니다. GraphQL도 비슷합니다. 기술적으로는 모든 쿼리가 데이터 쓰기를 유발하도록 구현될 수 있습니다. 그러나 REST와 마찬가지로 쓰기를 유발하는 모든 작업은 변형을 통해 명시적으로 전송해야 한다는 규칙을 준수하는 것이 좋습니다(자세한 내용은 [여기](#)를 참조하세요).

공식 [아폴로](#) 문서에서는 `upvotePost()` 변이 예제를 사용합니다. 이 변이는 게시물의 `투표` 속성 값을 증가시키는 메서드를 구현합니다. Nest에서 이와 동등한 변형을 만들려면 `@Mutation()` 데코레이터를 사용하겠습니다.

코드 우선

이전 섹션에서 사용한 `AuthorResolver`에 다른 메서드를 추가해 보겠습니다([리졸버](#) 참조).

```
@Mutation(returns => Post)
async upvotePost(@Args({ name: 'postId', type: () => Int }) postId: number) {
  this.postsService.upvoteById({ id: postId })를 반환합니다;
}
```

정보 힌트 모든 데코레이터(예: `@Resolver`, `@ResolveField`, `@Args` 등)는 [nestjs/graphql](#) 패키지.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 돌연변이 { upvotePost(postId:
  Int!): Post
}
```

`upvotePost()` 메서드는 `postId(Int)`를 인자로 받아 업데이트된 `Post` 엔티티를 반환합니다. [리졸버](#) 섹션에서 설명한 이유 때문에 예상 유형을 명시적으로 설정해야 합니다.

돌연변이가 객체를 인자로 받아야 하는 경우 입력 유형을 만들 수 있습니다. 입력 유형은 인자로 전달할 수 있는 특수한 종류의 객체 유형입니다(자세한 내용은 [여기](#)를 참조하세요). 입력 유형을 선언하려면 `@InputType()` 데코레이터를 사용합니다.

```
'@nestjs/graphql'에서 { InputType, Field } 임포트;  
  
@InputType()  
내보내기 클래스 UpvotePostInput {  
    @Field()  
    postId: 번호;  
}
```

정보 힌트 `@InputType()` 데코레이터는 옵션 객체를 인수로 받으므로 예를 들어 입력 유형에 대한 설명을 지정할 수 있습니다. TypeScript의 메타데이터 반영 시스템 제한으로 인해 `@Field` 데코레이터를 사용하여 유형을 수동으로 표시하거나 [CLI 플러그인](#)을 사용해야 합니다.

그런 다음 리졸버 클래스에서 이 유형을 사용할 수 있습니다:

```
@Mutation(returns => Post)
async upvotePost(
  @Args('upvotePostData') upvotepostData: 업보트포스트입력,
) {}
```

스키마 우선

이전 섹션에서 사용한 `AuthorResolver`를 확장해 보겠습니다([리졸버](#) 참조).

```
@Mutation()
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

위에서 비즈니스 로직이 게시물을 쿼리하고 해당 `투표` 속성을 증가시키는 `PostsService`로 옮겨졌다고 가정했습니다. `PostsService` 클래스 내부의 로직은 필요에 따라 단순하거나 정교하게 만들 수 있습니다. 이 예제의 요점은 리졸버가 다른 공급자와 상호 작용하는 방법을 보여주기 위한 것입니다.

마지막 단계는 기존 유형 정의에 돌연변이를 추가하는 것입니다.

```
type Author {  
    id: Int!  
    이름: 성: 문자열입니다:  
    문자열 글입니다: [포스트  
]  
}
```

```
유형 Post {  
    id: Int!  
    제목: 문자열 투표:  
    Int  
}
```

```
유형 쿼리 {  
    author(id: Int!): 저자  
}
```

```
유형 돌연변이 {
```

```
    upvotePost(postId: Int!): Post  
}
```

`upvotePost(postId: Int!): Post`: 이제 애플리케이션의 GraphQL API의 일부로 [포스트](#) 변형을 호출할 수 있습니다.

구독

쿼리를 사용하여 데이터를 가져오고 변형을 사용하여 데이터를 수정하는 것 외에도 GraphQL 사양은 구독이라는 세 번째 작업 유형을 지원합니다. GraphQL 구독은 서버에서 서버의 실시간 메시지를 수신하도록 선택한 클라이언트로 데이터를 푸시하는 방법입니다. 구독은 클라이언트에 전달할 필드 집합을 지정한다는 점에서 쿼리와 유사하지만, 단일 응답을 즉시 반환하는 대신 서버에서 특정 이벤트가 발생할 때마다 채널이 열리고 결과가 클라이언트로 전송됩니다.

구독의 일반적인 사용 사례는 새 개체 생성, 업데이트된 필드 등 특정 이벤트에 대해 클라이언트 측에 알리는 것입니다(자세한 내용은 [여기](#)를 참조하세요).

Apollo 드라이버로 구독 사용 설정하기

구독을 사용하도록 설정하려면 `installSubscriptionHandlers` 속성을 `true`로 설정합니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  installSubscriptionHandlers: true,
}),
```

경고 설치 구독 핸들러 구성 옵션은 최신 버전의 Apollo 서버에서 제거되었으며 이 패키지에서도 곧 더 이상 사용되지 않을 예정입니다. 기본적으로 `installSubscriptionHandlers`는 구독-트랜스포트-ws([자세히 보기](#))를 사용하도록 대체하지만, 대신 graphql-ws([자세히 보기](#)) 라이브러리를 사용할 것을 강력히 권장합니다.

대신 graphql-ws 패키지를 사용하도록 전환하려면 다음 구성을 사용합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  구독을 추가합니다: {
    'graphql-ws': true
  },
}),
```

정보 힌트 예를 들어 이전 버전과의 호환성을 위해 두 패키지(구독-트랜스포트-ws 및 그래프ql-ws)를 동시에 사용할 수도 있습니다.

코드 우선

코드 퍼스트 접근 방식을 사용하여 구독을 생성하려면 `@Subscription()` 데코레이터(`@nestjs/graphql` 패키지에서 내보낸 것)와 간단한 게시/구독 API를 제공하는 `graphql-subscriptions` 패키지의 `PubSub` 클래스를 사용합니다.

다음 구독 핸들러는 다음을 호출하여 이벤트 구독을 처리합니다.

`PubSub#asyncIterator`. 이 메서드는 단일 인자, `트리거 이름(triggerName)`을 받습니다.

이벤트 주제 이름입니다.

```
const pubSub = new PubSub();

@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  구독((반환값) => 댓글) commentAdded() {
    pubSub.asyncIterator('commentAdded')를 반환합니다;
  }
}
```

정보 힌트 모든 데코레이터는 [@nestjs/graphql](#) 패키지에서 내보내지만 [PubSub](#) 클래스는 [graphql-subscriptions](#) 패키지에서 내보냅니다.

경고 참고 [PubSub](#)는 간단한 [계시](#) 및 [구독 API](#)를 노출하는 클래스입니다. [여기에서](#) 자세히 알아보세요. Apollo 문서에서는 기본 구현이 프로덕션에 적합하지 않다고 경고하고 있습니다([여기에서](#) 자세히 읽어보세요). 프로덕션 앱은 외부 스토어에서 지원하는 [PubSub](#) 구현을 사용해야 합니다(자세한 내용은 [여기](#)를 참조하세요).

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 구독 { commentAdded() :
  코멘트!
}
```

구독은 정의상 구독의 이름을 키로 하는 단일 최상위 프로퍼티를 가진 객체를 반환한다는 점에 유의하세요. 이 이름은 구독 핸들러 메서드의 이름에서 상속되거나(예: 위의 [commentAdded](#)) , 아래 표시된 것처럼 키 [이름](#)을 두 번째 인수로 하는 옵션을 [@Subscription\(\)](#) 데코레이터에 전달하여 명시적으로 제공될 수 있습니다.

```
구독(반환값 => 댓글, { 이름: '댓글 추가됨',
}) subscribeToCommentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

이 구성은 이전 코드 샘플과 동일한 SDL을 생성하지만 메서드 이름을 구독에서 분리할 수 있습니다.

게시

이제 이벤트를 게시하기 위해 `PubSub#publish` 메서드를 사용합니다. 이 메서드는 오브젝트 그래프의 일부가 변경되었을 때 클라이언트 측 업데이트를 트리거하기 위해 변이 내에서 자주 사용됩니다. 예를 들어

```

@@파일명(posts/posts.resolver)
@Mutation(returns => Post)
비동기 추가 코멘트(
  @Args('postId', { type: () => Int }) postId: 숫자,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment });
  pubSub.publish('commentAdded', { commentAdded: newComment });
  return newComment;
}

```

`PubSub#publish` 메서드는 첫 번째 매개변수로 트리거 이름(이벤트 주제 이름이라고 생각하면 됩니다)을, 두 번째 매개변수로 이벤트 페이로드를 받습니다. 앞서 언급했듯이 구독은 정의에 따라 값을 반환하며 그 값은 모양을 갖습니다. 댓글 추가 구독에 대해 생성된 SDL을 다시 살펴보세요:

```

유형 구독 { commentAdded() :
  코멘트!
}

```

이는 구독이 최상위 프로퍼티 이름이 `commentAdded`이고 값이 `Comment` 객체인 객체를 반환해야 한다는 것을 알려줍니다. 여기서 주의해야 할 중요한 점은 `PubSub#publish` 메서드가 보내내는 이벤트 페이로드의 모양이 구독에서 반환할 것으로 예상되는 값의 모양과 일치해야 한다는 것입니다. 따라서 위의 예제에서 `pubSub.publish('commentAdded', {{ '{' }} commentAdded: newComment {{ '}' }})` 문은 적절한 모양의 페이로드가 포함된 `commentAdded` 이벤트를 게시합니다. 이러한 모양이 일치하지 않으면 GraphQL 유효성 검사 단계에서 구독이 실패합니다.

구독 필터링

특정 이벤트를 필터링하려면 `필터` 속성을 필터 함수로 설정합니다. 이 함수는 배열 `필터에` 전달된 함수와 유사하게 작동합니다. 이 함수에는 이벤트 페이로드가 포함된 `페이로드`(이벤트 게시자가 보낸 것)와 구독 요청 중에 전달된 인수를 받는 `변수의` 두 가지 인수가 필요합니다. 이 함수는 이 이벤트를 클라이언트 리스너에 게시할지 여부를 결정하는 부울을 반환합니다.

```
구독(반환 => 댓글, { 필터: (페이지로드, 변수)
=>
  payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: string) {
  return pubSub.asyncIterator('commentAdded')를
  반환합니다;
}
```

구독 페이지로드 변경

게시된 이벤트 페이로드를 변경하려면 `resolve` 속성을 함수로 설정합니다. 함수는 이벤트 게시자가 전송한 이벤트 페이로드를 수신하고 적절한 값을 반환합니다.

```
구독(반환 => 댓글, { 해결: 값 => 값,
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

경고 `resolve` 옵션을 사용하는 경우 언래핑된 페이로드를 반환해야 합니다(예: 이 예제에서는 `{}'{}'` `commentAdded: newComment {{ '}}' }}` 객체를 반환해야 합니다.)

주입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용합니다.

```
구독(반환값 => 댓글, { resolve(this:
  AuthorResolver, value) {
    // "this"는 "AuthorResolver" 반환 값의 인스턴스를 참조합니다;
  }
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

필터에서도 동일한 구조가 작동합니다:

```
구독(반환 => 댓글, {
  filter(this: AuthorResolver, payload, variables) {
    // "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환
    payload.commentAdded.title === variables.title;
  }
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

스키마 우선

Nest에서 이와 동등한 구독을 생성하기 위해 `@Subscription()` 데코레이터를 사용하겠습니다.

```
const pubSub = new PubSub();
```

```
@Resolver('Author')
내보내기 클래스 AuthorResolver {
    // ... 구독()
    commentAdded() {
        pubSub.asyncIterator('commentAdded')를 반환합니다;
    }
}
```

컨텍스트 및 인수를 기반으로 특정 이벤트를 필터링하려면 필터 속성을 설정합니다.

```
구독('commentAdded', { 필터: (페이지,
    변수) =>
    payload.commentAdded.title === variables.title,
})
commentAdded() {
    pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

게시된 페이지를 변경하려면 해결 함수를 사용할 수 있습니다.

```
구독('commentAdded', { resolve: value
    => value,
})
commentAdded() {
    pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

삽입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용하세요:

```
구독('commentAdded', { resolve(this:
    AuthorResolver, value) {
    // "this"는 "AuthorResolver" 반환 값의 인스턴스를 참조합니다;
    다;
})
commentAdded() {
    pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

필터에서도 동일한 구조가 작동합니다:

```
구독('commentAdded', {  
  filter(this: AuthorResolver, payload, variables) {
```

```
// "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환
payload.commentAdded.title === variables.title;
}
})
commentAdded() {
  pubSub.asyncIterator('commentAdded')를 반환합니다;
}
```

마지막 단계는 유형 정의 파일을 업데이트하는 것입니다.

```
type Author {
  id: Int!
  이름: 성: 문자열입니다:
  문자열 글입니다: [포스트
  ]
}

유형 Post {
  id: Int!
  제목: 문자열 투표:
  Int
}

유형 쿼리 {
  author(id: Int!): 저자
}

유형 코멘트 {
  id:
  문자열 내용입니다:
  문자열
}

유형 구독 {
  commentAdded(title: 문자열!): 코멘트
}
```

이렇게 해서 하나의 `commentAdded(제목: 문자열!)`를 만들었습니다: 맷글 구독을 생성했습니다. 전체 샘플 구현은 [여기에서](#) 확인할 수 있습니다.

PubSub

위에서 로컬 PubSub 인스턴스를 인스턴스화했습니다. 선호하는 접근 방식은 PubSub를 [프로바이더로](#) 정의하

고 생성자(@Inject()) 데코레이터 사용)를 통해 인스턴스를 주입하는 것입니다. 이렇게 하면 전체 애플리케이션에서 인스턴스를 재사용할 수 있습니다. 예를 들어, 다음과 같이 프로바이더를 정의한 다음 필요한 곳에 'PUB_SUB'을 주입합니다.

```
{
```

```
제공: 'PUB_SUB',
```

```
사용값: 새로운 PubSub(),  
}
```

구독 서버 사용자 지정

구독 서버를 사용자 지정하려면(예: 경로 변경) 구독 옵션 속성을 사용합니다.

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  구독을 설정합니다: {  
    '구독-트랜스포트-ws': { 경로:  
      '/graphql'  
    },  
  }  
}),
```

구독에 `graphql-ws` 패키지를 사용하는 경우 구독-트랜스포트-ws를 구독-트랜스포트-ws로 대체합니다.

키를 사용하여 다음과 같이 설정합니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({  
  driver: ApolloDriver,  
  구독을 추가합니다: {  
    'graphql-ws': {  
      경로: '/graphql'  
    },  
  }  
}),
```

웹소켓을 통한 인증

사용자가 인증되었는지 확인하는 것은 구독 옵션에서 지정할 수 있는 `onConnect` 콜백 함수 내에서 수행할 수 있습니다.

`onConnect`는 첫 번째 인자로 연결 매개변수인
구독 클라이언트 (자세히 읽기).

