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 처리가 완료된 후 가비지 수집됩니다.
TRANSIENT	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 컨트롤러와* 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` 속성을 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`. `CatsService가` 요청 범위가 지정되어 있고 나머지는 기본 싱글톤인 경우, 주입된 서비스에 종속되어 있기 때문에 CatsController는 요청 범위가 지정됩니다. 종속적이지 않은 CatsRepository는 싱글톤 범위로 유지됩니다.

일시적 범위의 종속성은 이러한 패턴을 따르지 않습니다. 싱글톤 범위의 `DogsService가` 일시적인 `LoggerService` 프로바이더를 주입하면 새로운 인스턴스를 받게 됩니다. 그러나 DogsService는 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 `CONTEXT`를 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
클래스 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 임포트합
니다;
```

```
@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 지연 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 지연 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.

```
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;
  }
```

> 정보 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 거짓으로 설정되지 않은 경우).

> 경고 경고 이 전략은 많은 수의 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 tenantSubTreeId를 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 true로 설정하고 해당 범위를 Scope.REQUEST로 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/await을` 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 레시피에는 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 UsersService를 제공하고 내보낼 UsersModule을 정의하겠습니다. UsersModule은 UsersService의 호스트 모듈입니다.

```
'@nestjs/common'에서 { Module }을 가져오고,
'./users.service'에서 { UsersService }를 가져옵니
다;

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

다음으로, UsersModule을 가져와서 UsersModule의 내보낸 프로바이더를 AuthModule 내에서 사용할 수 있게 만드는 AuthModule을 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 { UsersService }를 가져옵니다;

@Injectable()
내보내기 클래스 AuthService {
  constructor(private usersService: UsersService) {}
  /*
    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 `UsersService`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 `UsersService` 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 모듈이* 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 `config라는` 폴더(즉, `src의` 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({ 폴더: './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 `DynamicModule입니다`. 동적 모듈은 런타임에 생성되는 모듈로, 정적 모듈과 동일한 프로퍼티에 `module이라는` 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 `module이라는` 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 컨피그모듈 {
  정적 register(): DynamicModule { return
    {
      모듈을 사용합니다: 컨피그모듈, 공
      급자: [컨피그서비스], 내보내기:
      [ConfigService],
    };
  }
}
```

이제 조각들이 어떻게 서로 연결되는지 명확해졌을 것입니다. `ConfigModule.register(...)`를 호출하면 지금까지 `@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 흥미롭지는 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({ 폴더: './config' })], 컨트
  롤러: [AppController],
  공급자: [앱서비스],
})
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 인젝터블 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, forFeature와 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 @nestjs/axios를 사용합니다: 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.
- forRoot를 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.
- forFeature의 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, forFeatureAsync가 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 ConfigModule이 받는 옵션을 나타내는 전용 인터페이스를 만들어 보겠습니다.

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

이렇게 하면 기존 `config.module.ts` 파일과 함께 새로운 전용 파일을 생성하고 이름을 `config.module-definition.ts`로 지정합니다. 이 파일에서 `ConfigurableModuleBuilder`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 `ConfigurableModuleClass`를 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 {}
```

`ConfigurableModuleClass`를 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({ 폴더: './config' }),
    // 또는 그 반대로:
    // ConfigModule.registerAsync({
    //useFactory     : () => {
           //return {
    //폴더    : './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

ConfigurableModuleClass는 기본적으로 register와 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 ConfigurableModuleBuilder가 forRoot를 노출하는 클래스를 생성하도록 지시합니다.

를 대신 사용할 수 있습니다. 예시:

