

읽기-값-출력-루프(REPL)

REPL은 단일 사용자 입력을 받아 실행하고 결과를 사용자에게 반환하는 간단한 대화형 환경입니다. REPL 기능을 사용하면 터미널에서 직접 공급자(및 컨트롤러)의 의존성 그래프와 호출 메서드를 검사할 수 있습니다.

사용법

REPL 모드에서 NestJS 애플리케이션을 실행하려면 기존 `main.ts`와 함께 새 `repl.ts` 파일을 생성합니다. 파일을 열고 그 안에 다음 코드를 추가합니다:

```
@@파일명 (답글)

'@nestjs/core'에서 { repl } 임포트;
'./app.module'에서 { AppModule } 임포트;

비동기 함수 bootstrap() { await
  repl(AppModule);
}
부트스트랩(); @@스

위치

'@nestjs/core'에서 { repl } 임포트;
'./app.module'에서 { AppModule } 임포트;

비동기 함수 bootstrap() { await
  repl(AppModule);
}
부트스트랩();
```

이제 터미널에서 다음 명령어로 REPL을 시작합니다:

```
$ npm 실행 시작 -- --entryFile repl
```

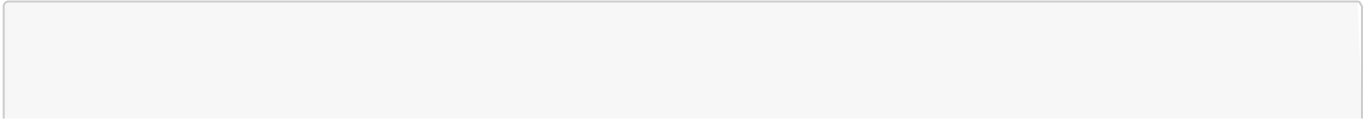
정보 힌트 repl은 [Node.js REPL 서버](#) 객체를 반환합니다.

실행이 완료되면 콘솔에 다음 메시지가 표시됩니다:

```
LOG [NestFactory] Nest 애플리케이션을 시작하는 중...
LOG [InstanceLoader] 앱모듈 종속성 초기화됨 LOG REPL 초기화됨
```

이제 종속성 그래프와 상호작용을 시작할 수 있습니다. 예를 들어

`AppService`(여기서는 스타터 프로젝트를 예로 사용합니다)를 열고 `getHello()` 메서드를 호출합니다:



```
> get(AppService).getHello()
'Hello World!'
```

예를 들어 터미널 내에서 자바스크립트 코드를 실행할 수 있습니다.

`AppController`를 로컬 변수로 설정하고 `await`을 사용하여 비동기 메서드를 호출합니다:

```
> appController = get(AppController)
AppController { appService: AppService {} }
> await appController.getHello()
'Hello World!'
```

지정된 공급자 또는 컨트롤러에서 사용 가능한 모든 공개 메서드를 표시하려면 다음과 같이 `methods()` 함수를 사용합니다:

```
> 메소드(앱 컨트롤러) 메소드:
  □ getHello
```

등록된 모든 모듈을 컨트롤러 및 프로바이더와 함께 목록으로 인쇄하려면 `debug()`를 사용합니다.

```
> 디버그()

앱모듈:
  - 컨트롤러:
    □ 앱 컨트롤러
  - 제공업체:
    □ 앱 서비스
```

빠른 데모:



REPL 예제

미리 정의된 기존 네이티브 메서드에 대한 자세한 내용은 아래 섹션에서 확인할 수 있습니다. 네이티브 함

수

기본 제공 NestJS REPL에는 REPL을 시작할 때 전역적으로 사용할 수 있는 몇 가지 기본 함수가 포함되어 있습니다.

`help()`를 호출하여 나열할 수 있습니다.

함수의 시그니처(예: 예상 매개변수 및 반환 유형)가 무엇인지 기억나지 않는 경우 `<function_name>.help`를 호출할 수 있습니다. 예를 들어

```
> .help
```

인젝터블 또는 컨트롤러의 인스턴스를 검색하고, 그렇지 않으면 예외를 던집니다.

인터페이스: \$(토큰: InjectionToken) => any

정보 힌트 이러한 함수 인터페이스는 [TypeScript 함수 유형 표현식 구문](#)으로 작성됩니다.