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  구독을 설정합니다: {
    '구독-트랜스포트-ws': { onConnect:
      (connectionParams) => {
        const authToken = connectionParams.authToken;
        if (!isValid(authToken)) {
          새로운 오류('токен이 유효하지 않습니다')를 던집니다;
        }
        // 토큰에서 사용자 정보 추출
    }
  }
})
```

```
const user = parseToken(authToken);
// 나중에 컨텍스트에 추가하기 위해 사용자 정보를 반환합니다 { 사용자
}를 반환합니다;
},
}
},
},
컨텍스트: ({ 연결 }) => {
// connection.context는 "onConnect" 콜백이 반환한 것과 동일합니다.
},
},
});
```

이 예의 인증 토큰은 연결이 처음 설정될 때 클라이언트에 의해 한 번만 전송됩니다. 이 연결로 이루어진 모든 구독은 동일한 인증 토큰을 가지며, 따라서 동일한 사용자 정보를 갖게 됩니다.

경고 구독-transport-ws에 연결이 onConnect 단계를 건너뛸 수 있는 버그가 있습니다([자세한 내용 참조](#)). 사용자가 구독을 시작할 때 onConnect가 호출되었다고 가정해서는 안 되며 항상 컨텍스트가 채워져 있는지 확인해야 합니다.

`graphql-ws` 패키지를 사용하는 경우, `onConnect` 콜백의 서명이 약간 달라집니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  driver: ApolloDriver,
  // 구독을 추가합니다: {
  'graphql-ws': {
    onConnect: (context: Context<any>) => {
      const { connectionParams, extra } = context;
      // 사용자 유효성 검사는 위 예제와 동일하게 유지됩니다.
      // 그래프QL-WS와 함께 사용할 경우 추가 컨텍스트 값을 추가 필드에 저장해야 합니다.
      extra.user = { user: {} };
    },
  },
  context: ({ extra }) => {
    // 이제 추가 필드를 통해 추가 컨텍스트 값에 액세스할 수 있습니다.
  },
});
```

Mercurius 드라이버로 구독 활성화하기

구독을 활성화하려면 구독 속성을 true로 설정합니다.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({  
  driver: MercuriusDriver,  
  구독: true,  
}),
```

정보 힌트 옵션 객체를 전달하여 사용자 지정 이미터를 설정하고 들어오는 연결의 유효성을 검사하는 등
의 작업을 수행할 수도 있습니다. [여기에서](#) 자세히 알아보세요([구독](#) 참조).

코드 우선

코드 퍼스트 접근 방식을 사용하여 구독을 생성하려면 [@Subscription\(\)](#) 데코레이터([@nestjs/graphql](#) 패키지에서 내보낸 것)와 간단한 게시/구독 API를 제공하는 [mercurius](#) 패키지의 [PubSub](#) 클래스를 사용합니다.

다음 구독 핸들러는 [PubSub#asyncIterator](#)를 호출하여 이벤트 구독을 처리합니다. 이 메서드는 이벤트 주제 이름에 해당하는 단일 인자 [triggerName](#)을 받습니다.

```
@Resolver((of) => Author)
export class AuthorResolver {
  // ...
  구독((반환값) => 댓글) commentAdded(@Context('pubsub')
    pubSub: PubSub) {
    pubSub.subscribe('commentAdded')를 반환합니다;
  }
}
```

정보 힌트 위 예제에서 사용된 모든 데코레이터는 [@nestjs/graphql](#)에서 내보낸 것입니다.

패키지로 내보내는 반면 [PubSub](#) 클래스는 [mercurius](#) 패키지에서 내보냅니다.

경고 참고 [PubSub](#)는 간단한 게시 및 구독 API를 노출하는 클래스입니다. 사용자 지정 [PubSub](#) 클래스를 등록하는 방법은 [이 섹션을](#) 참조하세요.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 구독 { commentAdded() :
  코멘트!
}
```

구독은 정의상 구독의 이름을 키로 하는 단일 최상위 프로퍼티를 가진 객체를 반환한다는 점에 유의하세요. 이 이름은 구독 핸들러 메서드의 이름에서 상속되거나(예: 위의 [commentAdded](#)) , 아래 표시된 것처럼 키 이름을 두 번째 인수로 하는 옵션을 [@Subscription\(\)](#) 데코레이터에 전달하여 명시적으로 제공될 수 있습니다.

```
구독(반환값 => 댓글, { 이름: '댓글 추가됨',
})
subscribeToCommentAdded(@Context('pubsub') pubSub: PubSub) {
    return pubSub.subscribe('commentAdded')를 반환합니다;
}
```

이 구성은 이전 코드 샘플과 동일한 SDL을 생성하지만 메서드 이름을 구독에서 분리할 수 있습니다.

게시

이제 이벤트를 게시하기 위해 `PubSub#publish` 메서드를 사용합니다. 이 메서드는 오브젝트 그래프의 일부가 변경되었을 때 클라이언트 측 업데이트를 트리거하기 위해 변이 내에서 자주 사용됩니다. 예를 들어

```
@@파일명(posts/posts.resolver)
@Mutation(returns => Post)
비동기 추가 코멘트(
  @Args('postId', { type: () => Int }) postId: 숫자,
  @Args('comment', { type: () => Comment }) comment: CommentInput,
  @Context('pubsub') pubSub: PubSub,
) {
  const newComment = this.commentsService.addComment({ id: postId, comment });
  await pubSub.publish({
    topic: 'commentAdded',
    payload: {
      댓글 추가됨: 새로운 댓글
    }
  });
  새로운 댓글을 반환합니다;
}
```

앞서 언급했듯이 구독은 정의에 따라 값을 반환하며 그 값에는 모양이 있습니다. 댓글 추가 구독에 대해 생성된 SDL을 다시 살펴보세요:

```
유형 구독 { commentAdded() :
  코멘트!
}
```

이는 구독이 최상위 프로퍼티 이름이 `commentAdded`이고 값이 `Comment` 객체인 객체를 반환해야 한다는 것을 알려줍니다. 여기서 주의해야 할 중요한 점은 `PubSub#publish` 메서드가 보내내는 이벤트 페이로드의 모양이 구독에서 반환할 것으로 예상되는 값의 모양과 일치해야 한다는 것입니다. 따라서 위의 예제에서는 `pubSub.publish({{ '{' }} 주제: 'commentAdded', payload: {{ '{' }} commentAdded: newComment {{ '}' }} {{ '}' }})` 문은 적절한 모양의 페이로드가 포함된 `commentAdded` 이벤트를 게시합니다. 이러한 모양이 일치하지 않으면 GraphQL 유효성 검사 단계에서 구독이 실패합니다.

구독 필터링

특정 이벤트를 필터링하려면 `필터` 속성을 필터 함수로 설정합니다. 이 함수는 배열 `필터에` 전달된 함수와 유사하

게 작동합니다. 이 함수에는 이벤트 페이로드가 포함된 페이로드(이벤트 게시자가 보낸 대로)와 구독 요청 중에 전달된 인수를 받는 변수의 두 가지 인수가 필요합니다. 이 함수는 이 이벤트를 클라이언트 리스너에 게시할지 여부를 결정하는 부울을 반환합니다.

```

구독(반환 => 댓글, { 필터: (페이지, 변수)
=>
    payload.commentAdded.title === variables.title,
})
commentAdded(@Args('title') title: 문자열, @Context('pubsub') pubSub: PubSub) {
    pubSub.subscribe('commentAdded')를 반환합니다;
}

```

주입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용합니다.

```

구독(반환 => 댓글, {
    filter(this: AuthorResolver, payload, variables) {
        // "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환
        payload.commentAdded.title === variables.title;
    }
})
commentAdded(@Args('title') title: 문자열, @Context('pubsub') pubSub: PubSub) {
    pubSub.subscribe('commentAdded')를 반환합니다;
}

```

스키마 우선

Nest에서 이와 동등한 구독을 생성하기 위해 `@Subscription()` 데코레이터를 사용하겠습니다.

```

const pubSub = new PubSub();

@Resolver('Author')
내보내기 클래스 AuthorResolver {
    // ... 구독()
    commentAdded(@Context('pubsub') pubSub: PubSub) {
        return pubSub.subscribe('commentAdded')를 반환합니다;
    }
}

```

컨텍스트 및 인수를 기반으로 특정 이벤트를 필터링하려면 `필터` 속성을 설정합니다.

```
구독('commentAdded', { 필터: (페이지로드,  
변수) =>  
    payload.commentAdded.title === variables.title,  
})  
commentAdded(@Context('pubsub') pubSub: PubSub) {
```

```
    pubSub.subscribe('commentAdded')를 반환합니다;  
}
```

주입된 공급자에 액세스해야 하는 경우(예: 외부 서비스를 사용하여 데이터 유효성을 검사하는 경우) 다음 구성을 사용합니다:

```
구독('commentAdded', {  
  filter(this: AuthorResolver, payload, variables) {  
    // "this"는 "AuthorResolver"의 인스턴스를 참조합니다. 반환  
    payload.commentAdded.title === variables.title;  
  }  
})  
commentAdded(@Context('pubsub') pubSub: PubSub) {  
  return pubSub.subscribe('commentAdded')를 반환합니다;  
}
```

마지막 단계는 유형 정의 파일을 업데이트하는 것입니다.

```
type Author {  
    id: Int!  
  
    이름: 성: 문자열입니다:  
  
    문자열 글입니다: [포스트  
]  
}  
  
유형 Post {  
    id: Int!  
  
    제목: 문자열 투표:  
        Int  
}  
  
유형 쿼리 {  
    author(id: Int!): 저자  
}  
  
유형 Comment {  
    id: 문자열 내용입  
  
    니다: 문자열  
}
```

```
유형 구독 {  
    commentAdded(title: 문자열!): 코멘트  
}
```

이렇게 해서 하나의 `commentAdded(제목: 문자열!)`를 만들었습니다: 댓글 구독을 생성합니다.

PubSub

위의 예에서는 기본 PubSub 이미터(mqemitter)를 사용했습니다. (프로덕션의 경우) 선호하는 접근 방식은 mqemitter-redis를 사용하는 것입니다. 또는 사용자 정의 PubSub 구현을 제공할 수도 있습니다(자세한 내용은 [여기를 참조하세요](#)).

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  구독입니다: {
    emitter: require('mqemitter-redis')({
      port: 6579,
      호스트: '127.0.0.1',
    }),
  },
});
```

웹소켓을 통한 인증

사용자가 인증되었는지 확인하는 것은 구독 옵션에서 지정할 수 있는 verifyClient 콜백 함수 내에서 수행 할 수 있습니다.

verifyClient는 요청의 헤더를 검색하는 데 사용할 수 있는 정보 객체를 첫 번째 인수로 받습니다.

```
GraphQLModule.forRoot<MercuriusDriverConfig>({
  driver: MercuriusDriver,
  구독입니다: {
    verifyClient: (정보, 다음) => {
      const authorization = info.req.headers?.authorization as string;
      if (!authorization?.startsWith('Bearer ')) {
        반환 다음(거짓);
      }
      next(true);
    },
  },
});
```

스칼라

GraphQL 객체 유형에는 이름과 필드가 있지만, 어느 시점에서는 이러한 필드가 구체적인 데이터로 해석되어야 합니다. 스칼라 유형이 바로 쿼리의 리프를 나타내는 역할을 합니다(자세한 내용은 [여기](#)를 참조하세요). GraphQL에는 다음과 같은 기본 유형이 포함되어 있습니다: `Int`, `Float`, `String`, `Boolean` 및 `ID`. 이러한 기본 제공 유형 외에도 사용자 정의 원자 데이터 유형(예: 날짜)을 지원해야 할 수도 있습니다.

코드 우선

코드 우선 접근 방식은 5개의 스칼라를 제공하며, 이 중 3개는 기존 GraphQL 유형의 간단한 별칭입니다.

- `ID`(GraphQL `ID`의 별칭) - 객체를 리페치하거나 캐시의 키로 자주 사용되는 고유 식별자를 나타냅니다.
- `Int`(GraphQL `Int`의 별칭) - 부호화된 32비트 정수
- `부동 소수점`(GraphQL `Float`의 별칭) - 부호화된 배정밀도 부동 소수점 값
- `GraphQLISODateTime` - UTC 기준 날짜-시간 문자열(기본적으로 날짜 유형을 나타내는 데 사용됨)
- `GraphQLTimestamp` - 날짜와 시간을 UNIX 시대 시작부터 밀리초 수로 나타내는 부호화된 정수입니다.

날짜 유형을 나타내기 위해 기본적으로 `GraphQLISODateTime`(예: `2019-12-03T09:54:33Z`)이 사용됩니다. 대신 `GraphQLTimestamp`을 사용하려면 다음과 같이 빌드스케마옵션 객체의 `dateScalarMode`를 `'timestamp'`로 설정합니다:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    dateScalarMode: '타임스탬프',
  }
}),
```

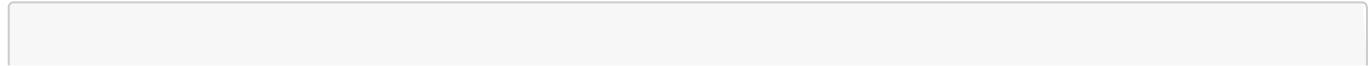
마찬가지로 숫자 유형을 나타내는 데 기본적으로 `GraphQLFloat`가 사용됩니다. `GraphQLInt`을 사용하려면 대신 다음과 같이 `buildSchemaOptions` 객체의 `numberScalarMode`를 '정수'로 설정합니다:

```
GraphQLModule.forRoot({
  buildSchemaOptions: {
    numberScalarMode: '정수',
  }
}),
```

또한 사용자 정의 스칼라를 만들 수도 있습니다.

다. 기본 스칼라 재정의

Date 스칼라에 대한 사용자 정의 구현을 만들려면 새 클래스를 만들면 됩니다.



```
'@nestjs/graphql'에서 { Scalar, CustomScalar }를 임포트하고
, 'graphql'에서 { Kind, ValueNode }를 임포트합니다;

@Injectable('Date', { type: Date })
export class DateScalar 구현 CustomScalar<number, Date> {

  description = '날짜 사용자 정의 스칼라 유형';

  parseValue(value: 문자): Date {
    반환 새로운 날짜(값); // 클라이언트로부터의 값
  }

  serialize(value: Date): number {
    반환 value.getTime(); // 클라이언트로 전송된 값
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      새로운 Date(ast.value)를 반환합니다;
    }
    널을 반환합니다;
  }
}
```

이 준비가 완료되면 `DateScalar`를 공급자로 등록합니다.

```
모듈({
  공급자: [날짜스칼라],
})
내보내기 클래스 CommonModule {}
```

이제 클래스에서 `날짜` 유형을 사용할 수 있습니다.

```
필드() 생성Date: 날짜;
```

사용자 정의 스칼라 가져오기

사용자 정의 스칼라를 사용하려면 해당 스칼라를 가져와 리졸버로 등록합니다. 데모 목적으로 `graphql-type-json` 패키지를 사용하겠습니다. 이 npm 패키지는 `JSON` GraphQL 스칼라 유형을 정의합니다.

패키지를 설치하는 것으로 시작하세요:

```
$ npm i --save graphql-type-json
```

패키지가 설치되면 사용자 정의 리졸버를 `forRoot()` 메서드에 전달합니다:

```
'graphql-type-json'에서 GraphQLJSON import;

@Module({
    imports: [
        GraphQLModule.forRoot({
            resolvers: { JSON: GraphQLJSON },
        }),
    ],
})
내보내기 클래스 AppModule {}
```

이제 클래스에서 JSON 유형을 사용할 수 있습니다.

필드((유형) => GraphQLJSON) 정보:
JSON;

유용한 스칼라를 보려면 [graphql-scalars](#) 패키지를 살펴보세요. 사용자 지정 스

칼라 만들기

사용자 정의 스칼라를 정의하려면 새 `GraphQLScalarType` 인스턴스를 만듭니다. 사용자 정의 `UUID` 스칼라를 생성하겠습니다.

```
함수 validate(uuid: unknown): string | never {
  if (typeof uuid !== "string" || !regex.test(uuid)) {
    throw new Error("유효하지 않은 uuid");
  }
  반환 UUID;
}
```

```
export const CustomUuidScalar = new GraphQLScalarType({
  name: 'UUID',
  description: '간단한 UUID 구문 분석기',
  serialize: (value) => validate(value),
  parseValue: (value) => validate(value),
  parseLiteral: (ast) => validate(ast.value)
})
```

사용자 정의 리졸버를 `forRoot()` 메서드에 전달합니다:

```
모듈({ import:  
[  
  GraphQLModule.forRoot({  
    리졸버: { UUID: CustomUuidScalar },
```

```

    },
],
})
내보내기 클래스 AppModule {}

```

이제 클래스에서 **UUID** 유형을 사용할 수 있습니다.

```

@Field((type) => CustomUuidScalar)
uuid: 문자열;

```

스키마 우선

사용자 정의 스칼라를 정의하려면(스칼라에 대한 자세한 내용은 [여기를 참조하세요](#)) 유형 정의와 전용 리졸버를 생성합니다. 여기서는 (공식 문서에서와 마찬가지로) 데모 목적으로 **graphql-type-json** 패키지를 사용하겠습니다. 이 npm 패키지는 **JSON** GraphQL 스칼라 타입을 정의합니다.

