소개

Nest(NestJS)는 효율적이고 확장 가능한 Node.js 서버 측 애플리케이션을 구축하기 위한 프레임워크입니다. 이 프레임워크는 프로그레시브 자바스크립트를 사용하며, 타입스크립트를 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 Fastify도 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, Vue와 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 CLI를 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 degit을 사용하면 됩니다.

브라우저를 열고 `http://localhost:3000/` 으로 이동합니다.

스타터 프로젝트의 JavaScript 버전을 설치하려면 위의 명령 순서대로 `javascript-starter.`git을 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 TypeScript를 사랑하지만 무엇보다도 Node.js를 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. npm이 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. NestFactory는 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 `1`과 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. 기본적으로 `@nestjs/platform-express` 패키지가 사용됩니다. 많은 사용자가 Express를 잘 사용하고 있으며, 이를 활성화하기 위해 별도의 조치를 취할 필요가 없습니다.
플랫폼- 패스트파이브	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 cats와 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. 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	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 `true로` 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(), @응답(), @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 `HttpCode`를 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 `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, url과 `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` 고양이를 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.id를 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 `AppModule을` 제외한 다른 모듈을 아직 정의하지 않았으므로 이를 사용하여 `CatsController를` 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 객체를 사용하는 것입니다. 특정 응답 객체를 삽입하려면 `@Res()` 데코레이터를 사용해야 합니다. 차이점을 보여주기 위해 `CatsController를` 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 `true로` 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 `CatsService는` 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 CatsModule을 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 `CatsModule을` 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 `CatsModule을` 임포트하는 모든 모듈은 `CatsService에` 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 CommonModule을 `CoreModule에서` 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 종속성으로 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 필요한 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 `true로` 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 `next라는` 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 개체를 변경합니다. 요청-응답 주기를 종료
> - 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 `생성자를` 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 `LoggerMiddleware를` 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`을 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.

경로 와일드카드

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 `LoggerMiddleware는 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 미들웨어를 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(HttpException도 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 `메시지` 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, CatsController에는 `findAll()` 메서드(`GET` 라우트 핸들러)가 있습니다. 이 라우트 핸들러가 어떤 이유로 예외를 던진다고 가정해 봅시다. 이를 시연하기 위해 다음과 같이 하드코딩하겠습니다:

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

> 정보 힌트 여기서는 `HttpStatus`를 사용했습니다. 이것은 헬퍼 열거 형의 `nestjs/common` 패키지.

클라이언트가 이 엔드포인트를 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 `객체일` 수 있습니다.
- `status` 인수는 HTTP 상태 코드를 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 status는 유효한 HTTP 상태 코드여야 합니다. 가장 좋은 방법은 `@nestjs/common`에서 가져온 `HttpStatus` 열거형을 사용하는 것입니다.

오류 원인을 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, HttpException을 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);
  }
}
```

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

- `BadRequestException`
- `UnauthorizedException`
- `NotFoundException`
- `ForbiddenException`
- `NotAceptableException`
- `RequestTimeoutException`
- `ConflictException`
- `GoneException`
- `HttpVersionNotSupportedException`
-
-

```
PayloadTooLargeException
UnsupportedMediaTypeException
UnprocessableEntityException
InternalServerErrorException
```

- NotImplementedException
- ImATeapotException
- MethodNotAllowedException
- BadGatewayException
- ServiceUnableException
- 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 유형을 나타냅니다.

> 경고 @nestjs/platform-fastify를 사용하는 경우 `응답.send()`를 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 유틸리티 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 모듈에서 동일한 클래스의 인스턴스를 쉽게 재사용할 수 있으므로 메모리 사용량을 줄일 수 있습니다.

위의 예제에서 HttpExceptionFilter는 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 `HttpExceptionFilter를` 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 어댑터를 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 임포트합니
다;

@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 인스턴스화해서는 안 됩니다. 대신 프레임워크가 자동으로 인스턴스화하도록 하세요.

위의 구현은 접근 방식을 보여주는 셸일 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 `인수를` 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 `ParseIntPipe에` 대한 간단한 사용자 정의 구현을 보여드리겠습니다. 아래의 예제 기법은 다른 기본 제공 변환 파이프에도 적용됩니다.

(이 장에서는 `ParseBoolPipe`, `ParseFloatPipe`, `ParseEnumPipe`, `ParseArrayPipe` 및 ParseUUIDPipe를 `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 매개변수, 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)를 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 기술을 참조하세요.

맞춤형 파이프

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

용자 정의 버전을 처음부터 빌드해 보겠습니다.

간단한 `ValidationPipe부터` 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 T를 사용하여 입력 `값의` 유형을 나타내고 R을 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.