기능	설명	서명
<code>debug</code>	등록된 모든 모듈을 컨트롤러 및 공급자와 함께 목록으로 인쇄합니다.	<code>debug(moduleCls?: ClassRef \ 문자열) => void</code>
<code>get</code> 또는 <code>\$ 메서드</code>	인젝터블 또는 컨트롤러의 인스턴스를 검색하고, 그렇지 않으면 예외를 던집니다.	<code>get(token: InjectionToken) => any</code>
<code>resolve</code>	지정된 공급자 또는 컨트롤러에서 사용 가능한 모든 공개 메소드를 표시합니다.	<code>resolve(token: InjectionToken, contextId: any) => Promise<any></code>
<code>select</code>	인젝터블 또는 컨트롤러의 일시적이거나 요청 범위가 지정된 인스턴스를 해결하고, 그렇지 않으면 예외를 던집니다.	<code>select(token: DynamicModule \ ClassRef) => INestApplicationContext</code>
<code>resolveAndSelect</code>	예를 들어 모듈 트리를 탐색하여 선택한 모듈에서 특정 인스턴스를 가져올 수 있습니다.	

시계 모드

개발 중에는 모든 코드 변경 사항을 자동으로 반영하기 위해 감시 모드에서 REPL을 실행하는 것이 유용합니다:

```
$ npm 실행 시작 -- --watch --entryFile repl
```

여기에는 한 가지 결함이 있는데, REPL의 명령 기록은 새로로드할 때마다 삭제되므로 번거로울 수 있습니다. 다행히도 아주 간단한 해결책이 있습니다. [부트스트랩](#) 함수를 다음과 같이 수정하면 됩니다:

```
비동기 함수 부트스트랩() {  
  const replServer = await repl(AppModule);  
  replServer.setupHistory(".nestjs_repl_history", (err) => {  
    if (err) {  
      console.error(err);  
    }  
  });  
}
```

이제 실행/재로드 사이에 기록이 보존됩니다.

CRUD 생성기

프로젝트의 수명 기간 동안 새로운 기능을 구축할 때 애플리케이션에 새로운 리소스를 추가해야 하는 경우가 많습니다. 이러한 리소스에는 일반적으로 새 리소스를 정의할 때마다 반복해야 하는 여러 번의 반복 작업이 필요합니다.

소개

사용자 및 제품 엔티티라는 두 개의 엔티티에 대한 CRUD 엔드포인트를 노출해야 하는 실제 시나리오를 가정해 보겠습니다. 모범 사례에 따라 각 엔티티에 대해 다음과 같이 몇 가지 작업을 수행해야 합니다:

- 모듈을 생성하여 코드를 체계적으로 정리하고 명확한 경계를 설정(관련 컴포넌트 그룹화)하세요.
- CRUD 경로(또는 GraphQL 애플리케이션의 경우 쿼리/변형)를 정의하는 컨트롤러(`nest g co`)를 생성합니다.
- 비즈니스 로직을 구현하고 격리하기 위한 서비스 생성(네스트 `지에스`)
- 리소스 데이터 형태를 나타내는 엔티티 클래스/인터페이스 생성
- 데이터 전송 개체(또는 GraphQL 애플리케이션의 경우 입력)를 생성하여 네트워크를 통해 데이터를 전송하는 방법을 정의합니다.

정말 많은 단계입니다!

`Nest CLI`는 이러한 반복적인 프로세스의 속도를 높이기 위해 모든 상용구 코드를 자동으로 생성하는 제너레이터(회로도)를 제공하여 이 모든 작업을 피하고 개발자 환경을 훨씬 더 단순하게 만듭니다.

정보 참고 이 스키마는 HTTP 컨트롤러, 마이크로서비스 컨트롤러, GraphQL 리졸버(코드 우선 및 스키마 우선 모두) 및 웹소켓 게이트웨이 생성을 지원합니다.

새 리소스 생성

새 리소스를 만들려면 프로젝트의 루트 디렉터리에서 다음 명령을 실행하면 됩니다:

```
nest g resource
```

`nest g resource` 명령은 모든 NestJS 빌딩 블록(모듈, 서비스, 컨트롤러 클래스)뿐만 아니라 엔티티 클래스,

DTO 클래스, 테스트(.spec) 파일도 생성합니다.

아래에서 생성된 컨트롤러 파일(REST API용)을 확인할 수 있습니다:

```
컨트롤러('users')  
사용자 컨트롤러 클래스 내보내기 {  
  생성자(비공개 읽기 전용 userService: UsersService) {}  
  
  @Post()  
  create(@Body() createUserDto: CreateUserDto) {
```



```

        이.userService.create(createUserDto)를 반환합니다;
    }

    @Get()
    findAll() {
        this.userService.findAll()을 반환합니다;
    }

    @Get('/:id')
    findOne(@Param('id') id: 문자열) { return
        this.userService.findOne(+id);
    }

    @Patch('/:id')
    update(@Param('id') id: 문자열, @Body() updateUserDto: UpdateUserDto) {
        return this.userService.update(+id, updateUserDto);
    }

    삭제('/:id') remove(@Param('id')
    id: 문자열) {
        this.userService.remove(+id)를 반환합니다;
    }
}

```

또한 손가락 하나 까딱하지 않고도 모든 CRUD 엔드포인트(REST API의 경우 경로, GraphQL의 경우 쿼리 및 변이, 마이크로서비스 및 웹소켓 게이트웨이의 경우 메시지 구독)에 대한 자리 표시자를 자동으로 생성합니다.