패키지를 설치하는 것으로 시작하세요:

```
$ npm i --save graphql-type-json
```

패키지가 설치되면 사용자 정의 리졸버를 **forRoot()** 메서드에 전달합니다:

```

'graphql-type-json'에서 GraphQLJSON import;

@Module({
  임포트합니다: [
    GraphQLModule.forRoot({
      typePaths: ['./**/*.graphql'], 리졸버:
        { JSON: GraphQLJSON },
    }),
  ],
})
내보내기 클래스 AppModule {}

```

이제 유형 정의에 **JSON** 스칼라를 사용할 수 있습니다:

스칼라 JSON 유

```
형 Foo {  
  필드입니다: JSON  
}
```

스칼라 유형을 정의하는 또 다른 방법은 간단한 클래스를 만드는 것입니다. `Date` 타입으로 스키마를 개선하고 싶다고 가정해 보겠습니다.

```
'@nestjs/graphql'에서 { Scalar, CustomScalar }를 임포트하고
, 'graphql'에서 { Kind, ValueNode }를 임포트합니다;

@Scalar('Date')
export class DateScalar 구현 CustomScalar<number, Date> {

  description = '날짜 사용자 정의 스칼라 유형';

  parseValue(value: 숫자): Date {
    반환 새로운 날짜(값); // 클라이언트로부터의 값
  }

  serialize(value: Date): number {
    반환 value.getTime(); // 클라이언트로 전송된 값
  }

  parseLiteral(ast: ValueNode): Date {
    if (ast.kind === Kind.INT) {
      새로운 Date(ast.value)를 반환합니다;
    }
    널을 반환합니다;
  }
}
```

이 준비가 완료되면 `DateScalar`를 공급자로 등록합니다.

```
모듈({
  공급자: [날짜스칼라],
})
내보내기 클래스 CommonModule {}
```

이제 유형 정의에서 `Date` 스칼라를 사용할 수 있습니다.

스칼라 날짜

기본적으로 모든 스칼라에 대해 생성되는 타입스크립트 정의는 [임의의](#) 것으로, 특별히 타입 안전하지 않습니다. 그러나 유형 생성 방법을 지정할 때 Nest가 사용자 정의 스칼라에 대한 유형을 생성하는 방법을 구성할 수 있습니다:

'@nestjs/graphql'에서 { GraphQLDefinitionsFactory }를 가져오고,
'path'에서 { join }를 가져옵니다;

```
const definitionsFactory = 새로운 GraphQLDefinitionsFactory();
```

```
definitionsFactory.generate({
  typePaths: ['./src/**/*.graphql'],
  경로: join(process.cwd(), 'src/graphql.ts'),
  outputAs: 'class',
  기본 스칼라 타입: '알 수 없음',
  customScalarTypeMapping: {
    DateTime: 'Date',
    BigNumber: '_BigNumber',
  },
  추가 헤더: "'bignumber.js'에서 _BigNumber 가져오기",
});
```

정보 힌트 또는 유형 참조를 대신 사용할 수도 있습니다: `DateTime: Date`. 이 경우

`GraphQLDefinitionsFactory`는 지정된 유형(`Date.name`)의 이름 속성을 추출하여 TS 정의를 생성합니다. 참고: 기본 제공 유형이 아닌 유형(사용자 정의 유형)에 대해서는 가져오기 문을 추가해야 합니다.

이제 다음과 같은 GraphQL 사용자 정의 스칼라 유형이 주어집니다:

```
스칼라 날짜/시간 스칼라  
라 빅넘버 스칼라 페이  
로드
```

이제 다음과 같이 생성된 타입스크립트 정의가 `src/graphql.ts`에 표시됩니다:

```
'bignumber.js'에서 _BigNumber를 임포트하고,  
내보내기 유형 DateTime = 날짜를 내보냅니다;  
내보내기 유형 BigNumber = _BigNumber; 내보내  
기 유형 페이로드 = 알 수 없음;
```

여기서는 `customScalarTypeMapping` 프로퍼티를 사용하여 사용자 정의 스칼라에 대해 선언하려는 유형의 맵을 제공했습니다. 또한 이러한 유형 정의에 필요한 임포트를 추가할 수 있도록 `additionalHeader` 프로퍼티를 제공했습니다. 마지막으로, `customScalarTypeMapping`에 지정되지 않은 모든 사용자 정의 스칼라가 아무 것도 아닌 알 수 없음으로 별칭이 지정되도록 `defaultScalarType`을 '`unknown`'으로 추가했습니다 (`TypeScript`은 유형 안전성 강화를 위해 3.0부터 사용을 권장하고 있습니다).

정보 힌트 [순환형 참조를 피하기 위해 bignumber.js에서 _BigNumber를 가져왔음을 참고하세요.](#)

지시어

지시어는 필드 또는 조각 포함에 첨부할 수 있으며 서버가 원하는 방식으로 쿼리 실행에 영향을 줄 수 있습니다(자세한 내용은 [여기](#)를 참조하세요). GraphQL 사양은 몇 가지 기본 지시어를 제공합니다:

- `include(if: 부울)` - 인수가 참인 경우에만 이 필드를 결과에 포함시킵니다.
- `@skip(if: 부울)` - 인수가 참이면 이 필드를 건너뜁니다.
- `사용 중단됨(이유: 문자열)` - 메시지와 함께 필드를 사용 중단된 것으로 표시합니다.

지시어는 `@` 문자가 앞에 오는 식별자이며, 선택적으로 GraphQL 쿼리 및 스키마 언어의 거의 모든 요소 뒤에 나타날 수 있는 명명된 인수 목록이 뒤따릅니다.

사용자 지정 지시문

아폴로/머큐리우스가 지시문을 만나면 어떤 일이 일어나야 하는지 지시하려면 트랜스포머 함수를 만들 수 있습니다. 이 함수는 `mapSchema` 함수를 사용하여 스키마의 위치(필드 정의, 유형 정의 등)를 반복하고 해당 변환을 수행합니다.

'@graphql-tools/utils'에서 { getDirective, MapperKind, mapSchema }
를 가져옵니다;
'graphql'에서 { defaultFieldResolver, GraphQLSchema }를 가져옵니다;

내보내기 함수 upperDirectiveTransformer(스키마:

```
GraphQLSchema,  
지시어 이름: 문자열,  
) {  
    return mapSchema(schema, {  
        [MapperKind.OBJECT_FIELD]: (fieldConfig) => {  
            const upperDirective = getDirective(  
                스키마,  
                fieldConfig, 지시어  
                이름,  
            )?.[0];  
  
            if (upperDirective) {  
                const { resolve = defaultFieldResolver } = fieldConfig;  
  
                // 원래 리졸버를 *먼저* 다음과 같은 함수로 바꿉니다.  
            }  
        }  
    });  
},
```

통화

```
// 원래 리졸버를 호출한 다음 결과를 대문자로 변환합니다. fieldConfig.resolve =  
async 함수(source, args, context, info).  
{  
    const result = await resolve(source, args, context, info);  
    if (typeof result === 'string') {  
        결과값을 반환합니다;  
    }  
    결과를 반환합니다;  
};  
필드 컨피그를 반환합니다;  
},  
},
```

```
    });
}
```

이제 `GraphQLModule#forRoot`에서 상위 지시어 트랜스포머 변환 함수를 적용합니다.

메서드를 사용하여 **변환 스키마** 함수를 사용합니다:

```
GraphQLModule.forRoot({
  // ...
  transformSchema: (스키마) => upperDirectiveTransformer(스키마, 'upper'),
});
```

등록이 완료되면 `@upper` 지시문을 스키마에서 사용할 수 있습니다. 그러나 지시문을 적용하는 방식은 사용하는 접근 방식(코드 우선 또는 스키마 우선)에 따라 달라집니다.

코드 우선

코드 우선 접근 방식에서는 `@Directive()` 데코레이터를 사용하여 지시문을 적용합니다.

```
@디렉티브('코드') @필드()
제목: 문자열;
```

정보 힌트 `@Directive()` 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

지시어는 필드, 필드 해석기, 입력 및 객체 유형은 물론 쿼리, 변이 및 구독에도 적용할 수 있습니다. 다음은 쿼리 핸들러 수준에서 적용된 지시문의 예입니다:

```
지시어('@deprecated(reason: "이 쿼리는 다음 버전에서 제거될 예정입니다.")')
쿼리(반환값 => 작성자, { 이름: '작성자' })
async getAuthor(@Args({ name: 'id', type: () => Int }) id: number) {
  return this.authorsService.findOneById(id);
}
```

경고 `@Directive()` 데코레이터를 통해 적용된 지시문은 생성된 스키마 정의 파일에 반영되지 않습니다.

마지막으로 `GraphQLModule`에서 다음과 같이 지시문을 선언해야 합니다:

```
GraphQLModule.forRoot({
  // ...
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
  buildSchemaOptions: {
    지시어: [
```

```
새로운 그레프QL디렉티브({ 이
  름: 'upper',
  위치: [DirectiveLocation.FIELD_DEFINITION],
},
],
},
}) ,
```

정보 힌트 그레프QL디렉티브와 [디렉티브](#) 위치는 모두 [그레프큐엘에서](#) 내보낸다.

패키지입니다.

스키마 우선

스키마 우선 접근 방식에서는 SDL에서 직접 지시문을 적용합니다.

지시문 @upper는 FIELD_DEFINITION 유형

```
Post {
  id: Int!
  제목: String! 상위 투표:
  Int
}
```

인터페이스

많은 탑 시스템과 마찬가지로 GraphQL은 인터페이스를 지원합니다. 인터페이스는 유형이 인터페이스를 구현하기 위해 포함해야 하는 특정 필드 집합을 포함하는 추상 유형입니다(자세한 내용은 [여기](#)를 참조하세요).

코드 우선

코드 우선 접근 방식을 사용하는 경우, `@nestjs/graphql`에서 내보낸 `@InterfaceType()` 데코레이터로 주석이 달린 추상 클래스를 생성하여 GraphQL 인터페이스를 정의합니다.

```
'@nestjs/graphql'에서 { 필드, ID, 인터페이스 유형 } 임포트; @인터페이스 유형

()

내보내기 추상 클래스 Character {
  @Field((type) => ID)
  ID: 문자열;

  @Field() 이름:
  문자열;
}
```

경고 경고 TypeScript 인터페이스는 GraphQL 인터페이스를 정의하는 데 사용할 수 없습니다.

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
인터페이스 Character {
  id: ID!
  이름: 문자열!
}
```

이제 `캐릭터` 인터페이스를 구현하려면 `구현` 키를 사용합니다:

```
객체 유형({
  구현합니다: () => [문자],
})

내보내기 클래스 Human 구현 문자 { id: 문자열;
  이름: 문자열;
}
```

정보 힌트 `@ObjectType()` 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

라이브러리에서 생성된 기본 `resolveType()` 함수는 리졸버 메서드에서 반환된 값을 기반으로 유형을 추출합니다. 즉, 클래스 인스턴스를 반환해야 합니다(리터럴 JavaScript 객체를 반환할 수 없음).

사용자 정의된 `resolveType()` 함수를 제공하려면 다음과 같이 `@InterfaceType()` 데코레이터에 전달된 옵션 객체에 `resolveType` 속성을 전달합니다:

```
@InterfaceType({
  resolveType(book) {
    if (book.colors) { 컬러
      링북을 반환합니다;
    }
    교과서를 반환합니다;
  },
})
내보내기 추상 클래스 Book { @Field((type)
  => ID)
  ID: 문자열;

  @Field()
  title: 문자열;
}
```

인터페이스 리졸버

지금까지는 인터페이스를 사용하여 필드 정의만 객체와 공유할 수 있었습니다. 실제 필드 리졸버 구현도 공유하려면 다음과 같이 전용 인터페이스 리졸버를 생성하면 됩니다:

```
'@nestjs/graphql'에서 { Resolver, ResolveField, Parent, Info }를 가져옵니다;

@Resolver(type => Character) // 알림: Character는 인터페이스 내보내기 클래스
CharacterInterfaceResolver {
  @ResolveField(() => [문자]) 친구(
    부모() 문자, // 문자를 구현하는 확인된 객체 @정보() { 부모 유형 }, // 구현하는 객체의 유형
    캐릭터
    @Args('search', { type: () => String }) searchTerm: 문자열,
  ) {
    // 캐릭터의 친구 반환 []을 가져옵니다;
  }
}
```

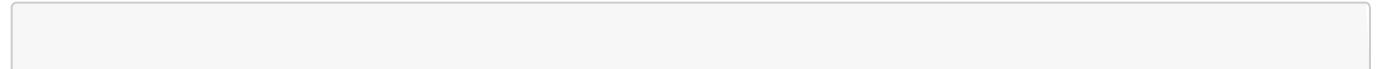
이제 친구 필드 리졸버는 `Character`를 구현하는 모든 객체 유형에 대해 자동으로 등록됩니다.

인터페이스

입니다. 스키

마 우선

스키마 우선 접근 방식으로 인터페이스를 정의하려면 SDL로 GraphQL 인터페이스를 생성하기만 하면 됩니다.



```
인터페이스 Character {  
    id: ID!  
    이름: 문자열!  
}
```

그런 다음 [빠른 시작](#) 챕터에 표시된 대로 타이핑 생성 기능을 사용하여 해당 타입스크립트 정의를 생성할 수 있습니다:

```
내보내기 인터페이스 Character {  
    id: 문자열;  
    이름: 문자열;  
}
```

인터페이스는 리졸버 맵에 추가 `_resolveType` 필드를 추가하여 인터페이스가 어떤 유형으로 해석할지 결정해야 합니다. `CharactersResolver` 클래스를 생성하고 다음과 같이 정의해 보겠습니다. `_resolveType` 메서드를 정의해 보겠습니다:

```
@Resolver('Character')  
내보내기 클래스 CharactersResolver {  
    @ResolveField()  
    _resolveType(value) {  
        if ('age' in value) {  
            사람을 반환합니다;  
        }  
        널을 반환합니다;  
    }  
}
```

정보 힌트 모든 데코레이터는 `@nestjs/graphql` 패키지에서 내보냅니다.

유니온

유니온 유형은 인터페이스와 매우 유사하지만 유형 간에 공통 필드를 지정할 수 없습니다(자세한 내용은 [여기](#)를 참조하세요). 유니온은 단일 필드에서 분리된 데이터 유형을 반환하는 데 유용합니다.

코드 우선

GraphQL 유니온 유형을 정의하려면 이 유니온이 구성될 클래스를 정의해야 합니다. Apollo 설명서의 [예제에](#) 따라 두 개의 클래스를 만들겠습니다. 첫째, `Book`:

```
'@nestjs/graphql'에서 { Field, ObjectType } 임포트; @ObjectType()
내보내기 클래스 Book {
  @Field()
  제목: 문자열;
}
```

그런 다음 작성합니다:

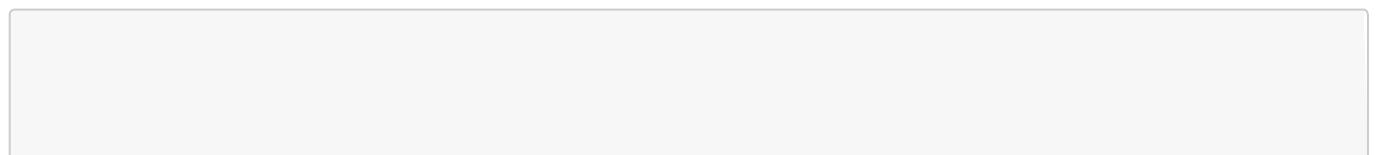
```
'@nestjs/graphql'에서 { Field, ObjectType } 임포트; @ObjectType()
내보내기 클래스 Author {
  @Field()
  이름: 문자열;
}
```

이 준비가 완료되면 `@nestjs/graphql` 패키지에서 내보낸 `createUnionType` 함수를 사용하여 `ResultUnion` 유니온을 등록합니다:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  유형: () => [저자, 책] const,
});
```

경고 `createUnionType` 함수의 `types` 프로퍼티가 반환하는 배열에 `const` 어설션을 지정해야 합니다. `@Query(returns => [ResultUnion])` 어설션을 제공하지 않으면 컴파일 시 잘못된 선언 파일이 생성되어 다른 프로젝트에서 사용할 때 오류가 발생합니다.