```
모듈({ import: [
    ConfigModule.forRoot({ 폴더: './config' }), // <-- "register" 대신
"forRoot"를 사용했음에 유의하세요.
    // 또는 그 반대로:
    // ConfigModule.forRootAsync({
    //useFactory     : () => {
           //return {
    //폴더    : './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 구성을 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 `ConfigurableModuleBuilder에` 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 에지 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, 폴더:
      './config',
    }),
  ],
})
내보내기 클래스 AppModule {}
```

그러나 isGlobal은 "추가" 속성으로 선언되었기 때문에

`MODULE_옵션_토큰` 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 임포트합니
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 임포트하
고, './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 컨테이너에서 관리할 수 있는 클래스로 선언합니다.

. `cats.controller.ts`에서 `CatsController`는 생성자 주입을 통해 `CatsService` 토큰에 대한 종속성을 선언합니다:

```
생성자(private catsService: CatsService)
```

. app.module.ts에서 토큰 CatsService를 cats.service.ts 파일의 CatsService 클래스와 연결합니다. 아래에서 이 연결(*등록이라고도 함*)이 정확히 어떻게 발생하는지 살펴보겠습니다.

Nest IoC 컨테이너가 CatsController를 인스턴스화할 때, 먼저 모든 종속성*을 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.module에서 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 CatsService를 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_DEBUG를` 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 `CatsService를` 사용하도록 강제하고 싶다고 가정해 보겠습니다.

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

const mockCatsService = {
  /* 모의 구현
  ...
  */
};

모듈({
  임포트: [CatsModule], 제공자:
  [
    {
      제공: CatsService, useValue:
      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 `new로` 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 기초를 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 기존 연결에 연결합니다.

객체를 가져올 수 있습니다.

> 경고 토큰 값으로 문자열을 사용하는 것 외에도 JavaScript 기호 또는 TypeScript 열거형을 사용할 수도 있습니다.

앞서 표준 생성자 기반 주입 패턴을 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.ts와` 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 기능적으로는 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
  //          이 공급자
  //        는 필수입니다.                   공급자가 있는 공급자
};                                          토큰은 '정의되지 않음'으로 해석될 수 있습
                                            니다.

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
  //          이 공급자
  //        는 필수입니다.                   공급자가 있는 공급자
};                                          토큰은 '정의되지 않음'으로 해석될 수 있습
                                            니다.

모듈({ providers: [

    연결 제공자, 옵션 제공자,

    // { provide: 'SomeOptionalProvider', useValue: 'anything' },
  ],
})
내보내기 클래스 AppModule {}
```

별칭 공급자: 사용기존

`사용Existing` 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 `LoggerService의` 별칭입니다. `'AliasedLoggerService'에` 대한 `종속성과LoggerService에` 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 앨리어스 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 cats를 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` 유틸리티 클래스에 대해 자세히 알아보세요.

모듈 순방향 참조

모듈 간의 순환 종속성을 해결하려면 모듈 연결의 양쪽에서 동일한 `forwardRef()` 유틸리티 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. `ModuleRef는` 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. CatsService가 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 서버리스 환경에서 실행되는 앱/워커의 경우 병목 현상이 발생할 수 있습니다.

지연 로딩은 특정 서버리스 함수 호출에 필요한 모듈만 로드하여 부트스트랩 시간을 단축하는 데 도움이 될 수 있습니다. 또한 서버리스 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 훅 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.module을 "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 서버리스 함수/웹훅이 입력 인수(경로 경로/날짜/쿼리 매개변수 등)에 따라 다른 서비스(다른 로직)를 트리거해야 하는 상황에서 지연 로드된 모듈을 볼 수 있습니다. 반면에 지연 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 유틸리티 클래스를 제공합니다. 이러한 유틸리티는 현재 실행 컨텍스트에 대한 정보를 제공하여 광범위한 컨트롤러, 메서드 및 실행 컨텍스트에서 작동할 수 있는 일반 가드, 필터 및 인터셉터를 빌드하는 데 사용할 수 있습니다.

이 장에서는 이러한 클래스 두 가지를 다룹니다: ArgumentsHost와 `ExecutionContext입니다`.

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 `ArgumentsHost의` 인스턴스를 사용자가 액세스하려는 위치에 제공합니다. 예를 들어 예외 필터의 `catch()` 메서드는 `ArgumentsHost 인스턴스와` 함께 호출됩니다.