경고 생성된 서비스 클래스는 특정 ORM(또는 데이터 소스)에 연결되지 않습니다. 따라서 생성기는 모든 프로젝트의 요구 사항을 충족할 수 있을 만큼 충분히 일반적입니다. 기본적으로 모든 메서드에는 프로젝트에 특정한 데이터 소스로 채울 수 있는 자리 표시자가 포함됩니다.

마찬가지로 GraphQL 애플리케이션용 리졸버를 생성하려면 전송 계층으로 GraphQL(코드 우선)(또는 GraphQL(스키마 우선))을 선택하기만 하면 됩니다.

이 경우 NestJS는 REST API 컨트롤러 대신 리졸버 클래스를 생성합니다:

nest g 리소스 사용자

- > ? 어떤 전송 계층을 사용하시나요? GraphQL(코드 우선)
- > ? CRUD 진입점을 생성하시겠습니까? 예
- > CREATE src/users/users.module.ts (224바이트)
- > src/users/users.resolver.spec.ts 생성 (525 바이트)
- > src/users/users.resolver.ts 생성(1109바이트)
- > CREATE src/users/users.service.spec.ts (453바이트)
- > CREATE src/users/users.service.ts (625바이트)
- > CREATE src/users/dto/create-user.input.ts (195 바이트)
- > src/users/dto/update-user.input.ts 생성 (281바이트)
- > CREATE src/users/entities/user.entity.ts (187 바이트)
- > src/app.module.ts 업데이트 (312바이트)

정보 힌트 테스트 파일을 생성하지 않으려면 다음과 같이 `--no-spec` 플래그를 전달할 수 있습니다: `nest g resource users --no-spec`

아래에서 모든 상용구 변이와 쿼리가 생성되었을 뿐만 아니라 모든 것이 서로 연결되어 있음을 알 수 있습니다. 우리는 사용자 서비스, 사용자 엔티티, DTO를 활용하고 있습니다.

```
'@nestjs/graphql'에서 { Resolver, Query, Mutation, Args, Int }를 가져오고,
'./users.service'에서 { UsersService }를 가져옵니다;
'./entities/user.entity'에서 { 사용자 }를 가져옵니다;
'./dto/create-user.input'에서 { CreateUserInput } 가져오기;
'./dto/update-user.input'에서 { UpdateUserInput } 가져오기;

@Resolver(() => 사용자)
사용자 해결자 클래스 내보내기 {
  생성자(비공개 읽기 전용 usersService: UsersService) {}

  @Mutation(() => 사용자)
  createUser(@Args('createUserInput') createUserInput: CreateUserInput) {
    return this.usersService.create(createUserInput);
  }

  @Query(() => [User], { name: 'users' })
  findAll() {
    this.usersService.findAll()을 반환합니다;
  }

  쿼리(() => 사용자, { 이름: '사용자' })
  findOne(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.findOne(id);
  }

  @Mutation(() => 사용자)
  updateUser(@Args('updateUserInput') updateUserInput: UpdateUserInput) {
    return this.usersService.update(updateUserInput.id, updateUserInput);
  }

  @Mutation(() => 사용자)
  removeUser(@Args('id', { type: () => Int }) id: number) {
    return this.usersService.remove(id);
  }
}
```

SWC

SWC(Speedy Web Compiler)는 컴파일과 번들링에 모두 사용할 수 있는 확장 가능한 Rust 기반 플랫폼입니다. Nest CLI와 함께 SWC를 사용하면 개발 프로세스의 속도를 크게 높일 수 있는 간단하고 훌륭한 방법입니다.

정보 힌트 SWC는 기본 TypeScript 컴파일러보다 약 20배 빠릅니다.