이제 쿼리에서 [ResultUnion](#)을 참조할 수 있습니다:



```
반환 [새로운 저자(), 새로운 책()];
}
```

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
유형 저자 { 이름: 문
  자열!
}

유형 Book {
  title: 문자열!
}

유니온 ResultUnion = 저자 | 도서 유형 퀘
리 {
  검색: [ResultUnion!]!
}
```

라이브러리에서 생성된 기본 `resolveType()` 함수는 리졸버 메서드에서 반환된 값을 기반으로 유형을 추출합니다. 즉, 리터럴 JavaScript 객체 대신 클래스 인스턴스를 반환해야 합니다.

사용자 정의된 `resolveType()` 함수를 제공하려면 다음과 같이 `createUnionType()` 함수에 전달된 옵션 객체에 `resolveType` 속성을 전달합니다:

```
export const ResultUnion = createUnionType({
  name: 'ResultUnion',
  types: () => [Author, Book] as const,
  resolveType(value) {
    if (value.name) {
      Author를 반환합니다;
    }
    if (value.title) { 반환
      책;
    }
    널을 반환합니다;
  },
});
```

스키마 유형 저자 { 이름: 문
자열!

스키마 우선 접근 방식에서 유니온을 정의하려면 SDL을 사용하여 GraphQL 유니온을 생성하기만 하면 됩니다.

```
}
```

```
유형 Book {
    title: 문자열!
}
```

```
유니온 ResultUnion = 저자 | 책
```

그런 다음 [빠른 시작](#) 챕터에 표시된 대로 타이핑 생성 기능을 사용하여 해당 타입스크립트 정의를 생성할 수 있습니다:

```
내보내기 클래스 Author { 이름
    : 문자열;
}
```

```
내보내기 클래스 Book { 제
    목: 문자열;
}
```

```
내보내기 유형 ResultUnion = 저자 | 책;
```

유니온에는 리졸버 맵에 추가 _____ `_resolveType` 필드를 추가하여 유니온이 어떤 유형으로 해석할지 결정해야 합니다. 또한 `ResultUnionResolver` 클래스는 모든 모듈에서 프로바이더로 등록해야 합니다. `ResultUnionResolver` 클래스를 생성하고 `_resolveType` 메서드를 정의해 보겠습니다.

```
@Resolver('ResultUnion')
export 클래스 ResultUnionResolver {
    @ResolveField()
    _resolveType(value) {
        if (value.name) {
            '저자'를 반환합니다;
        }
        if (value.title) {
            '책'을 반환합니다;
        }
        널을 반환합니다;
    }
}
```

정보 힌트 모든 데코레이터는 [@nestjs/graphql](#) 패키지에서 내보냅니다.

열거형

열거형은 허용되는 특정 값 집합으로 제한되는 특별한 종류의 스칼라입니다(자세한 내용은 [여기](#)를 참조하세요).

이를 통해 다음을 수행할 수 있습니다:

- 이 유형의 인수가 허용된 값 중 하나인지 확인합니다.

- 필드가 항상 유한한 값 집합 중 하나라는 것을 유형 시스템을 통해 전달합니다. 먼저 코드화합니다

코드 우선 접근 방식을 사용하는 경우, 간단히 TypeScript를 생성하여 GraphQL 열거형 유형을 정의합니다. 열거형.

```
export enum AllowedColor {  
    RED,  
    녹색,  
    파란색,  
}
```

이 작업을 완료한 후 [@nestjs/graphql](#) 패키지에서 내보낸 `registerEnumType` 함수를 사용하여 `AllowedColor` 열거형을 등록합니다:

```
registerEnumType(AllowedColor, {  
    name: 'AllowedColor',  
});
```

이제 유형에서 허용된 색상을 참조할 수 있습니다:

```
@Field(유형 => 허용된 색상)  
favoriteColor: 허용된 색상;
```

이렇게 하면 SDL에서 GraphQL 스키마의 다음 부분이 생성됩니다:

```
열거형 허용된 색상 { 빨간  
    색  
    녹색  
    파란색  
}
```

열거형에 대한 설명을 제공하려면 `설명` 프로퍼티를 `registerEnumType()`에 전달합니다. 함수입니다.

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  설명: '지원되는 색상입니다.',
});
```

열거형 값에 대한 설명을 제공하거나 값을 더 이상 사용되지 않는 것으로 표시하려면 [값맵](#)을 전달합니다.

속성을 다음과 같이 변경합니다:

```
registerEnumType(AllowedColor, {
  name: 'AllowedColor',
  설명: '지원되는 색상입니다.',
  valuesMap: {
    RED: {
      설명: '기본 색상입니다.',
    },
    BLUE: {
      depreciationReason: '너무 파랗다.',
    },
  },
});
```

이렇게 하면 SDL에 다음과 같은 GraphQL 스키마가 생성됩니다:

```
"""
지원되는 색상입니다.

enum AllowedColor {
  """
  기본 색상입니다.

  레드
  그린
  BLUE @deprecated(이유: "너무 파란색입니다.")
}
```

스키마 우선

스키마 퍼스트 접근 방식에서 열거자를 정의하려면 SDL로 GraphQL 열거자를 생성하기만 하면 됩니다.

```
열거형 허용된 색상 { 빨간
  색
  녹색
  파란색
}
```

그런 다음 [빠른 시작](#) 챕터에 표시된 대로 타이핑 생성 기능을 사용하여 해당 타입스크립트 정의를 생성할 수 있습

니다:

내보내기 열거형 허용된 색상 { 빨

간색

```
  녹색
```

```
  파란색
```

```
}
```

때때로 백엔드에서 공개 API와 내부적으로 열거형에 대해 다른 값을 강제하는 경우가 있습니다. 이 예제에서는 API에 RED가 포함되어 있지만, 리졸버에서는 #f00을 대신 사용할 수 있습니다(자세한 내용은 [여기](#)를 참조하세요). 이렇게 하려면 AllowedColor 열거형에 대한 리졸버 객체를 선언합니다:

```
내보내기 const allowedColorResolver: 레코드<허용된 색의 유형 키, any>
= {
  빨간색: '#f00',
};
```

정보 힌트 모든 데코레이터는 [@nestjs/graphql](#) 패키지에서 내보냅니다.

그런 다음 이 리졸버 객체를 GraphQLModule#forRoot()의 리졸버 프로퍼티와 함께 사용합니다. 메서드에 대해 다음과 같이 설명합니다:

```
GraphQLModule.forRoot({
  resolvers: {
    허용된 색상: 허용된 색상 해결기,
  },
});
```

필드 미들웨어

경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

필드 미들웨어를 사용하면 필드가 확인되기 전이나 후에 임의의 코드를 실행할 수 있습니다. 필드 미들웨어를 사용하여 필드의 결과를 변환하거나, 필드의 인수를 검증하거나, 필드 수준 역할을 확인할 수도 있습니다(예: 미들웨어 함수가 실행되는 대상 필드에 액세스하는 데 필요한 경우).

여러 미들웨어 함수를 필드에 연결할 수 있습니다. 이 경우 이전 미들웨어가 다음 미들웨어를 호출하기로 결정한 체인을 따라 순차적으로 호출됩니다. 미들웨어 배열에 있는 미들웨어 함수의 순서가 중요합니다. 첫 번째 리졸버는 "가장 바깥쪽" 레이어이므로 가장 먼저 그리고 마지막으로 실행됩니다(그래프 쿼리 미들웨어 패키지와 유사하게). 두 번째 리졸버는 "두 번째 외부" 계층이므로 두 번째로 실행되고 두 번째에서 마지막으로 실행됩니다.

시작하기

필드 값을 클라이언트로 다시 보내기 전에 기록하는 간단한 미들웨어를 만드는 것부터 시작해 보겠습니다:

```
'@nestjs/graphql'에서 { FieldMiddleware, MiddlewareContext, NextFn
}을 가져옵니다;

const loggerMiddleware: FieldMiddleware = async (
  ctx: MiddlewareContext,
  다음: NextFn,
) => {
  const value = await next();
  console.log(value);
  반환 값입니다;
};
```

정보 힌트 `MiddlewareContext`는 GraphQL 리졸버 함수가 일반적으로 수신하는 것과 동일한 인수(`{ source, args, context, info }`)로 구성된 객체이며, `NextFn`은 스택의 다음 미들웨어(이 필드에 바인딩된) 또는 실제 필드 리졸버를 실행할 수 있게 해주는 함수입니다. 경고 필드 미들웨어 함수는 매우 가볍게 설계되었으며 데이터베이스에서 데이터를 검색하는 등 잠재적으로 시간이 많이 소요되는 작업을 수행해서는 안 되므로 종속성을 주입하거나 Nest의 DI 컨테이너에 액세스할 수 없습니다. 데이터 소스에서 외부 서비스를 호출하거나 데이터를 쿼리해야 하는 경우, 루트 쿼리/변이 핸들러에 바인딩된 가드/인터셉터에서 이를 수행하고 필드 미들웨어 내에서 액세스할 수 있는 `컨텍스트` 객체에 할당해야 합니다(특히, `MiddlewareContext` 객체에서).

필드 미들웨어는 필드 미들웨어 인터페이스와 일치해야 합니다. 위의 예제에서는 먼저 실제 필드 리졸버를 실행하고 필드 값을 반환하는 `next()` 함수를 실행한 다음, 이 값을 터미널에 로깅합니다. 또한 미들웨어 함수에서 반환된 값은 이전 값을 완전히 재정의하므로 변경을 수행하고 싶지 않으므로 원래 값을 반환하기만 하면 됩니다.

이렇게 하면 다음과 같이 `@Field()` 데코레이터에 미들웨어를 직접 등록할 수 있습니다:

객체 유형()

```
내보내기 클래스 레시피 {
  필드({ 미들웨어: [loggerMiddleware] }) 제목: 문
  자열;
}
```

이제 [레시피](#) 개체 유형의 [제목](#) 필드를 요청할 때마다 원래 필드 값이 콘솔에 기록됩니다.

정보 힌트 [확장](#) 기능을 사용하여 필드 수준 권한 시스템을 구현하는 방법을 알아보려면 이 [섹션을](#) 참조하세요.
경고 필드 미들웨어는 [ObjectType](#) 클래스에만 적용할 수 있습니다. 자세한 내용은 이 [이슈를](#) 확인하세요.

또한 위에서 언급했듯이 미들웨어 함수 내에서 필드 값을 제어할 수 있습니다. 데모를 위해 레시피의 제목(있는 경우)을 대문자로 표시해 보겠습니다:

```
const value = await next();
return value?.toUpperCase();
```

이 경우 요청 시 모든 제목이 자동으로 대문자로 바뀝니다.

마찬가지로 필드 미들웨어를 사용자 지정 필드 리졸버에 바인딩할 수 있습니다([@ResolveField\(\)](#) 데코레이터)를 다음과 같이 변경합니다:

```
@ResolveField(() => String, { middleware: [loggerMiddleware] })
title() {
  '자리 표시자'를 반환합니다;
}
```

경고 필드 리졸버 수준에서 인핸서가 활성화된 경우([자세히 읽기](#)) 필드 미들웨어 기능은 메서드에 바인딩된 인터셉터, 가드 등보다 먼저 실행됩니다(쿼리 또는 변이 처리기에 등록된 루트 수준 인핸서 이후).

글로벌 필드 미들웨어

미들웨어를 특정 필드에 직접 바인딩하는 것 외에도 하나 또는 여러 개의 미들웨어 함수를 전역적으로 등록할 수도 있습니다. 이 경우 개체 유형의 모든 필드에 자동으로 연결됩니다.

```
GraphQLModule.forRoot({
  autoSchemaFile: 'schema.gql',
  buildSchemaOptions: {
    fieldMiddleware: [loggerMiddleware],
```

```
},  
}) ,
```

정보 힌트 전역으로 등록된 필드 미들웨어 함수는 로컬로 등록된 함수(특정 필드에 직접 바인딩된 함수)
보다 먼저 실행됩니다.

매핑된 유형

경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

CRUD(만들기/읽기/업데이트/삭제)와 같은 기능을 구축할 때 기본 엔티티 유형에서 변형을 구성하는 것이 유용할 때가 많습니다. Nest는 유형 변환을 수행하는 여러 유틸리티 함수를 제공하여 이 작업을 더 편리하게 만듭니다.

부분

입력 유효성 검사 유형(데이터 전송 개체 또는 DTO라고도 함)을 구축할 때 동일한 유형에 대해 만들기 및 업데이트 변형을 구축하는 것이 유용한 경우가 많습니다. 예를 들어, 생성 변형은 모든 필드를 필수로 설정하고 업데이트 변형은 모든 필드를 선택 사항으로 설정할 수 있습니다.

Nest는 이 작업을 더 쉽게 수행하고 상용구를 최소화하기 위해 `PartialType()` 유틸리티 함수를 제공합니다.

`PartialType()` 함수는 입력 유형의 모든 속성이 선택 사항으로 설정된 유형(클래스)을 반환합니다. 예를 들어 다음과 같은 `create` 유형이 있다고 가정해 보겠습니다:

입력 유형()

```
CreateUserInput 클래스 {  
    @Field()  
    이메일: 문자열;  
  
    @Field()  
    비밀번호:  
    문자열;  
  
    필드() 이름: 문자열입  
    니다;  
}
```

기본적으로 이러한 필드는 모두 필수입니다. 필드는 동일하지만 각 필드가 선택 사항인 유형을 만들려면 클래스 참조(`CreateUserInput`)를 인수로 전달하는 `PartialType()`을 사용합니다:

입력 유형()

```
export class UpdateUserInput extends PartialType(CreateUserInput) {}
```

정보 힌트 `PartialType()` 함수는 `@nestjs/graphql` 패키지에서 가져온 것입니다.

`PartialType()` 함수는 데코레이터 팩토리에 대한 참조인 선택적 두 번째 인수를 받습니다. 이 인수는 결과(자식) 클래스에 적용되는 데코레이터 함수를 변경하는 데 사용할 수 있습니다. 지정하지 않으면 자식 클래스는 부모 클래스(첫 번째 인수에서 참조된 클래스)와 동일한 데코레이터를 효과적으로 사용합니다. 위 예제에서는 `@InputType()` 데코레이터로 어노테이션된 `CreateUserInput`을 확장하고 있습니다. `UpdateUserInput`도 `@InputType()`으로 장식된 것처럼 취급되기를 원하므로 두 번째 인수로 `InputType`을 전달할 필요가 없습니다. 부모와 자식

유형이 다른 경우(예: 부모가 `@ObjectType`으로 장식된 경우) 두 번째 인수로 `InputType`을 전달합니다. 예를 들어

입력 유형()

```
export class UpdateUserInput extends PartialType(User, InputType) {}
```

선택

`PickType()` 함수는 입력 유형에서 속성 집합을 선택하여 새 유형(클래스)을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

입력 유형()

```
CreateUserInput 클래스 {  
  @Field()  
  이메일: 문자열;  
  
  @Field()  
  비밀번호:  
  문자열;  
  
  필드() 이름: 문자열입  
  니다;  
}
```

이 클래스에서 `PickType()` 유틸리티 함수를 사용하여 프로퍼티 집합을 선택할 수 있습니다:

입력 유형()

```
내보내기 클래스 UpdateEmailInput extends PickType(CreateUserInput, [  
  '이메일,  
  const) {}
```

정보 힌트 `PickType()` 함수는 `@nestjs/graphql` 패키지에서 가져온 것입니다.

생략

`OmitType()` 함수는 입력 유형에서 모든 속성을 선택한 다음 특정 키 집합을 제거하여 유형을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

입력 유형()

```
CreateUserInput 클래스 {
```

```
    @Field()
```

```
        이메일: 문자열;
```

```
    @Field() 비밀번호:
```

```
        문자열;
```

```
필드() 이름: 문자열입
니다;
}
```

아래와 같이 [이메일을](#) 제외한 모든 속성을 가진 파생 유형을 생성할 수 있습니다. 이 구조에서 [OmitType의](#) 두 번째 인수는 속성 이름의 배열입니다.

입력 유형()

```
내보내기 클래스 UpdateUserInput extends OmitType(CreateUserInput, [
  '이메일',
  const) {}
```

정보 힌트 [OmitType\(\)](#) 함수는 [@nestjs/graphql](#) 패키지에서 가져온 것입니다.

교차로

[IntersectionType\(\)](#) 함수는 두 유형을 하나의 새로운 유형(클래스)으로 결합합니다. 예를 들어 다음과 같은 두 가지 유형으로 시작한다고 가정해 보겠습니다:

입력 유형()

```
CreateUserInput 클래스 {
  @Field()
  이메일: 문자열;

  @Field() 비밀번호:
  문자열;
}
```

객체 유형()

```
내보내기 클래스 AdditionalUserInfo {
  @Field()
  이름: 문자열입니다;

  @Field() 성: 문자열
  ;
}
```

두 유형의 모든 속성을 결합한 새로운 유형을 생성할 수 있습니다.