`ArgumentsHost는` 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 인터셉터를 빌드할 때는 메서드가 현재 실행 중인 애플리케이션 유형을 확인할 수 있는 방법이 필요합니다. 이 작업은 `ArgumentsHost의 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 유틸리티 메서드 중 하나를 사용하여 애플리케이션에 적합한 애플리케이션 컨텍스트로 전환함으로써 코드를 보다 강력하고 재사용 가능하게 만들 수 있습니다. 컨텍스트 전환 유틸리티 메서드는 다음과 같습니다.

```
/**
 * 컨텍스트를 RPC로 전환합니다.
 */
switchToRpc(): RpcArgumentsHost;
/**
 * 컨텍스트를 HTTP로 전환합니다.
 */
switchToHttp(): HttpArgumentsHost;
/**
 * 컨텍스트를 웹소켓으로 전환합니다.
 */
switchToWs(): WsArgumentsHost;
```

`스위치투햇트()` 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 WsArgumentsHost와 `RpcArgumentsHost에는` 마이크로서비스 및 웹 소켓 컨텍스트에서 적절한 객체를 반환하는 메서드가 있습니다. 다음은 `WsArgumentsHost에` 대한 메서드입니다:

```
내보내기 인터페이스 WsArgumentsHost {
  /**
   * 데이터 객체를 반환합니다.
```

```
   */
  getData<T>(): T;
  /**
   * 클라이언트 객체를 반환합니다.
   */
  getClient<T>(): T;
}
```

다음은 `RpcArgumentsHost의` 메서드입니다:

```
내보내기 인터페이스 RpcArgumentsHost {
  /**
   * 데이터 객체를 반환합니다.
   */
  getData<T>(): T;

  /**
   * 컨텍스트 객체를 반환합니다.
   */
  getContext<T>(): T;
}
```

실행 컨텍스트 클래스

`ExecutionContext는 ArgumentsHost를` 확장하여 현재 실행 프로세스에 대한 추가 세부 정보를 제공합니다. `ArgumentsHost와` 마찬가지로 Nest는 가드의 `canActivate()` 메서드나 인터셉터의 `intercept()` 메서드 등 필요할 수 있는 곳에 ExecutionContext의 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 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.

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 `Roles입니다` (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 유틸리티 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#createDecorator를 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(roles는 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 @SetMetadata를 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 훅을 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 훅 메서드를 호출합니다(아래 설명된 대로 셧다운 훅을 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 onApplicationShutdown은 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
`onModuleInit()`	호스트 모듈의 종속성이 다음과 같이 설정되면 호출됩니다.
`onApplicationBootstrap()`	`beforeApplicationShutdown()`*
`onModuleDestroy()`*	

해결되었습니다.	종료 신호(예: `SIGTERM`)가 수신된 후 호출됩니다.
모든 모듈이 초기화되면 호출되지만 연결을 수신 대기하기 전에 호출됩니다.	모든 `onModuleDestroy()` 핸들러가 완료된 후 호출됩니다(프로미스가 해결되거나 거부된 경우); 완료되면(프로미스가 해결되거나 거부되면) 기존의 모든 연결이 닫힙니다(`app.close()` 호출).
`onApplicationShutdown()`*	연결이 닫힌 후 호출됩니다(`app.close()` 해결).

* 이러한 이벤트의 경우 `app.close()`를 명시적으로 호출하지 않는 경우, `SIGTERM`과 같은 시스템 신호와 함께 작동하도록 옵트인해야 합니다. 아래 애플리케이션 종료를 참조하세요.

> 경고 경고 위에 나열된 라이프사이클 훅은 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 훅을 기다리는 모듈 가져오기 순서에 직접적으로 의존합니다.