설치

시작하려면 먼저 몇 가지 패키지를 설치하세요:

```
$ npm i --save-dev @swc/cli @swc/core
```

시작하기

설치 프로세스가 완료되면 다음과 같이 Nest CLI와 함께 **swc** 빌더를 사용할 수 있습니다:

```
nest start -b swc  
# OR nest start --builder swc
```

정보 힌트 리포지토리가 모노레포인 경우 [이 섹션](#)을 확인하세요.

b 플래그를 전달하는 대신 다음과 같이 **nest-cli.json** 파일에서 **compilerOptions.builder** 속성을 **"swc"**로 설정할 수도 있습니다:

```
{  
  "컴파일러옵션": { "빌더":  
    "swc"  
  }  
}
```

빌더의 동작을 사용자 정의하려면 두 가지 속성이 포함된 객체, **유형("swc")**과 옵션은 다음과 같습니다:

```
"컴파일러옵션": { "빌더": {  
  "유형": "SWC", "옵션"  
  ": {  
    "swcrcPath": "인프라/.swcrc",  
  }  
}  
}
```

감시 모드에서 애플리케이션을 실행하려면 다음 명령을 사용합니다:

```
nest start -b swc -w
# 또는 nest start --builder swc --watch
```

유형 검사

SWC는 기본 타입스크립트 컴파일러와 달리 자체적으로 타입 검사를 수행하지 않으므로 이를 켜려면 `--type-check` 플래그를 사용해야 합니다:

```
nest start -b swc --type-check
```

이 명령은 Nest CLI가 유형 검사를 비동기적으로 수행하는 SWC와 함께 `noEmit` 모드에서 `tsc`를 실행하도록 지시합니다. 다시 말하지만, `--type-check` 플래그를 전달하는 대신 다음과 같이 `nest-cli.json` 파일에서 `compilerOptions.typeCheck` 속성을 `true`로 설정할 수도 있습니다:

```
{
  "컴파일러옵션": { "빌더":
    "swc", "typeCheck": true
  }
}
```

CLI 플러그인(SWC)

`type-check` 플래그를 사용하면 NestJS CLI 플러그인을 자동으로 실행하고 직렬화된 메타데이터 파일을 생성하여 런타임에 애플리케이션에서 로드할 수 있습니다.

SWC 구성

SWC 빌더는 NestJS 애플리케이션의 요구 사항에 맞게 사전 구성되어 있습니다. 그러나 루트 디렉터리에 `.swcrc` 파일을 생성하고 원하는 대로 옵션을 조정하여 구성을 사용자 지정할 수 있습니다.

```
{
  "$schema": "https://json.schemastore.org/swcrc",
  "sourceMaps": true,
  "jsc": {
    "파서": {
      "구문": "typescript",
      "decorators": true,
      "dynamicImport": true
    },
    "baseUrl": "./"
  }
}
```

```
  },
  "minify": false
}
```

모노레포

리포지토리가 모노레포인 경우 `swc` 빌더를 사용하는 대신 `웹팩`을 사용하도록 구성해야 합니다.
`swc-loader`.

먼저 필요한 패키지를 설치해 보겠습니다:

```
$ npm i --save-dev swc-loader
```

설치가 완료되면 애플리케이션의 루트 디렉터리에 다음 내용으로 `webpack.config.js` 파일을 생성합니다:

```
const swcDefaultConfig = require('@nestjs/cli/lib/compiler/defaults/swc-
defaults').swcDefaultsFactory().swcOptions;

module.exports = {
  module: {
    규칙: [
      {
        test: /\.ts$/,
        제외합니다: /node_modules/, 사
        용: {
          로더: 'swc-loader', 옵션:
            swcDefaultConfig,
        },
      },
    ],
  },
};
```

모노레포 및 CLI 플러그인

이제 CLI 플러그인을 사용하는 경우 `swc-loader`가 자동으로 로드하지 않습니다. 대신 수동으로 로드할 별도의 파일을 만들어야 합니다. 이렇게 하려면 `main.ts` 파일 근처에 `생성 메타데이터.ts` 파일을 다음 내용으로 선언합니다:


```
'@nestjs/cli/lib/compiler/plugins'에서 {  
PluginMetadataGenerator }를 임포트합니다;  
'@nestjs/swagger/dist/plugin'에서 { ReadonlyVisitor }를 가져옵니다;  
  
const generator = new PluginMetadataGenerator();  
generator.generate({
```

```
방문자: [new ReadonlyVisitor({ introspectComments: true, pathToSource:
__dirname })],
outputDir: __dirname,
watch: true,
tsconfigPath: 'apps/<이름>/tsconfig.app.json',
});
```

정보 힌트 이 예제에서는 `@nestjs/swagger` 플러그인을 사용했지만 원하는 플러그인을 사용할 수 있습니다.