입력 유형()

```
내보내기 클래스 UpdateUserInput extends IntersectionType<
    CreateUserInput,
    추가 사용자 정보,
)> {}
```

정보 힌트 `IntersectionType()` 함수는 `@nestjs/graphql` 패키지에서 가져온 것입니다.

구성

유형 매핑 유ти리티 함수는 컴포저블이 가능합니다. 예를 들어 다음은 `이메일을 제외한 CreateUserInput` 유형의 모든 속성을 가진 유형(클래스)을 생성하며, 이러한 속성은 선택 사항으로 설정됩니다:

입력 유형()

```
내보내기 클래스 UpdateUserInput extends PartialType(  
  OmitType(CreateUserInput, ['email'] as const),  
) {}
```

아폴로 플러그인

플러그인을 사용하면 특정 이벤트에 대한 응답으로 사용자 지정 작업을 수행하여 Apollo Server의 핵심 기능을 확장할 수 있습니다. 현재 이러한 이벤트는 GraphQL 요청 수명 주기의 개별 단계와 Apollo Server 자체의 시작에 해당합니다(자세한 내용은 [여기를 참조하세요](#)). 예를 들어 기본 로깅 플러그인은 Apollo Server로 전송되는 각 요청과 관련된 GraphQL 쿼리 문자열을 로깅할 수 있습니다.

사용자 지정 플러그인

플러그인을 만들려면 `@nestjs/apollo` 패키지에서 내보낸 `@Plugin` 데코레이터로 주석이 달린 클래스를 선언하세요. 또한 코드 자동 완성을 개선하려면 `@apollo/server` 패키지에서 `ApolloServerPlugin` 인터페이스를 구현하세요.

```
'@apollo/server'에서 { ApolloServerPlugin, GraphQLRequestListener }
```

를 가져옵니다;

```
'@nestjs/apollo'에서 { Plugin }을 가져옵니다;
```

플러그인()

로깅 플러그인 내보내기 클래스 `ApolloServerPlugin` 구현 {

```
async requestDidStart(): Promise<GraphQLRequestListener<any>> {
  console.log('요청 시작');

  반환 {
    async willSendResponse() { console.log('응
      답을 보냅니다');

    },
  };
}
```

이 작업을 완료하면 `로깅 플러그인을` 공급자로 등록할 수 있습니다.

```
모듈({
```

제공자: [로깅 플러그인],

```
})
```

```
내보내기 클래스 CommonModule {}
```

Nest는 플러그인을 자동으로 인스턴스화하여 아폴로 서버에 적용합니다. 외부 플

러그인 사용

기본으로 제공되는 플러그인은 여러 가지가 있습니다. 기존 플러그인을 사용하려면 해당 플러그인을 가져와서 다음 위치에 추가하면 됩니다.

플러그인 배열:

```
GraphQLModule.forRoot({  
  // ...
```

```
    플러그인: [ApolloServerOperationRegistry({ /* 옵션 */})]  
},
```

정보 힌트 [ApolloServerOperationRegistry](#) 플러그인은 [@apollo/server](#)- 플러그인-운영-레지스트리 패키지에서 내보내집니다.

머큐리우스 플러그인

기존 머큐리우스 전용 Fastify 플러그인 중 일부는 플러그인 트리에서 머큐리우스 플러그인([여기에서](#) 자세히 읽어보세요) 다음에 로드해야 합니다.

경고 경고 머큐리업로드는 예외이므로 메인 파일에 등록해야 합니다.

이를 위해 머큐리우스 드라이버는 선택적 플러그인 구성 옵션을 노출합니다. 이는 플러그인과 옵션이라는 두 가지 속성으로 구성된 객체 배열을 나타냅니다. 따라서 [캐시 플러그인을](#) 등록하면 다음과 같이 표시됩니다:

```
GraphQLModule.forRoot({  
  driver: MercuriusDriver,  
  // ... 플러그인:  
  plugin: [  
    {  
      플러그인: 캐시, 옵션: {  
        ttl: 10,  
        policy: {  
          쿼리: { add:  
            true  
          }  
        }  
      },  
    }  
  ]  
}),
```

복잡성

경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

쿼리 복잡도를 사용하면 특정 필드의 복잡도를 정의하고 최대 복잡도를 가진 쿼리를 제한할 수 있습니다. 간단한 숫자를 사용하여 각 필드의 복잡도를 정의하는 것이 좋습니다. 일반적인 기본값은 각 필드에 복잡도 1을 지정하는 것입니다. 또한 복잡도 추정기를 사용하여 GraphQL 쿼리의 복잡도 계산을 사용자 지정할 수 있습니다. 복잡성 추정기는 필드의 복잡성을 계산하는 간단한 함수입니다. 규칙에 복잡도 추정기를 원하는 수만큼 추가한 다음 차례로 실행할 수 있습니다. 숫자 볍잡도 값을 반환하는 첫 번째 추정기가 해당 필드의 복잡도를 결정합니다.

`nestjs/graphql` 패키지는 비용 분석 기반 솔루션을 제공하는 [그래프 쿼리 복잡성](#) 같은 도구와 매우 잘 통합됩니다. 이 라이브러리를 사용하면 실행 비용이 너무 많이 드는 것으로 간주되는 GraphQL 서버에 대한 쿼리를 거부할 수 있습니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
$ npm install --save graphql-query-complexity
```

시작하기

설치 프로세스가 완료되면 `ComplexityPlugin` 클래스를 정의할 수 있습니다:

"@nestjs/graphql"에서 { GraphQLSchemaHost }를 가져오고,

"@nestjs/apollo"에서 { Plugin }을 가져옵니다;

가져 오기 {

 ApolloServerPlugin,
 GraphQLRequestListener,

 'apollo-server-plugin-base'에서 } import

 { GraphQLError } from 'graphql'; import

 {

 fieldExtensionsEstimator,
 getComplexity,
 simpleEstimator,

 그리프 쿼리 복잡도'에서 }를 가져옵니다;

플러그인()

```
export class ComplexityPlugin 구현 ApolloServerPlugin { constructor(private
```

```
gqlSchemaHost: GraphQLSchemaHost) {}
```

```
async requestDidStart(): Promise<GraphQLRequestListener> {
```

```
    const maxComplexity = 20;
```

```
    const { 스키마 } = this.gqlSchemaHost;
```

반환 {

```

async didResolveOperation({ request, document }) {
  const complexity = getComplexity({
    스키마,
    작업 이름: 요청.작업 이름, 쿼리: 문서,
    변수: 요청.변수, 추정자: [
      fieldExtensionsEstimator(),
      simpleEstimator({ defaultComplexity: 1 }),
    ],
  });
  if (complexity > maxComplexity) {
    throw new GraphQLError(
      '쿼리가 너무 복잡합니다: ${complexity}. 허용되는 최대 복잡도: ${maxComplexity}',
    );
  }
  console.log('쿼리 복잡성:', 복잡성);
},
};

}
}

```

데모 목적으로 허용되는 최대 복잡도를 20으로 지정했습니다. 위의 예제에서는 단순 추정기와 필드 확장 추정기라는 두 가지 추정기를 사용했습니다.

- `simpleEstimator`: 단순 추정기는 각 필드에 대해 고정된 복잡도를 반환합니다.
- `fieldExtensionsEstimator`: 필드 확장 추정기는 스키마의 각 필드에 대한 복잡도 값을 추출합니다.

정보 힌트 모든 모듈의 공급자 배열에 이 클래스를 추가하는 것을 잊지 마세요.

필드 수준의 복잡성

이 플러그인을 사용하면 이제 복잡도를 지정하여 모든 필드에 대한 볍잡도를 정의할 수 있습니다.

프로퍼티를 `@Field()` 데코레이터에 전달하면 다음과 같이 됩니다:

```

@Field({ complexity: 3 })
title: 문자열;

```

또는 추정기 함수를 정의할 수도 있습니다:

```

@Field({ complexity: (options: ComplexityEstimatorArgs) => ... })
title: 문자열;

```

쿼리/변이 수준 복잡성

또한 `@Query()` 및 `@Mutation()` 데코레이터에는 다음과 같이 지정된 **복잡성** 속성이 있을 수 있습니다:

```
쿼리({ 복잡성: (옵션: ComplexityEstimatorArgs) => options.args.count
* options.childComplexity }) 항목(@Args('count') count: 숫자) {
  return this.itemsService.getItems({ count });
}
```

확장 기능

경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

확장은 유형 구성에서 임의의 데이터를 정의할 수 있는 고급 저수준 기능입니다. 특정 필드에 사용자 지정 메타데이터를 첨부하면 보다 정교하고 일반적인 솔루션을 만들 수 있습니다. 예를 들어, 확장 기능을 사용하면 특정 필드에 액세스하는 데 필요한 필드 수준 역할을 정의할 수 있습니다. 이러한 역할은 런타임에 반영되어 호출자가 특정 필드를 검색할 수 있는 충분한 권한을 가지고 있는지 여부를 결정할 수 있습니다.

사용자 지정 메타데이터 추가

필드에 대한 사용자 지정 메타데이터를 첨부하려면 `@Extensions()` 데코레이터를 내보낸 후 `nestjs/graphql` 패키지.

```
@Field()  
확장 기능({ 역할: 역할.관리자 }) 비밀번호: 문  
자열;
```

위의 예에서는 `역할` 메타데이터 속성에 `Role.ADMIN` 값을 할당했습니다. `Role`은 시스템에서 사용 가능한 모든 사용자 역할을 그룹화하는 간단한 TypeScript 열거형입니다.

필드에 메타데이터를 설정하는 것 외에도 클래스 수준 및 메서드 수준(예: 쿼리 핸들러)에서 `@Extensions()` 데코레이터를 사용할 수 있다는 점에 유의하세요.

사용자 지정 메타데이터 사용

사용자 정의 메타데이터를 활용하는 로직은 필요에 따라 얼마든지 복잡해질 수 있습니다. 예를 들어 메서드 호출당 이벤트를 저장/기록하는 간단한 인터셉터 또는 필드를 검색하는 데 필요한 역할을 호출자 권한(필드 수준 권한 시스템)과 일치시키는 [필드 미들웨어](#)를 만들 수 있습니다.

예시를 위해 사용자의 역할(여기에 하드코딩됨)을 대상 필드에 액세스하는 데 필요한 역할과 비교하는 `checkRoleMiddleware`를 정의해 보겠습니다:

```
export const checkRoleMiddleware: FieldMiddleware = async (
  ctx: 미들웨어 컨텍스트,
  다음: 다음Fn,
) => {
  const { 정보 } = CTX;
  const { extensions } = info.parentType.getFields()[info.fieldName];

  /**
   * 실제 애플리케이션에서 "userRole" 변수
   * 은 호출자(사용자)의 역할을 나타내야 합니다(예: "ctx.user.role").
   */
  const UserRole = Role.USER;
  if (UserRole === extensions.role) {
```

```
// 또는 그냥 "널을 반환"하여 무시할 수 있습니다

    ` 사용자에게 "${info.fieldName}" 필드에 액세스할 수 있는 충분한 권한이
    없습니다. `,
);

}

다음()을 반환합니다;

};
```

이 준비가 완료되면 다음과 같이 **비밀번호** 필드에 대한 미들웨어를 등록할 수 있습니다:

```
@Field({ 미들웨어: [checkRoleMiddleware] })
@Extensions({ 역할: Role.ADMIN })
비밀번호: 문자열;
```

CLI 플러그인

경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

TypeScript의 메타데이터 반영 시스템에는 몇 가지 제약이 있어 클래스가 어떤 프로퍼티로 구성되어 있는지 확인하거나 특정 프로퍼티가 선택 사항인지 필수 사항인지 인식할 수 없습니다. 하지만 이러한 제약 조건 중 일부는 컴파일 시점에 해결할 수 있습니다. Nest는 필요한 상용구 코드의 양을 줄이기 위해 TypeScript 컴파일 프로세스를 개선하는 플러그인을 제공합니다.

정보 힌트 이 플러그인은 옵트인입니다. 원하는 경우 모든 데코레이터를 수동으로 선언하거나 필요한 곳에 특정 데코레이터만 선언할 수 있습니다.

개요

GraphQL 플러그인이 자동으로 실행됩니다:

- HideField를 사용하지 않는 한 모든 입력 객체, 객체 유형 및 인수 클래스 속성에 `@Field`을 주석으로 추가 합니다.
- 물음표에 따라 `nullable` 가능 속성을 설정합니다(예: `이름?: 문자열!` 설정됩니다).
무효화 가능: 참
- 유형에 따라 `유형` 속성을 설정합니다(배열도 지원).
- 댓글을 기반으로 프로퍼티에 대한 설명을 생성합니다(`introspectComments`가 `true`로 설정된 경우).

플러그인에서 분석하려면 파일 이름에 다음 접미사 중 하나가 포함되어야 한다는 점에 유의하세요:

`['.input.ts', '.args.ts', '.entity.ts', '.model.ts']`(예: `author.entity.ts`). 다른 접미사를 사용하는 경우 `typeFileNameSuffix` 옵션을 지정하여 플러그인의 동작을 조정할 수 있습니다(아래 참조).

지금까지 배운 내용을 바탕으로 GraphQL에서 유형을 어떻게 선언해야 하는지 패키지에 알리기 위해 많은 코드를 복제해야 합니다. 예를 들어 간단한 `Author` 클래스를 다음과 같이 정의할 수 있습니다:

```
@@파일명(authors/models/author.model) @ObjectType()
export class Author {
  @Field(type => ID)
  id: number;

  @Field({ nullable: true })
  firstName?: 문자열;

  @Field({ nullable: true })
  lastName?: 문자열;

  필드(유형 => [게시물]) 게시물
  : Post[];
}
```

중간 규모의 프로젝트에서는 큰 문제가 되지 않지만, 클래스의 수가 많아지면 장황해지고 유지 관리가 어려워집니다.

GraphQL 플러그인을 활성화하면 위의 클래스 정의를 간단하게 선언할 수 있습니다:

```
@@파일명(authors/models/author.model) @ObjectType()
export class Author {
  @Field(type => ID)
  id: 숫자;
  firstName?: 문자열;
  lastName?: 문자열;
  posts: Post[];
}
```

플러그인은 추상 구문 트리를 기반으로 적절한 데코레이터를 즉석에서 추가합니다. 따라서 코드 곳곳에 흩어져 있는 `@Field` 데코레이터로 고생할 필요가 없습니다.

정보 힌트 플러그인은 누락된 GraphQL 프로퍼티를 자동으로 생성하지만, 재정의해야 하는 경우

`@Field()`를 통해 명시적으로 설정하기만 하면 됩니다.

댓글 성찰

댓글 인트로스펙션 기능을 활성화하면 CLI 플러그인이 댓글을 기반으로 필드에 대한 설명을 생성합니다.

예를 들어 `역할` 속성을 예로 들어 보겠습니다:

```
/**
 * 사용자 역할 목록
 */
@Field(() => [문자열], {
  설명: '사용자 역할 목록'
})
역할: 문자열[];
```

설명 값을 복제해야 합니다. `introspectComments`를 활성화하면 CLI 플러그인이 이러한 설명을 추출하여 속성에 대한 설명을 자동으로 제공할 수 있습니다. 이제 위의 필드는 다음과 같이 간단하게 선언할 수 있습니다:

```
/**
 * 사용자 역할 목록
 */
역할: 문자열[];
```

CLI 플러그인 사용

플러그인을 활성화하려면 `nest-cli.json`(Nest CLI를 사용하는 경우)을 열고 다음 플러그인을 추가합니다.

구성합니다:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src", "컴파일러옵션": {
    "plugins": ["@nestjs/graphql"]
  }
}
```

옵션 속성을 사용하여 플러그인의 동작을 사용자 정의할 수 있습니다.

```
"플러그인": [
  {
    "name": "@nestjs/graphql", "옵션": {
      "typeFileNameSuffix": [".input.ts", ".args.ts"],
      "introspectComments": true
    }
  }
]
```

옵션 속성은 다음 인터페이스를 충족해야 합니다:

```
export interface PluginOptions {
  typeFileNameSuffix?: string[];
  introspectComments?: boolean;
}
```

옵션	기본값	설명
typeFileName접미사	['.input.ts', '.args.ts', '.entity.ts', '.model.ts']	GraphQL 유형 파일 접미사
introspectComments 거짓	true	true로 설정하면 플러그인이 다음을 생성합니다. 댓글을 기반으로 한 속성 설명

CLI를 사용하지 않고 대신 사용자 정의 웹팩 구성성을 사용하는 경우 이 플러그인을 `ts-loader`와 함께 사용할 수 있습니다:

```
getCustomTransformers: (program: any) => ({{  
    이전: [require('@nestjs/graphql/plugin').before({}, program)]  
}},
```

SWC 빌더

표준 설정(비모노레포)의 경우 SWC 빌더에서 CLI 플러그인을 사용하려면 [여기에](#) 설명된 대로 유형 검사를 사용하도록 설정해야 합니다.

```
nest start -b swc --type-check
```

모노레포 설정의 경우 [여기](#) 지침을 따르세요.

```
$ npx ts-node src/generate-metadata.ts  
# 또는 npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

이제 아래와 같이 직렬화된 메타데이터 파일을 [GraphQLModule](#) 메서드를 통해 로드해야 합니다:

```
'./metadata'에서 메타데이터 가져오기; // <-- "플러그인 메타데이터 생성기"에 의해 자동 생성된  
파일입니다.  
  