사용법

각 라이프사이클 훅은 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 훅을 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 UsersService 구현 온 모듈 초기화 { onModuleInit() {
    console.log(`모듈이 초기화되었습니다.`);
  }
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트합니다;

@Injectable()
내보내기 클래스 UsersService {
  onModuleInit() {
    console.log(`모듈이 초기화되었습니다.`);
  }
}
```

비동기 초기화

OnModuleInit 및 OnApplicationBootstrap 훅을 사용하면 애플리케이션 초기화 프로세스를 지연시킬 수 있습니다(프로미스를 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()` 훅은 종료 단계에서 호출됩니다(`app.close()`) 명시적 호출에 대한 응답 또는 옵트인한 경우 SIGTERM과 같은 시스템 신호를 수신할 때). 이 기능은 컨테이너의 라이프사이클을 관리하기 위해 Kubernetes와 함께 자주 사용되며, Heroku는 다이노 또는 이와 유사한 서비스를 위해 사용합니다.

셧다운 후크 리스너는 시스템 리소스를 소모하므로 기본적으로 비활성화되어 있습니다. `셧다운` 훅을 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);

  // 셧다운 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 SIGINT, SIGBREAK [illegible] Node의 관련 문제 [illegible] NestJS 앱 [illegible] 프로세스 신호 이벤트에 [illegible] Node.js 실행 [illegible] Node에서 과도한 리스너 프로세스에 대해 불만을 제기할 수 있습니다. 이러한 이유로 `enableShutdownHooks는` 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()
```

```
UsersService 클래스는 OnApplicationShutdown을 구현합니다 {
  onApplicationShutdown(신호: 문자열) {
    console.log(signal); // 예: "SIGINT"
  }
}
@@스위치
@Injectable()
UsersService 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.
- 는 테스트 도구에 구애받지 않고 Jest 및 Supertest와의 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: CatsController와 `CatsService를 테스트합니다`. 앞서 언급했듯이 Jest는 기본 테스트 프레임워크로 제공됩니다. 테스트 실행자 역할을 하며 모킹, 스파이 등에 도움이 되는 어설트 함수와 테스트-더블 유틸리티도 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(CatsService의 인스턴스를 `catsController에` 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 유틸리티

`nestjs/testing` 패키지는 보다 강력한 테스트 프로세스를 가능하게 하는 일련의 유틸리티를 제공합니다. 내장된 `Test` 클래스를 사용하여 이전 예제를 다시 작성해 보겠습니다:

```
@@파일명(cats.controller.spec)
'@nestjs/testing'에서 { Test }를 가져옵니
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);
  });
```

```
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 훅을 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((토큰) => {
        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-jest의 createMock과` 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 임포트합
니다;
'../../src/cats/cats.module'에서 { CatsModule }을 임포트하고,
'../../src/cats/cats.service'에서 { CatsService }를 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`, () => {
    반환 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 임포트합
니다;
'../../src/cats/cats.module'에서 { CatsModule }을 임포트하고,
'../../src/cats/cats.service'에서 { CatsService }를 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`, () => {
    반환 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 `앱` 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 `CatsService의` 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:

`createNestApplication()`	주어진 모듈을 기반으로 네스트 애플리케이션(`INestApplication` 인스턴스)을 생성하고 반환합니다. `init()` 메서드를 사용하여 애플리케이션을 수동으로 초기화해야 한다는 점에 유의하세요.
`createNestMicroservice()`	주어진 모듈을 기반으로 Nest 마이크로서비스(`INestMicroservice` 인스턴스)를 생성하고 반환합니다.
`get()`	애플리케이션 컨텍스트에서 사용할 수 있는 컨트롤러 또는 공급자(가드, 필터 등 포함)의 정적 인스턴스를 검색합니다. 모듈 참조 클래스에서 상속됩니다.
`resolve()`	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.
`select()`	모듈의 종속성 그래프를 탐색하고, 선택한 모듈에서 특정 인스턴스를 검색하는 데 사용할 수 있습니다(`get()` 메서드에서 엄격 모드(`strict: true`)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

`JwtAuthGuard`를 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 JwtAuthGuard는 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. 요

청 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);
```