```
메타타입
유형
```

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

> 경고 TypeScript 인터페이스는 트랜슬레이션 중에 사라집니다. 따라서 메서드 매개변수의 유형이 클래스 대신 인터페이스로 선언된 경우 `메타타입` 값은 `Object가` 됩니다.

스키마 기반 유효성 검사

유효성 검사 파이프를 좀 더 유용하게 만들어 봅시다. 서비스 메서드를 실행하기 전에 포스트 본문 객체가 유효한지 확인해야 하는 `CatsController의 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 `CreateCatDto입니다`:

```
@@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 수행할 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 핸

들러와 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.숫자(), 품종:
    z.문자열(),
  })
  .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 `메타타입에` 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
  : 문자열;
}
```

> 정보 힌트 클래스 유효성 검사기 데코레이터에 대한 자세한 내용은 여기를 참조하세요.

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

```
@@파일명(유효성 검사.파이프)
import { PipeTransform, 인젝터블, 인자 메타데이터, BadRequestException
}를 '@nestjs/common'에서 가져옵니다;
'class-validator'에서 { validate }를 가져옵니다;
'class-transformer'에서 { plainToInstance }를 가져옵니다;
```

```
@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. 전체 기능과 예제에 대한 자세한 내용은 여기에서 확인할 수 있습니다.
>
> 경고 위의 클래스-변환기 라이브러리는 클래스-검증기 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 유형일 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

`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 `ValidationPipe는` 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 `ParseIntPipe입니다.` (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 파이프로 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. `DefaultValuePipe가 바로` 그 역할을 합니다. 아래 그림과 같이 관련 `Parse*` 파이프 앞에 `@Query()` 데코레이터에서 DefaultValuePipe를 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, `CanActivate를` 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 멍청합니다. `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 `AuthGuard는` 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 임포트
합니다;
'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 옵저버블을 통해)으로 반환할 수 있습니다. Nest는 반환값을 사용하여 다음 작업을 제어합니다:

- `참을` 반환하면 요청이 처리됩니다.• `거짓을` 반환하면 Nest는 요청을 거부합니다.

실행 컨텍스트

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

`ArgumentsHost를` 확장함으로써 `ExecutionContext는` 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 `ExecutionContext에` 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 임포트
합니다;
'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:

```
@@파일명()
const app = await NestFactory.create(AppModule);
app.useGlobalGuards(new RolesGuard());
```

> 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 임포트합니
다;

모듈({ providers:
  [
    {
      제공: APP_GUARD,
      useClass: RolesGuard,
    },
  ],
})
내보내기 클래스 AppModule {}
```

> 정보 힌트 이 접근 방식을 사용하여 가드에 대한 종속성 주입을 수행할 때, 이 구조가 사용되는 모듈에 관계없이 가드는 실제로 전역이라는 점에 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

`RolesGuard가` 작동하고 있지만 아직은 그다지 똑똑하지는 않습니다. 가장 중요한 가드 기능인 실행 컨텍스트를 아직 활용하지 못하고 있습니다. 아직 역할이나 각 핸들러에 허용되는 역할에 대해 알지 못합니다. 예를 들어, CatsController는 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 데코레이터

를 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 `RolesGuard와` 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 임포트
합니다;
'@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.user에` 사용자 인스턴스와 허용된 역할이 포함되어 있다고 가정합니다. 앱에서는 사용자 지정 인증 가드(또는 미들웨어)에서 이러한 연결을 만들 것입니다. 이 주제에 대한 자세한 내용은 이 장을 확인하세요.
>
> 결과 `matchRoles()` 함수 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 ForbiddenException을 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 반환된 결과를 변환합니다.
- 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 `Observable을` 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 Observable이 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,
    },
  ],
})
내보내기 클래스 AppModule {}
```

> 정보 힌트 이 접근 방식을 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 TransformInterceptor를 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 만드는 데 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 매핑

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 CacheInterceptor를 사용하는 엔드포인트를 호출하면 응답(하드코딩된 빈 배열)이 즉시 반환됩니다. 일반적인 솔루션을 만들기 위해 Reflector를 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 엔드포인트에서 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(), 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 `아무것이나` 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 실행됩니

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 파이프`는 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.