`generate()` 메서드는 다음 옵션을 허용합니다:

<code>watch</code>	프로젝트의 변경 사항을 주시할지 여부입니다.
<code>tsconfigPath</code>	<code>tsconfig.json</code> 파일의 경로입니다. 현재 작업 디렉터리를 기준으로 합니다. (<code>process.cwd()</code>).
<code>outputDir</code>	메타데이터 파일이 저장될 디렉터리 경로입니다. 메타데이터를
생성하는 데 사용할 방문자 배열입니다. <code>파일 이름</code> 메타데이터 파일의 이름입니다. 기	

본값은 `metadata.ts`입니다. `printDiagnostics` 콘솔에 진단을 인쇄할지 여부입니다

. 기본값은 `true`입니다.

마지막으로 다음 명령을 사용하여 별도의 터미널 창에서 `생성-메타데이터` 스크립트를 실행할 수 있습니다:

```
$ npx ts-node src/generate-metadata.ts
# 또는 npx ts-node apps/{YOUR_APP}/src/generate-metadata.ts
```

일반적인 함정

애플리케이션에서 TypeORM/MikroORM 또는 다른 ORM을 사용하는 경우 순환 가져오기 문제가 발생할 수 있습니다. SWC는 순환 가져오기를 잘 처리하지 못하므로 다음 해결 방법을 사용해야 합니다:

엔티티()

사용자 클래스 보내기 {

@OneToMany(() => 프로필, (프로필) => 프로필.사용자)

프로필: Relation<Profile>; // <--- 여기서는 "Profile" 대신 "Relation<>" 유형
을 참조하십시오.

정보 힌트 관계 유형은 `typeorm` 패키지에서 내보냅니다.

이렇게 하면 프로퍼티 메타데이터의 트랜스파일 코드에 프로퍼티 유형이 저장되지 않으므로 순환 종속성 문제를 방지할 수 있습니다.

ORM에서 유사한 해결 방법을 제공하지 않는 경우 래퍼 유형을 직접 정의할 수 있습니다:

```
/**
 * ESM 모듈 순환 종속성 문제를 우회하는 데 사용되는 래퍼 유형
 * 속성 유형을 저장하는 리플렉션 메타데이터로 인해 발생합니다.
 */
export type WrapperType<T> = T; // WrapperType === Relation
```

프로젝트의 모든 [순환 종속성 주입](#)에 대해서도 위에서 설명한 사용자 정의 래퍼 유형을 사용해야 합니다:

```
@Injectable()
내보내기 클래스 UserService { 생
  성자(
    @Inject(forwardRef(() => ProfileService))
    비공개 읽기 전용 profileService: 래퍼 유형<프로필 서비스>,
  ) {};
```

제스트 + SWC

Jest에서 SWC를 사용하려면 다음 패키지를 설치해야 합니다:

```
$ npm i --save-dev jest @swc/core @swc/jest
```

설치가 완료되면 구성에 따라 `package.json/jest.config.js` 파일을 다음 내용으로 업데이트합니다:

```
{
  "농담": {
    "transform": {
      "^.+\\. (t|j)s?$": ["@swc/jest"]
    }
  }
}
```

또한 `.swcrc` 파일에 다음 `트랜스폼` 속성을 추가해야 합니다:

`레거시 데코레이터`, `데코레이터 메타데이터`:

```
{  
  "$schema": "https://json.schemastore.org/swcrc",  
  "sourceMaps": true,  
  "jsc": {
```

```

"파서": {
  "구문": "typescript",
  "decorators": true,
  "dynamicImport": true
},
"transform": {
  "legacyDecorator": true,
  "decoratorMetadata": true
},
"baseUrl": "./"
},
"minify": false
}

```

프로젝트에서 NestJS CLI 플러그인을 사용하는 경우 [플러그인 메타데이터 생성기](#)를 수동으로 실행해야 합니다. 자세한 내용을 알아보려면 [이 섹션으로](#) 이동하세요.

Vitest

Vitest는 Vite와 함께 작동하도록 설계된 빠르고 가벼운 테스트 러너입니다. NestJS 프로젝트와 통합할 수 있는 현대적이고 빠르며 사용하기 쉬운 테스트 솔루션을 제공합니다.

설치

시작하려면 먼저 필요한 패키지를 설치하세요:

```
$ npm i --save-dev vitest unplugin-swc @swc/core @vitest/coverage-c8
```

구성

애플리케이션의 루트 디렉터리에 다음 내용으로 `vitest.config.ts` 파일을 생성합니다:

```
'unplugin-swc'에서 SWC를 가져옵니다;
'vite.config'에서 { defineConfig }를 가져옵니다;

export default defineConfig({
  test: {
    globals: true,
    root: './',
  },
  plugins: [
    // SWC로 테스트 파일을 빌드하는 데 필요합니다 swc.vite({
    // 모듈 유형을 명시적으로 설정하여 `.swcrc` 구성 파일에서 이 값을 상속하지 않도록 합니다.
    // 모듈을 추가합니다: { 유형: 'es6' },
  ]),
```

```
  ],
});
```