GraphQLModule.forRoot<...>({  
  ... // 기타 옵션 메타데이터  
  '  
}),
```

[ts-jest](#)와 통합(e2e 테스트)

이 플러그인을 활성화한 상태에서 e2e 테스트를 실행할 때 스키마 컴파일과 관련된 문제가 발생할 수 있습니다. 예를 들어 가장 일반적인 오류 중 하나는 다음과 같습니다:

```
개체 유형 <이름>은 하나 이상의 필드를 정의해야 합니다.
```

이 문제는 [jest](#) 구성이 [@nestjs/graphql/plugin](#) 플러그인을 어디에도 가져오지 않기 때문에 발생합니다.

이 문제를 해결하려면 e2e 테스트 디렉터리에 다음 파일을 생성하세요:

```
const transformer = require('@nestjs/graphql/plugin');

module.exports.name = 'nestjs-graphql-transformer';
// 아래 구성을 변경할 때마다 버전 번호를 변경해야 합니다. 그렇지 않으면 jest가 변
경 사항을 감지하지 못합니다 module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
```

```
// @nestjs/graphql/플러그인 옵션(비워둘 수 있음)
},
cs.program, // 이전 버전의 Jest의 경우 "cs.tsCompiler.program" (<= v27)
);
};
```

이 설정이 완료되면 **jest** 구성 파일에서 AST 트랜스포머를 가져옵니다. 기본적으로(스타터 애플리케이션에서) e2e 테스트 구성 파일은 **테스트 폴더** 아래에 있으며 이름은 **jest-e2e.json**입니다.

```
{
  ...
  // 기타 구성 "전역": {
    "ts-jest": {
      "astTransformers": {
        "전역": ["<위에서 생성한 파일 경로>"]
      }
    }
  }
}
```

jest@^29를 사용하는 경우 이전 접근 방식이 더 이상 사용되지 않으므로 아래 스니펫을 사용하세요.

```
{
  ...
  // 기타 구성 "변환": {
    "^.+\\.\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "전역": ["<위에서 생성한 파일 경로>"]
        }
      }
    ]
  }
}
```

공유 모델

경고 경고 이 장은 코드 우선 접근 방식에만 적용됩니다.

프로젝트 백엔드에 타입스크립트를 사용할 때의 가장 큰 장점 중 하나는 일반적인 타입스크립트 패키지를 사용하여 타입스크립트 기반 프론트엔드 애플리케이션에서 동일한 모델을 재사용할 수 있다는 점입니다.

하지만 문제가 있습니다. 코드 우선 접근 방식을 사용하여 생성된 모델은 GraphQL 관련 데코레이터로 많이 장식되어 있습니다. 이러한 데코레이터는 프론트엔드에서 관련성이 없어 성능에 부정적인 영향을 미칩니다.

모델 심 사용

이 문제를 해결하기 위해 NestJS는 웹팩(또는 이와 유사한) 구성을 사용하여 원래 데코레이터를 비활성 코드로 대체할 수 있는 "shim"을 제공합니다. 이 shim을 사용하려면 `@nestjs/graphql` 패키지와 shim 사이에 별칭을 구성합니다.

예를 들어 웹팩의 경우 이 문제는 다음과 같이 해결됩니다:

```
해결합니다: { // 참조: https://webpack.js.org/configuration/resolve/
  alias: {
    "@nestjs/graphql": path.resolve(__dirname,
    "../node_modules/@nestjs/graphql/dist/extras/graphql-model-shim")
  }
}
```

정보 힌트 [TypeORM](#) 패키지에는 [여기에서](#) 찾을 수 있는 유사한 심이 있습니다.

기타 기능

GraphQL 세계에서는 인증이나 작업의 부작용과 같은 문제를 처리하는 것에 대해 많은 논쟁이 있습니다. 비즈니스 로직 내부에서 처리해야 할까요? 고차 함수를 사용하여

권한 부여 로직으로 쿼리 및 변형을 개선해야 하나요? 아니면 [스키마 지시문을](#) 사용해야 할까요? 이러한 질문에 대한 정답은 하나도 없습니다.

Nest는 [가드](#) 및 [인터셉터와](#) 같은 크로스 플랫폼 기능으로 이러한 문제를 해결하도록 지원합니다. Nest의 철학은 중복을 줄이고 잘 구조화되고 가독성이 높으며 일관된 애플리케이션을 만드는 데 도움이 되는 도구를 제공하는 것입니다.

개요

GraphQL을 사용하면 표준 [가드](#), [인터셉터](#), [필터](#) 및 [파이프](#)를 다른 RESTful 애플리케이션과 동일한 방식으로 사용할 수 있습니다. 또한 [사용자 지정 데코레이터](#) 기능을 활용하여 자신만의 데코레이터를 쉽게 만들 수 있습니다. 샘플 GraphQL 쿼리 핸들러를 살펴보겠습니다.

```
쿼리('author') @사용가드
(AuthGuard)
async getAuthor(@Args('id', ParseIntPipe) id: number) {
  return this.authorsService.findOneById(id)를 반환합니다;
}
```

보시다시피 GraphQL은 HTTP REST 핸들러와 동일한 방식으로 가드 및 파이프 모두에서 작동합니다. 따라서 인증 로직을 가드로 옮길 수 있으며, REST와 GraphQL API 인터페이스 모두에서 동일한 가드 클래스를 재사용할 수도 있습니다. 마찬가지로 인터셉터도 두 가지 유형의 애플리케이션에서 동일한 방식으로 작동합니다:

```
돌연변이() @사용인터셉터(이벤트인터셉터)
async upvotePost(@Args('postId') postId: number) {
  return this.postsService.upvoteById({ id: postId });
}
```

실행 [컨테이너/common](#)에서 { CanActivate, ExecutionContext, Injectable }을
임포트합니다;
'@nestjs/graphql'에서 { GqlExecutionContext }를 임포트합니다;

GraphQL은 들어오는 요청에서 다른 유형의 데이터를 수신하기 때문에 가드와 인터셉터에서 수신하는 [실행 컨텍스트](#)는 GraphQL과 REST에서 다소 다릅니다. GraphQL 리졸버에는 [루트](#), [args](#), [컨텍스트](#), [정보 등](#) 고유한 인자 집합이 있습니다. 따라서 가드와 인터셉터는 일반 [ExecutionContext](#)를 [GqlExecutionContext](#)로 변환해야 합니다. 이것은 간단합니다:

```
@Injectable()
export class AuthGuard implements CanActivate {
  canActivate(context: ExecutionContext): boolean {
    const ctx = GqlExecutionContext.create(context);
    return true;
  }
}
```

`GqlExecutionContext.create()`가 반환하는 GraphQL 컨텍스트 객체에는 각 GraphQL 리졸버 인자에 대한 get 메서드가 노출됩니다(예: `getArgs()`, `getContext()` 등). 변환이 완료되면 현재 요청에 대한 GraphQL 인수를 쉽게 선택할 수 있습니다.

예외 필터

Nest 표준 예외 필터는 GraphQL 애플리케이션과도 호환됩니다. `ExecutionContext`와 마찬가지로 GraphQL 앱은 `ArgumentsHost` 객체를 `GqlArgumentsHost` 객체로 변환해야 합니다.

```
@Catch(HttpException)
export class HttpExceptionFilter implements GqlExceptionFilter {
  catch(exception: HttpException, host: ArgumentsHost) {
    const gqlHost = GqlArgumentsHost.create(host); 반환
    예외;
  }
}
```

정보 힌트 `GqlExceptionFilter`와 `GqlArgumentsHost`는 모두

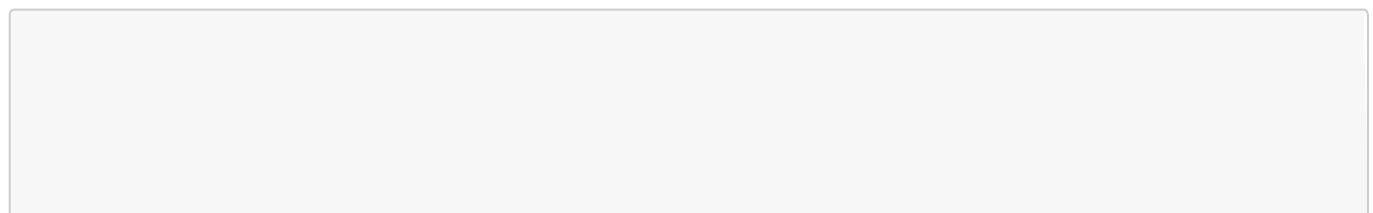
`nestjs/graphql` 패키지.

REST의 경우와 달리 기본 응답 객체를 사용하여 응답을 생성하지 않습니다. 사용자 지정 데코레이터

앞서 언급했듯이 사용자 지정 데코레이터 기능은 GraphQL 리졸버에서 예상대로 작동합니다.

```
export const User = createParamDecorator( (데이
터: 알 수 없음, ctx: ExecutionContext) =>
  GqlExecutionContext.create(ctx).getContext().user,
);
비동기 업보트포스트(
  @User() 사용자: 사용자 엔티티,
```

사용자() 사용자 지정 데코레이터를 다음과 같이 사용합니다:



```
@Args('postId') postId: 숫자,
) {}
```

정보 힌트 위의 예에서는 사용자 개체가 GraphQL 애플리케이션의 컨텍스트에 할당되어 있다고 가정했습니다.

필드 리졸버 수준에서 인핸서를 실행합니다.

GraphQL 컨텍스트에서 Nest는 필드 수준에서 인핸서(인터셉터, 가드 및 필터의 총칭)를 실행하지 않으며, 최상위 수준인 `@Query()`/`@Mutation()` 메서드에 대해서만 실행합니다([이 문제를 참조하세요](#)) .

`GqlModuleOptions`에서 `fieldResolverEnhancers` 옵션을 설정하여 `@ResolveField()`로 주석이 달린 메서드에 대해 인터셉터, 가드 또는 필터를 실행하도록 Nest에 지시할 수 있습니다. '인터셉터', '가드' 및/또는 '필터' 목록을 적절히 전달하세요:

```
GraphQLModule.forRoot({
  fieldResolverEnhancers: ['인터셉터']
}),
```

경고 필드 확인자에 인핸서를 활성화하면 많은 레코드를 반환하고 필드 확인자가 수천 번 실행될 때 성능 문제가 발생할 수 있습니다. 따라서 `fieldResolverEnhancers`를 활성화할 때는 필드 해석기에 꼭 필요하지 않은 인핸서의 실행을 건너뛰는 것이 좋습니다. 다음 도우미 함수를 사용하여 이 작업을 수행할 수

```
export 함수 isResolvingGraphQLField(context: ExecutionContext): boolean
{
  if (context.getType<GqlContextType>() === 'graphql') {
    const gqlContext = GqlExecutionContext.create(context);
    const info = gqlContext.getInfo();
    const parentType = info.parentType.name;
    반환 부모 유형 !== '쿼리' && 부모 유형 !== '돌연변이';
  }
  거짓을 반환합니다;
}
```

사용자 지정 드라이버 만들기

Nest는 두 가지 공식 드라이버를 기본으로 제공합니다: `nestjs/apollo`와 `@nestjs/mercurius`, 그리고 개발자가 새로운 커스텀 드라이버를 빌드할 수 있는 API를 제공합니다. 사용자 정의 드라이버를 사용하면 모든

GraphQL 라이브러리를 통합하거나 기존 통합을 확장하여 추가 기능을 추가할 수 있습니다.

예를 들어 `express-graphql` 패키지를 통합하려면 다음과 같은 드라이버 클래스를 만들 수 있습니다:

```
'@nestjs/graphql'에서 { AbstractGraphQLDriver, GqlModuleOptions }를 임포트하고,  
'express-graphql'에서 { graphqlHTTP }를 임포트합니다;
```

```
ExpressGraphQLDriver 클래스는 AbstractGraphQLDriver를 확장합니다 {  
    async start(옵션: GqlModuleOptions<any>): Promise<void> {  
        options = await this.graphQLFactory.mergeWithSchema(options);  
  
        const { httpAdapter } = this.httpAdapterHost;  
        httpAdapter.use(  
            '/graphql',  
            graphQLHTTP({  
                스키마: options.schema,  
                graphiql: true,  
            }),  
        );  
    }  
  
    async stop() {}  
}
```

그런 다음 다음과 같이 사용하세요:

```
GraphQLModule.forRoot({  
    드라이버: ExpressGraphQLDriver,  
});
```

연맹

페더레이션은 모놀리식 GraphQL 서버를 독립적인 마이크로서비스로 분할하는 수단을 제공합니다. 페더레이션은 게이트웨이와 하나 이상의 페더레이션 마이크로서비스라는 두 가지 구성 요소로 이루어져 있습니다. 각 마이크로서비스는 스키마의 일부를 보유하며 게이트웨이는 스키마를 클라이언트에서 사용할 수 있는 단일 스키마로 병합합니다.

[아폴로 문서를](#) 인용하자면, 페더레이션은 이러한 핵심 원칙에 따라 설계되었습니다:

- 그래프 작성은 선언적이어야 합니다. 페더레이션을 사용하면 명령형 스키마 스티칭 코드를 작성하는 대신 스키마 내에서 선언적으로 그래프를 작성할 수 있습니다.
- 코드는 유형이 아닌 관심사별로 분리해야 합니다. 사용자나 제품처럼 중요한 유형의 모든 측면을 한 팀이 제어할 수 없는 경우가 많으므로 이러한 유형의 정의는 중앙 집중식보다는 여러 팀과 코드베이스에 분산되어야 합니다.
- 그래프는 고객이 이해하기 쉽도록 단순해야 합니다. 연합된 서비스를 함께 사용하면 클라이언트에서 어 떻게 소비되는지 정확하게 반영하는 완전한 제품 중심의 그래프를 형성할 수 있습니다. 이는 사양을 준 수하는 언어의 기능만 사용하는 GraphQL일 뿐입니다. JavaScript뿐만 아니라 모든 언어가 페더레이션을 구현할 수 있습니다.

경고 경고 연합은 현재 구독을 지원하지 않습니다.

다음 섹션에서는 게이트웨이와 두 개의 페더레이션 엔드포인트로 구성된 데모 애플리케이션을 설정해 보겠습니다: 사용자 서비스 및 게시물 서비스입니다.

아폴로와의 연합

필요한 종속성을 설치하는 것으로 시작하세요:

```
$ npm install --save @apollo/federation @apollo/subgraph
```

스키마 우선

"사용자 서비스"는 간단한 스키마를 제공합니다. **key** 지시문에 주목하세요. 이 지시문은 Apollo 쿼리 플래너에 특정 **User** 인스턴스를 **지정하면** 해당 인스턴스를 가져올 수 있다고 지시합니다. 또한 **쿼리 유형을 확장한다는 점**

에 유의하세요.

```
type User @key(fields: "id") {  
    id: ID!  
    이름: 문자열!  
}
```

```
확장 유형 Query {  
    getUser(id: ID!): User  
}
```

리졸버는 `resolveReference()`라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 Apollo 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 `@ResolveReference()` 데코레이터를 사용하여 주석을 달아야 한다는 점에 유의하세요.