이 구성 파일은 Vitest 환경, 루트 디렉토리 및 SWC 플러그인을 설정합니다. 또한 테스트 경로 정규식을 지정하는 추가 **포함** 필드를 사용하여 e2e 테스트를 위한 별도의 구성 파일을 만들어야 합니다:

```
'unplugin-swc'에서 SWC를 가져옵니다;
'vitest/config'에서 { defineConfig }를 가져옵니다;

export default defineConfig({
  test: {
    포함: ['**/*.e2e-spec.ts'], globals:
    true,
    root: './',
  },
  플러그인: [swc.vite()],
});
```

또한 테스트에서 TypeScript 경로를 지원하도록 **별칭** 옵션을 설정할 수 있습니다:

```
'unplugin-swc'에서 SWC를 가져옵니다;
'vitest/config'에서 { defineConfig }를 가져옵니다;

export default defineConfig({
  test: {
    포함: ['**/*.e2e-spec.ts'], globals:
    true,
    별칭: {
      '@src': './src',
      '@test': './test',
    },
    root: './',
  },
  해결합니다:
  { alias:
    {
      '@src': './src',
      '@test': './test',
    },
  },
  플러그인: [swc.vite()],
});
```


E2E 테스트에서 가져오기 업데이트

`import *`를 사용하는 모든 E2E 테스트 가져오기를 `'supertest'`의 요청으로 가져오기 요청으로 변경합니다. 이는 Vite와 번들로 제공되는 경우 Vitest가 기본 가져오기를 기대하기 때문에 필요합니다.

를 사용하세요. 네임스페이스 가져오기를 사용하면 이 특정 설정에서 문제가 발생할 수 있습니다. 마지막으로 package.json 파일의 테스트 스크립트를 다음과 같이 업데이트합니다:

```
{
  "스크립트": {
    "테스트": "바이테스트 실행",
    "test:watch": "vitest",
    "test:cov": "VITEST RUN --COVERAGE",
    "test:debug": "vitest --inspect-brk --inspect --logHeapUsage --threads=false",
    "test:e2e": "vitest run --config ./vitest.config.e2e.ts"
  }
}
```

이 스크립트는 테스트 실행, 변경 사항 감시, 코드 커버리지 보고서 생성 및 디버깅을 위해 Vitest를 구성합니다.

test:e2e 스크립트는 특히 사용자 지정 구성 파일로 E2E 테스트를 실행하기 위한 스크립트입니다.

이 설정을 통해 이제 테스트 실행 속도 향상 및 최신 테스트 환경 등 NestJS 프로젝트에서 Vitest를 사용하는 이점을 누릴 수 있습니다.

정보 힌트 이 [리포지토리](#)에서 작동하는 예제를 확인할 수 있습니다.

여권(인증)

Passport는 커뮤니티에서 잘 알려져 있고 많은 프로덕션 애플리케이션에서 성공적으로 사용되는 가장 인기 있는 node.js 인증 라이브러리입니다. 이 라이브러리를 Nest 애플리케이션과 통합하는 방법은 `@nestjs/passport` 모듈을 사용하면 간단합니다. 높은 수준에서 Passport는 다음과 같은 일련의 단계를 실행합니다:

- 사용자 '자격 증명'(예: 사용자 이름/비밀번호, JSON 웹 토큰(JWT) 또는 ID 공급자의 ID 토큰)을 확인하여 사용자를 인증합니다.
- 인증 상태 관리(JWT와 같은 휴대용 토큰을 발급하거나 Express 세션을 만들어서)
- 인증된 사용자에 대한 정보를 요청 개체에 첨부하여 라우트 핸들러에서 추가로 사용할 수 있습니다.

Passport에는 다양한 인증 메커니즘을 구현하는 풍부한 전략 생태계가 있습니다. 개념은 간단하지만, 선택할 수 있는 Passport 전략의 집합은 매우 크고 다양합니다. Passport는 이러한 다양한 단계를 표준 패턴으로 추상화하며, `@nestjs/passport` 모듈은 이 패턴을 익숙한 Nest 구조로 래핑하고 표준화합니다.

이 장에서는 이러한 강력하고 유연한 모듈을 사용하여 RESTful API 서버를 위한 완전한 엔드투엔드 인증 솔루션을 구현해 보겠습니다. 여기에 설명된 개념을 사용하여 인증 체계를 사용자 지정하기 위해 모든 Passport 전략을 구현할 수 있습니다. 이 장의 단계에 따라 이 완전한 예제를 구축할 수 있습니다.