```
'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference }를 가져오고,  
'./users.service'에서 { UsersService }를 가져옵니다;  
  
@Resolver('User')  
사용자 해결자 클래스 내보내기 {  
  constructor(private usersService: UsersService) {}  
  
  쿼리()  
  getUser(@Args('id') id: 문자열) {  
    this.usersService.findById(id)를 반환합니다;  
  }  
  
  @ResolveReference()  
  resolveReference(reference: { __typename: 문자열; id: 문자열 }) {  
    return this.usersService.findById(reference.id);  
  }  
}
```

마지막으로, `GraphQLModule`을 등록하고

`ApolloFederationDriver` 드라이버를 구성 개체에 추가합니다:

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./users.resolver'에서 { UsersResolver }를 가져옵니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  제공자: [UsersResolver],
})
내보내기 클래스 AppModule {}
```

코드 우선

먼저 사용자 엔티티에 몇 가지 데코레이터를 추가합니다.

```
'@nestjs/graphql'에서 { 지시어, 필드, ID, 객체 유형 } 임포트; @ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field((유형) => ID)
  id: 숫자;

  @Field() 이름:
  문자열;
}
```

리졸버는 `resolveReference()`라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 Apollo 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 `@ResolveReference()` 데코레이터를 사용하여 주석을 달아야 한다는 점에 유의하세요.

```
'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference } 가져오기;
'./user.entity'에서 { User } 가져오기;
'./users.service'에서 { UsersService }를 가져옵니다;

@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    this.usersService.findById(id)를 반환합니다;
  }

  @ResolveReference()
  resolveReference(reference: { __유형명: 문자열; id: 숫자 }): User {
    return this.usersService.findById(reference.id);
  }
}
```

마지막으로, GraphQLModule을 등록하고

`ApolloFederationDriver` 드라이버를 구성 개체에 추가합니다:

```
import {  
  ApolloFederationDriver,  
  ApolloFederationDriverConfig,  
}를 '@nestjs/apollo'에서 가져옵니다;  
'@nestjs/common'에서 { Module }을 가져옵니다;  
'./users.resolver'에서 { UsersResolver }를 가져옵니다;  
'./users.service'에서 { UserService } import; // 이 예제에는 포함되지 않았습니다.  
다.
```

```
모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
    }),
  ],
  제공자: [UsersResolver, UsersService],
}

내보내기 클래스 AppModule {}
```

코드 우선 모드에서는 [여기에서](#), 스키마 우선 모드에서는 [여기에서](#) 작동하는 예제를 확인할 수

있습니다. 페더레이션 예제: Posts

게시물 서비스는 `getPosts` 쿼리를 통해 집계된 게시물을 제공해야 하지만, 사용자 유형을 `user.posts` 필드와 함께 입력합니다.

스키마 우선

"게시물 서비스"는 `확장` 키워드로 표시하여 스키마에서 사용자 유형을 참조합니다. 또한 사용자 유형에 대해 하나의 추가 속성(`게시물`)을 선언합니다. User 인스턴스를 일치시키는 데 사용되는 `@key` 지시어와 `id` 필드가 다른 곳에서 관리됨을 나타내는 `@external` 지시어에 주목하세요.

```
type Post @key(fields: "id") {
  id: ID!
  제목: String!
  body: 문자열! 사
  용자: 사용자
}
```

```
확장 유형 사용자 @key(fields: "id") {
  id: ID! 외부
  게시물: [게시물]
}
```

```
확장 유형 Query { getPosts:
  [Post]
}
```

다음 예제에서 **포스트 리졸버는 다음을 포함하는 참조를 반환하는** `getUser()` 메서드를 제공합니다. _____ 유

형 이름과 애플리케이션이 참조를 확인하는 데 필요할 수 있는 몇 가지 추가 속성(이 경우 `id`)이 포함된 참조를 반환하는 메서드를 제공합니다. _____ 타입네임은 GraphQL 게이트웨이에서 사용자 유형을 담당하는 마이크로 서비스를 정확히 찾아내고 해당 인스턴스를 검색하는 데 사용됩니다. 위에서 설명한 "사용자 서비스"는 `resolveReference()` 메서드를 실행할 때 요청됩니다.

```
'@nestjs/graphql'에서 { Query, Resolver, Parent, ResolveField } 가져오기;
'./posts.service'에서 { PostsService } 가져오기;
'./posts.interfaces'에서 { Post }를 가져옵니다;

@Resolver('Post')
내보내기 클래스 PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    this.postsService.findAll()을 반환합니다;
  }

  @ResolveField('user')
  getUser(@Parent() post: Post) {
    반환 { _____유형명: '사용자', id: post.userId };
  }
}
```

마지막으로, '사용자 서비스' 섹션에서 한 것과 유사하게 GraphQLModule을 등록해야 합니다.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./posts.resolver'에서 { PostsResolver }를 가져옵니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      typePaths: ['**/*.graphql'],
    }),
  ],
  공급자: [게시물 해결자],
})
내보내기 클래스 AppModule {}
```

코드 우선

먼저 User 엔티티를 나타내는 클래스를 선언해야 합니다. 엔티티 자체는 다른 서비스에 있지만, 여기서는 이 엔티티를 사용(정의 확장)할 것입니다. extends 및 @external 지시어에 유의하세요.

```
'@nestjs/graphql'에서 { 지시문, 객체 유형, 필드, ID } 가져오기;
```

```
'./post.entity'에서 { Post } 가져오기;
```

```
객체유형() @디렉티브('@확장') @디렉티브  
( '@키(필드: "id")' ) 내보내기 클래스  
사용자 {  
    @Field((type) => ID) @디  
    렉티브('@외부') id: 숫자;  
  
    @Field((type) => [Post])  
    posts? Post[];  
}
```

이제 다음과 같이 `User` 엔티티에서 확장에 해당하는 리졸버를 만들어 보겠습니다:

```
'@nestjs/graphql'에서 { 부모, ResolveField, 해결자 }를 가져오고,  
'./posts.service'에서 { PostsService }를 가져옵니다;  
'./post.entity'에서 { Post } 가져오기;  
'./user.entity'에서 { User } 가져오기;  
  
@Resolver((of) => User)  
export class UsersResolver {  
    생성자(비공개 읽기 전용 postsService: PostsService) {}  
  
    @ResolveField((of) => [Post])  
    public posts(@Parent() user: User): Post[] {  
        return this.postsService.forAuthor(user.id);  
    }  
}
```

또한 `Post` 엔티티 클래스를 정의해야 합니다:

```
'@nestjs/graphql'에서 { 지시어, 필드, ID, Int, 객체 유형 } 가져오기;
'./user.entity'에서 { 사용자 } 가져오기;

객체 유형() @디렉티브( '@키(필드:
  "id")' ) 내보내기 클래스 Post {
  @Field((유형) => ID)
  id: 숫자;

  @Field()
  title: 문자열;

  @Field((유형) => Int)
  authorId: 숫자;

  @Field((type) => User)
  user?: 사용자;
}
```

그리고 그 해결사:

```
'@nestjs/graphql'에서 { Query, Args, ResolveField, Resolver, Parent
}를 임포트합니다;

'./posts.service'에서 { PostsService } 가져오기;
'./post.entity'에서 { Post } 가져오기;
'./user.entity'에서 { User }를 가져옵니다;

@Resolver((of) => Post)
export class PostsResolver {
  생성자(비공개 읽기 전용 postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    이.postsService.findOne(id)을 반환합니다;
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    this.postsService.all()을 반환합니다;
  }

  @ResolveField((of) => User)
  user(@Parent() post: Post): any {
    반환 { _____유형명: '사용자', id: post.authorId };
  }
}
```

마지막으로 모듈에 함께 묶습니다. 스키마 빌드 옵션에서 `User`가 고아(외부) 유형임을 지정하는 것에 주목하세요.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./posts.resolvers'에서 { PostsResolvers }을 가져오고,
'./users.resolvers'에서 { UsersResolvers }을 가져옵니다;
'./posts.service'에서 { PostsService } 가져오기; // 예제에는 포함되지 않았습니다.

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloFederationDriverConfig>({
      driver: ApolloFederationDriver,
      autoSchemaFile: true,
      buildSchemaOptions: {
        고아 유형: [사용자],
      },
    },
  ],
}
```

```

    },
],
공급자: [게시자 해결자, 사용자 해결자, 게시자 서비스],
})
내보내기 클래스 AppModule {}

```

코드 우선 모드의 경우 [여기에서](#), 스키마 우선 모드의 경우 [여기에서](#) 작동하는 예제를 확인할 수 있습니다. 페

더레이션 예제: 게이트웨이

필요한 종속성을 설치하는 것으로 시작하세요:

```
npm install --save @apollo/gateway
```

게이트웨이는 엔드포인트 목록을 지정해야 하며 해당 스키마를 자동으로 검색합니다. 따라서 게이트웨이 서비스의 구현은 코드 접근 방식과 스키마 우선 접근 방식 모두 동일하게 유지됩니다.

```

'@apollo/gateway'에서 { IntrospectAndCompose }를 가져옵니다;
'@nestjs/apollo'에서 { ApolloGatewayDriver, ApolloGatewayDriverConfig }를 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;

```

```

모듈({ import:
[
  GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
    driver: ApolloGatewayDriver,
    서버: {
      // ... 아폴로 서버 옵션 cors:
      true,
    },
    게이트웨이: {
      supergraphSdl: new IntrospectAndCompose({
        subgraphs: [
          { name: 'users', url: 'http://user-service/graphql' },
          { name: 'posts', url: 'http://post-service/graphql' },
        ],
      }),
    },
  }),
]
})
내보내기 클래스 AppModule {}

```

코드 우선 모드의 경우 [여기에서](#), 스키마 우선 모드의 경우 [여기에서](#) 작업 예제를 확인할 수 있습니다.

머큐리우스와의 연맹

필요한 종속성을 설치하는 것으로 시작하세요:

```
$ npm install --save @apollo/subgraph @nestjs/mercurius
```

정보 참고 서브그래프 스키마를 빌드하려면 `@apollo/subgraph` 패키지가 필요합니다([빌드Sub그래프 스키마, 프린트Sub그래프스키마](#) 함수).

스키마 우선

"사용자 서비스"는 간단한 스키마를 제공합니다. `key` 지시문에 주목하세요. 이 지시문은 사용자가 `ID`를 지정하면 `사용자` 인스턴스를 가져올 수 있도록 Mercurius 쿼리 플래너에 지시합니다. 또한 `쿼리` 유형을 [확장한다는](#) 점에 유의하세요.

```
type User @key(fields: "id") {
  id: ID!
  이름: 문자열!
}
```

```
확장 유형 Query {
  getUser(id: ID!): User
}
```

리졸버는 `resolveReference()`라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 머큐리우스 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 `@ResolveReference()` 데코레이터로 주석을 달아야 한다는 점에 유의하세요.

'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference }를 가져오고,
'./users.service'에서 { UserService }를 가져옵니다;

```
@Resolver('User')
사용자 해결자 클래스 내보내기 {
  constructor(private userService: UserService) {}

  쿼리()
  getUser(@Args('id') id: 문자열) {
    this.userService.findById(id)를 반환합니다;
  }

  @ResolveReference()
  resolveReference(reference: { __typename: 문자열; id: 문자열 }) {
    return this.userService.findById(reference.id);
  }
}
```

마지막으로, GraphQLModule을 등록하고

MercuriusFederationDriver 드라이버를 구성 개체에 추가합니다:

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./users.resolver'에서 { UsersResolver }를 가져옵니다;

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      typePaths: ['**/*graphql'],
      federationMetadata: true,
    }),
  ],
  제공자: [UsersResolver],
})
내보내기 클래스 AppModule {}
```

코드 우선

먼저 사용자 엔티티에 몇 가지 데코레이터를 추가합니다.

```
'@nestjs/graphql'에서 { 지시어, 필드, ID, 객체 유형 } 임포트; @ObjectType()
@Directive('@key(fields: "id")')
export class User {
  @Field(() => ID)
  id: 문자;

  @Field() 이름:
  문자열;
}
```

리졸버는 resolveReference()라는 메서드 하나를 추가로 제공합니다. 이 메서드는 관련 리소스에 사용자 인스턴스가 필요할 때마다 머큐리우스 게이트웨이에 의해 트리거됩니다. 나중에 포스트 서비스에서 이에 대한 예를 살펴보겠습니다. 이 메서드에는 @ResolveReference() 데코레이터로 주석을 달아야 한다는 점에 유의하세요

요.

```
'@nestjs/graphql'에서 { Args, Query, Resolver, ResolveReference } 가져오기;  
'./user.entity'에서 { User } 가져오기;  
'./users.service'에서 { UsersService }를 가져옵니다;
```

```

@Resolver((of) => User)
export class UsersResolver {
  constructor(private usersService: UsersService) {}

  @Query((returns) => User)
  getUser(@Args('id') id: number): User {
    this.usersService.findById(id)를 반환합니다;
  }

  @ResolveReference()
  resolveReference(reference: { __유형명: 문자열; id: 숫자 }): User {
    return this.usersService.findById(reference.id);
  }
}

```

마지막으로, GraphQLModule을 등록하고

MercuriusFederationDriver 드라이버를 구성 개체에 추가합니다:

```

import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'./users.resolver'에서 { UsersResolver }를 가져옵니다;
'./users.service'에서 { UserService } import; // 이 예제에는 포함되지 않았습니다.

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true, 페더레이션
      메타데이터: true,
    }),
  ],
  제공자: [UsersResolver, UserService],
})
내보내기 클래스 AppModule {}

```

페더레이션 예시: 글

게시물 서비스는 getPosts 쿼리를 통해 집계된 게시물을 제공해야 하지만, 사용자

유형을 `user.posts` 필드와 함께 입력합니다.

스키마 우선

"게시물 서비스"는 `확장` 키워드로 표시하여 스키마에서 `사용자` 유형을 참조합니다. 또한 `사용자` 유형에 대한 추가 속성(`게시물`)을 하나 더 선언합니다. 매칭에 사용되는 `@key` 지시문에 유의하세요.

인스턴스 및 `@external` 지시문을 사용하여 `ID` 필드가 다른 곳에서 관리됨을 나타냅니다.

```
type Post @key(fields: "id") {
  id: ID!
  제목: String!
  body: 문자열! 사
  용자: 사용자
}

확장 유형 사용자 @key(fields: "id") {
  id: ID! 외부
  게시물: [게시물]
}

확장 유형 Query { getPosts:
  [Post]
}
```

다음 예제에서 `포스트` 리졸버는 다음을 포함하는 참조를 반환하는 `getUser()` 메서드를 제공합니다. _____ 유형 이름과 애플리케이션이 참조를 확인하는 데 필요할 수 있는 몇 가지 추가 속성(이 경우 `id`)이 포함된 참조를 반환하는 메서드를 제공합니다. _____ 타입네임은 GraphQL 게이트웨이에서 사용자 유형을 담당하는 마이크로서비스를 정확히 찾아내고 해당 인스턴스를 검색하는 데 사용됩니다. 위에서 설명한 "사용자 서비스"는 `resolveReference()` 메서드를 실행할 때 요청됩니다.

```
'@nestjs/graphql'에서 { Query, Resolver, Parent, ResolveField } 가져오기;
'./posts.service'에서 { PostsService } 가져오기;
'./posts.interfaces'에서 { Post }를 가져옵니다;

@Resolver('Post')
내보내기 클래스 PostsResolver {
  constructor(private postsService: PostsService) {}

  @Query('getPosts')
  getPosts() {
    this.postsService.findAll()을 반환합니다;
  }

  ResolveField('user')
  getUser(@Parent() post: Post) {
    반환 { _____유형명: '사용자', id: post.userId };
  }
}
```

마지막으로, '사용자 서비스' 섹션에서 한 것과 유사하게 `GraphQLModule`을 등록해야 합니다.

```
import {  
  MercuriusFederationDriver,  
  MercuriusFederationDriverConfig,
```

```

}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 가져오고,
'./posts.resolver'에서 { PostsResolver }를 가져옵니다;

모듈({ import:
[
  GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
    driver: MercuriusFederationDriver,
    페더레이션 메타데이터: true, 유형 경로
    : ['**/*.graphql'],
  }),
],
공급자: [게시물 해결자],
})
내보내기 클래스 AppModule {}

```

코드 우선

먼저 `User` 엔티티를 나타내는 클래스를 선언해야 합니다. 엔티티 자체는 다른 서비스에 있지만, 여기서는 이 엔티티를 사용(정의 확장)할 것입니다. `extends` 및 `@external` 지시어에 유의하세요.