인증 요구 사항

요구 사항을 구체화해 봅시다. 이 사용 사례에서 클라이언트는 사용자 이름과 비밀번호로 인증하는 것으로 시작합니다. 인증이 완료되면 서버는 인증을 증명하기 위해 후속 요청 시 인증 헤더에 베어러 토큰으로 전송할 수 있는 JWT를 발급합니다. 또한 유효한 JWT가 포함된 요청만 액세스할 수 있는 보호된 경로를 생성합니다.

첫 번째 요구 사항인 사용자 인증부터 시작하겠습니다. 그런 다음 JWT를 발행하여 이를 확장합니다. 마지막으로 요청에 대해 유효한 JWT를 검사하는 보호된 경로를 생성합니다.

먼저 필요한 패키지를 설치해야 합니다. Passport는 사용자 이름/비밀번호 인증 메커니즘을 구현하는 `passport-local`이라는 전략을 제공하므로 이 사용 사례의 요구 사항에 적합합니다.

```
$ npm install --save @nestjs/passport passport-local  
$ npm install --save-dev @types/passport-local
```

경고 주의 어떤 패스포트 전략을 선택하든 항상 `@nestjs/passport` 및 패스포트 패키지가 필요합니다. 그런 다음 구축하려는 특정 인증 전략을 구현하는 전략별 패키지(예: `passport-jwt` 또는 `passport-local`)를 설치해야 합니다. 또한 위에 표시된 것처럼 `@types/passport-local`을 사용하여 모든 Passport 전략에 대한 유형 정의를 설치하면 TypeScript 코드를 작성하는 동안 도움을 받을 수 있습니다.

패스포트 전략 구현

이제 인증 기능을 구현할 준비가 되었습니다. 먼저 Passport 전략에 사용되는 프로세스에 대한 개요부터 살펴보겠습니다. Passport는 그 자체로 하나의 미니 프레임워크라고 생각하면 도움이 됩니다. 이 프레임워크의 장점은 인증 프로세스를 구현하는 전략에 따라 사용자 지정할 수 있는 몇 가지 기본 단계로 추상화한다는 것입니다. 사용자 지정 매개변수(일반 JSON 객체)와 콜백 함수 형태의 사용자 지정 코드를 제공하여 구성하고, Passport가 적절한 시점에 이를 호출하기 때문에 프레임워크와 비슷합니다. 이 프레임워크를 Nest 스타일 패키지로 감싸는 `@nestjs/passport` 모듈은 Nest 애플리케이션에 쉽게 통합할 수 있도록 해줍니다. 아래에서는 `@nestjs/passport`를 사용하겠지만, 먼저 바닐라 Passport가 어떻게 작동하는지 살펴봅시다.

바닐라 패스포트에서는 두 가지를 제공하여 전략을 구성합니다:

- . 해당 전략에 특정한 옵션 집합입니다. 예를 들어 JWT 전략에서는 토큰에 서명하기 위한 비밀을 제공할 수 있습니다.
- . "확인 콜백"은 사용자 스토어(사용자 계정을 관리하는 곳)와 상호 작용하는 방법을 Passport에 알려주는 곳입니다. 여기에서 사용자가 존재하는지(및/또는 새 사용자를 생성하는지), 자격 증명이 유효한지 확인합니다. Passport 라이브러리는 이 콜백이 유효성 검사에 성공하면 전체 사용자를 반환하고, 실패하면 null을 반환할 것으로 예상합니다(실패는 사용자를 찾을 수 없거나 패스포트-로컬의 경우 비밀번호가 일치하지 않는 경우로 정의됨).

`nestjs/passport`를 사용하면 `PassportStrategy` 클래스를 확장하여 패스포트 전략을 구성할 수 있습니다. 하위 클래스에서 `super()` 메서드를 호출하여 전략 옵션(위 항목 1)을 전달하고, 선택적으로 옵션 객체를 전달합니다. 하위 클래스에서 `validate()` 메서드를 구현하여 확인 콜백(위 항목 2)을 제공합니다.

먼저 `AuthModule`을 생성하고 그 안에 `AuthService`를 생성하겠습니다:

```
nest g 모듈 인증
nest g 서비스 인증
```

`AuthService`를 구현하면서 사용자 작업을 `UserService`에 캡슐화하는 것이 유용하다는 것을 알게 될 것이므로 이제 해당 모듈과 서비스를 생성해 보겠습니다:

```
nest g 모듈 사용자
nest g 서비스 사용자
```