```

'@nestjs/graphql'에서 { 지시문, 객체 유형, 필드, ID } 가져오기;
'./post.entity'에서 { Post } 가져오기;

객체유형() @디렉티브('@확장') @디렉티브
('@키(필드: "id")') 내보내기 클래스

사용자 {
  @Field((type) => ID) @디
  렉티브('@외부') id: 숫자;

  @Field((type) => [Post])
  posts? Post[];
}

```

이제 다음과 같이 `User` 엔티티에서 확장에 해당하는 리졸버를 만들어 보겠습니다:

```
'@nestjs/graphql'에서 { 부모, ResolveField, 해결자 }를 가져오고,  
'./posts.service'에서 { PostsService }를 가져옵니다;  
'./post.entity'에서 { Post } 가져오기;  
'./user.entity'에서 { User } 가져오기;  
  
@Resolver((of) => User)  
export class UsersResolver {  
  생성자(비공개 읽기 전용 postsService: PostsService) {}  
  
  @ResolveField((of) => [Post])
```

```
public posts(@Parent() user: User): Post[] {
    return this.postsService.forAuthor(user.id);
}
```

또한 [포스트](#) 엔티티 클래스를 정의해야 합니다:

```
'@nestjs/graphql'에서 { Directive, Field, ID, Int, ObjectType } 가져오기;
'./user.entity'에서 { User } 가져오기;

객체 유형() @디렉티브('@키(필드:
"id")') 내보내기 클래스 Post {
    @Field((유형) => ID)
    id: 숫자;

    @Field()
    title: 문자열;

    @Field((유형) => Int)
    authorId: 숫자;

    @Field((type) => User)
    user?: 사용자;
}
```

그리고 그 해결사:

```
'@nestjs/graphql'에서 { Query, Args, ResolveField, Resolver, Parent
}를 임포트합니다;

'./posts.service'에서 { PostsService } 가져오기;
'./post.entity'에서 { Post } 가져오기;
'./user.entity'에서 { User }를 가져옵니다;

@Resolver((of) => Post)
export class PostsResolver {
  생성자(비공개 읽기 전용 postsService: PostsService) {}

  @Query((returns) => Post)
  findPost(@Args('id') id: number): Post {
    이.postsService.findOne(id)을 반환합니다;
  }

  @Query((returns) => [Post])
  getPosts(): Post[] {
    this.postsService.all()을 반환합니다;
  }

  @ResolveField((of) => User)
```

```
user(@Parent() post: Post): any {
  반환 { _____유형명: '사용자', id: post.authorId };
}
}
```

마지막으로 모듈에 함께 묶습니다. 스키마 빌드 옵션에서 `User`가 고아(외부) 유형임을 지정하는 것에 주목하세요.

```
import {
  MercuriusFederationDriver,
  MercuriusFederationDriverConfig,
}를 '@nestjs/mercurius'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./posts.resolvers'에서 { PostsResolvers }를 가져오고,
'./users.resolvers'에서 { UsersResolvers }를 가져옵니다;
'./posts.service'에서 { PostsService } 가져오기; // 예제에는 포함되지 않았습니다.

모듈({ import:
  [
    GraphQLModule.forRoot<MercuriusFederationDriverConfig>({
      driver: MercuriusFederationDriver,
      autoSchemaFile: true,
      federationMetadata: true,
      buildSchemaOptions: {
        고아 유형: [사용자],
      },
    }),
  ],
  공급자: [게시자 해결자, 사용자 해결자, 게시자 서비스],
})
내보내기 클래스 AppModule {}
```

페더레이션 예제: 게이트웨이

게이트웨이는 엔드포인트 목록을 지정해야 하며 해당 스키마를 자동으로 검색합니다. 따라서 게이트웨이 서비스의 구현은 코드 접근 방식과 스키마 우선 접근 방식 모두 동일하게 유지됩니다.

```
import {  
  MercuriusGatewayDriver,  
  MercuriusGatewayDriverConfig,  
}를 '@nestjs/mercurius'에서 가져옵니다;  
'@nestjs/common'에서 { Module }을 가져옵니다;  
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;
```

모듈({ import:

[

```

GraphQLModule.forRoot<MercuriusGatewayDriverConfig>({
  driver: MercuriusGatewayDriver,
  gateway: {
    services: [
      { name: 'users', url: 'http://user-service/graphql' },
      { name: 'posts', url: 'http://post-service/graphql' },
    ],
  },
}),
]
})
내보내기 클래스 AppModule {}

```

연맹 2

아폴로 문서를 인용하자면, 페더레이션 2는 원래 아폴로 페더레이션(이 문서에서는 페더레이션 1이라고 함)에서 개발자 환경을 개선한 것으로, 대부분의 원래 슈퍼그래프와 하위 호환됩니다.

경고 Mercurius는 페더레이션 2를 완전히 지원하지 않습니다. 페더레이션 2를 지원하는 라이브러리 목록은 [여기에서](#) 확인할 수 있습니다.

다음 섹션에서는 이전 예제를 페더레이션 2로 업그레이드하겠습니다. 페더레이션

예제: 사용자

페더레이션 2의 한 가지 변경 사항은 엔티티에 원본 하위 그래프가 없으므로 [쿼리를](#) 확장할 필요가 없다는 것입니다. 를 더 이상 사용할 수 없습니다. 자세한 내용은 아폴로 페더레이션 2 문서에서 [엔티티 주제를](#)

참조하세요. 스키마 먼저

스키마에서 [확장](#) 키워드를 제거하기만 하면 됩니다.

```

type User @key(fields: "id") {
  id: ID!
  이름: 문자열!
}

```

```

유형 쿼리 {
  getUser(id: ID!): User
}

```

코드 우선

페더레이션 2를 사용하려면 자동 스키마 파일 옵션에서 페더레이션 버전을 지정해야 합니다.

```
import {  
  ApolloFederationDriver,  
  ApolloFederationDriverConfig,
```

```
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져옵니다;
'./users.resolver'에서 { UsersResolver }를 가져옵니다;
'./users.service'에서 { UserService } import; // 이 예제에는 포함되지 않았습니다.
```

모듈({ import:

```
[  
  GraphQLModule.forRoot<ApolloFederationDriverConfig>({  
    driver: ApolloFederationDriver,  
    autoSchemaFile: { 폐  
      더레이션: 2,  
    },  
  }),  
],
```

제공자: [UsersResolver, UserService],
})

내보내기 클래스 AppModule {}

페더레이션 예시: 글

위와 같은 이유로 사용자 및 쿼리를 더 이상 확장할 필요가 없습니다. 스키마 먼저

스키마에서 확장 지시문과 외부 지시문을 간단히 제거할 수 있습니다.

```
type Post @key(fields: "id") {  
  id: ID!  
  제목: String!  
  body: 문자열! 사  
  용자: 사용자  
}
```

```
type User @key(fields: "id") {  
  id: ID!  
  게시물: [게시물]  
}
```

```
유형 Query {  
  getPosts: [Post]  
}
```

코드 우선

사용자 엔터티를 더 이상 확장하지 않으므로 사용자에서 확장 및 외부 지시문을 제거하기만 하면 됩니다.



```
'@nestjs/graphql'에서 { 지시문, 객체 유형, 필드, ID } 가져오기;
'./post.entity'에서 { Post } 가져오기;

객체 유형() @디렉티브('@키(필드:
"id")') 내보내기 클래스 사용자 {
  @Field((유형) => ID)
  id: 숫자;

  @Field((type) => [Post])
  posts? Post[];
}
```

또한 사용자 서비스와 마찬가지로 GraphQLModule에서 페더레이션 2를 사용하도록 지정해야 합니다.

```
import {
  ApolloFederationDriver,
  ApolloFederationDriverConfig,
}를 '@nestjs/apollo'에서 가져옵니다;
'@nestjs/common'에서 { Module }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./posts.resolvers'에서 { PostsResolvers }를 가져오고,
'./users.resolvers'에서 { UsersResolvers }를 가져옵니다;
'./posts.service'에서 { PostsService } 가져오기; // 예제에는 포함되지 않았습니다.

모듈({ import:
[
  GraphQLModule.forRoot<ApolloFederationDriverConfig>({
    driver: ApolloFederationDriver,
    autoSchemaFile: { 페
      더레이션: 2,
    },
    빌드 스키마 옵션: { 고아 유형: [사용자
      ],
    },
  }),
],
  공급자: [게시자 해결자, 사용자 해결자, 게시자 서비스],
})
내보내기 클래스 AppModule {}
```

v10에서 v11로 마이그레이션

이 장에서는 [@nestjs/graphql](#) 버전 10에서 버전 11로 마이그레이션하기 위한 일련의 지침을 제공합니다. 이번 주요 릴리스의 일환으로 Apollo 드라이버가 Apollo Server v4(v3 대신)와 호환되도록 업데이트되었습니다. 참고: Apollo Server v4에는 몇 가지 중요한 변경 사항(특히 플러그인 및 에코시스템 패키지 관련)이 있으므로 그에 따라 코드베이스를 업데이트해야 합니다. 자세한 내용은 [Apollo Server v4 마이그레이션 가이드](#)를 참조하세요.

아폴로 패키지

`apollo-server-express` 패키지를 설치하는 대신 [@apollo/server](#)를 설치해야 합니다:

```
$ npm 제거 아폴로 서버-익스프레스  
npm 설치 @apollo/server
```

Fastify 어댑터를 사용하는 경우 [@as-integrations/fastify](#) 패키지를 대신 설치해야 합니다:

```
$ npm 제거 아폴로-서버-패스티파이  
npm install @apollo/server @as-integrations/fastify
```

머큐리우스 패키지

머큐리우스 게이트웨이는 더 이상 [머큐리우스](#) 패키지의 일부가 아닙니다. 대신

[mercuriusjs/gateway](#) 패키지를 별도로 다운로드하세요:

```
$ npm 설치 @mercuriusjs/gateway
```

마찬가지로 페더레이션 스키마를 생성하려면 [@mercuriusjs/federation](#) 패키지를 설치해야 합니다:

```
$ npm 설치 @mercuriusjs/federation
```

v9에서 v10으로 마이그레이션

이 장에서는 [@nestjs/graphql](#) 버전 9에서 버전 10으로 마이그레이션하기 위한 일련의 지침을 제공합니다. 이번

메이저 버전 릴리즈의 초점은 플랫폼에 구애받지 않는 더 가벼운 코어 라이브러리를 제공하는 것입니다.

"드라이버" 패키지 소개

최신 버전에서는 [@nestjs/graphql](#) 패키지를 몇 개의 개별 라이브러리로 분리하여 프로젝트에서 Apollo([@nestjs/apollo](#)), Mercurius([@nestjs/mercurius](#)) 또는 다른 GraphQL 라이브러리를 사용할지 선택할 수 있도록 결정했습니다.

즉, 이제 애플리케이션에서 사용할 드라이버를 명시적으로 지정해야 합니다.

```
// 이전
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot({
      autoSchemaFile: 'schema.gql',
    }),
  ],
})
내보내기 클래스 AppModule {}

// 이후
'@nestjs/apollo'에서 { ApolloDriver, ApolloDriverConfig }를 임포트하고
, '@nestjs/common'에서 { Module }을 임포트합니다;
'@nestjs/graphql'에서 { GraphQLModule }을 임포트합니다;

모듈({ import:
  [
    GraphQLModule.forRoot<ApolloDriverConfig>({
      driver: ApolloDriver,
      자동 스키마 파일: 'schema.gql',
    }),
  ],
})
내보내기 클래스 AppModule {}
```

플러그인

Apollo 서버 플러그인을 사용하면 특정 이벤트에 대한 응답으로 사용자 정의 작업을 수행할 수 있습니다. 이 기능은 Apollo 전용 기능이기 때문에 `@nestjs/graphql`에서 새로 생성된 `@nestjs/apollo` 패키지로 옮겼으므로 애플리케이션에서 임포트를 업데이트해야 합니다.

```
// 이전
'@nestjs/graphql'에서 { Plugin }을 가져옵니다;

// 이후
'@nestjs/apollo'에서 { Plugin }을 가져옵니다;
```

지시어

`schemaDirectives` 기능이 `@graphql-tools/schema` 패키지의 v8에서 새로운 [Schema 지시문 API](#)로 대체되었습니다.

```
// 이전

'@graphql-tools/utils'에서 { SchemaDirectiveVisitor }를 가져오고,
'graphql'에서 { defaultFieldResolver, GraphQLField }를 가져옵니다;

export classUpperCaseDirective extends SchemaDirectiveVisitor {
  visitFieldDefinition(field: GraphQLField<any, any>) {
    const { resolve = defaultFieldResolver } = field;
    field.resolve = async 함수 (...args) {
      const result = await resolve.apply(this, args);
      if (typeof result === 'string') {
        결과값을 반환합니다;
      }
      결과를 반환합니다;
    };
  }
}

// 이후

'@graphql-tools/utils'에서 { getDirective, MapperKind, mapSchema }를 가져옵니다;
'graphql'에서 { defaultFieldResolver, GraphQLSchema }를 가져옵니다;

내보내기 함수 upperDirectiveTransformer( 스키마:
  GraphQLSchema,
  지시어 이름: 문자열,
) {
  return mapSchema(schema, {
    [MapperKind.OBJECT_FIELD]: (fieldConfig) => {
      const upperDirective = getDirective(스키마,
        fieldConfig, 지시어 이름,
        )?.[0];
      if (upperDirective) {
        const { resolve = defaultFieldResolver } = fieldConfig;
```

통화

```
{
  },
  );
}
```

```
// 원래 리졸  
버를 *먼저*  
다음과 같은  
함수로 바꿉  
니다.  
(source, args, context, info); if (typeof result ===  
'string') {  
    결과값을 반환합니다;  
}  
결과를 반환합니다;  
};  
필드 컨피그를 반환합니다;
```

// 원래 리졸

버를 호출한

다음 결과를

대문자로 변

환합니다.

```
fieldConfig.resolve  
= async 함  
수(source,  
args,  
context,  
info).
```

c
o
n
s
t

r
e
s
u
l
t

=

a
w
a
i
t

r
e
s
o
l
v
e

이 지시문 구현을 `@upper` 지시문이 포함된 스키마에 적용하려면

변환 스키마 함수입니다:

```
GraphQLModule.forRoot<ApolloDriverConfig>({
  ...
  transformSchema: schema => upperDirectiveTransformer(schema, 'upper'),
})
```

연맹

GraphQLFederationModule이 제거되고 해당 드라이버 클래스로 대체되었습니다:

```
// 이전에
GraphQLFederationModule.forRoot({
  autoSchemaFile: true,
});

// 이후 GraphQLModule.forRoot<ApolloFederationDriverConfig>({
  드라이버: ApolloFederationDriver,
  autoSchemaFile: true,
});
```

정보 힌트 `ApolloFederationDriver` 클래스와 `ApolloFederationDriverConfig`는 모두
`@nestjs/apollo` 패키지에서 내보냅니다.

마찬가지로 전용 `GraphQLGatewayModule`을 사용하는 대신 적절한 드라이버를 전달하기만 하면 됩니다.

클래스를 `GraphQLModule` 설정에 추가합니다:

```
// 이전에
GraphQLGatewayModule.forRoot({
  게이트웨이: {
    supergraphSdl: new IntrospectAndCompose({
      subgraphs: [
        { name: 'users', url: 'http://localhost:3000/graphql' },
        { name: 'posts', url: 'http://localhost:3001/graphql' },
      ],
    }),
  },
});

// 이후
GraphQLModule.forRoot<ApolloGatewayDriverConfig>({
  드라이버: 아폴로게이트웨이드라이버, 게이트웨이: {
    supergraphSdl: new IntrospectAndCompose({
```

```
하위 그래프: [
  { name: 'users', url: 'http://localhost:3000/graphql' },
  { name: 'posts', url: 'http://localhost:3001/graphql' },
],
}),
},
});
```

정보 힌트 [ApolloGatewayDriver](#) 클래스와 [ApolloGatewayDriverConfig](#)는 모두
[@nestjs/apollo](#) 패키지에서 내보냅니다.