생성된 파일의 기본 내용을 아래와 같이 바꿉니다. 샘플 앱의 경우, 사용자 서비스는 단순히 하드코딩된 인메모리

사용자 목록과 사용자 이름으로 사용자를 검색하는 찾기 메서드를 유지 관리합니다. 실제 앱에서는 선택한 라이브러리(예: TypeORM, Sequelize, 몽구스 등)를 사용하여 사용자 모델과 지속성 레이어를 빌드할 수 있습니다.

```
@@파일명 (사용자/사용자.서비스)
```

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

```
// 사용자 엔티티 내보내기 유형 User = any를 나타내는 실제 클래스/인터페이스여야 합니다;
```

```

@Injectable()
export 클래스 UsersService {
  private 읽기 전용 사용자 = [
    {
      userId: 1, 사용자명:
        'john',
      비밀번호: 'changeme',
    },
    {
      userId: 2, 사용자 이
      름: 'maria', 비밀번호
      : 'guess',
    },
  ];

  async findOne(username: 문자열): Promise<사용자 | 정의되지 않음> {
    return this.users.find(사용자 => 사용자.사용자이름 === 사용자이름
    );
  }
}
@switch
'@nestjs/common'에서 { Injectable }을 가져옵니다;

@Injectable()
내보내기 클래스 UsersService {
  constructor() {
    this.users = [
      {
        userId: 1, 사용자명
        : 'john',
        비밀번호: 'changeme',
      },
      {
        userId: 2, 사용자
        이름: 'maria', 비밀
        번호: 'guess',
      },
    ],
  ];
}

async findOne(username) {
  반환 이.사용자.찾기(사용자 => 사용자.사용자 이름 === 사용자 이름);
}
}

```

UsersModule에서 필요한 유일한 변경 사항은 UsersService를

모듈 데코레이터를 추가하여 이 모듈 외부에서 볼 수 있도록 합니다(곧 AuthService에서 사용하게 될 것입니다).

```
@@파일명 (사용자/사용자.모듈)
```

```
'@nestjs/common'에서 { Module }을 가져오고,
```

```
'./users.service'에서 { UsersService }를 가져옵니
```

```
다;
```



```

모듈({
  제공자: [UserService], 내보
  내기: [UserService],
})
사용자 모듈 클래스 {} @@스위치 내보
내기
'@nestjs/common'에서 { Module }을 가져오고,
'./users.service'에서 { UserService }를 가져옵니
다;

```

```

모듈({
  제공자: [UserService], 내보
  내기: [UserService],
})
사용자 모듈 클래스 {} 내보내기

```

인증 서비스는 사용자를 검색하고 비밀번호를 확인하는 작업을 수행합니다. 이를 위해 `validateUser()` 메서드를 생성합니다. 아래 코드에서는 편리한 ES6 스프레드 연산자를 사용하여 사용자 객체에서 비밀번호 속성을 제거한 후 반환합니다. 잠시 후 Passport 로컬 전략에서 `validateUser()` 메서드를 호출하겠습니다.

```

@@파일명(auth/auth.service)

'@nestjs/common'에서 { Injectable }을 imports합니다;
'../users/users.service'에서 { UsersService }를 가져옵니다;

@Injectable()
내보내기 클래스 AuthService {
  constructor(private usersService: UsersService) {}

  async validateUser(사용자 이름: 문자열, 패스: 문자열): Promise<any> { const
    user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user; 반
      환 결과;
    }
    널을 반환합니다;
  }
}

@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 imports하고,
'../users/users.service'에서 { UsersService }를 imports합니다;

주입 가능()
@Dependencies(UsersService)
내보내기 클래스 AuthService {
  constructor(usersService) {
    this.usersService = usersService;
  }

  async validateUser(username, pass) {
    const user = await this.usersService.findOne(username);

```

```

    if (user && user.password === pass) {
      const { password, ...result } = user; 반
      환 결과;
    }
    널을 반환합니다;
  }
}

```

경고 경고 물론 실제 애플리케이션에서는 비밀번호를 일반 텍스트로 저장하지 않습니다. 대신 솔트 처리된 단방향 해시 알고리즘이 포함된 `bcrypt`와 같은 라이브러리를 사용할 것입니다. 이 접근 방식을 사용하면 해시된 비밀번호만 저장한 다음 저장된 비밀번호를 들어오는 비밀번호의 해시된 버전과 비교하므로 사용자 비밀번호를 일반 텍스트로 저장하거나 노출하지 않습니다. 샘플 앱을 단순하게 유지하기 위해 이러한 절대적인 의무를 위반하고 일반 텍스트를 사용했습니다. 실제 앱에서는 이렇게 하지 마세요!

이제 `AuthModule`을 업데이트하여 `UsersModule`을 가져옵니다.

```

@@파일명(auth/auth.module)
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'../users/users.module'에서 { UsersModule }을 가져옵니다;

모듈({
  импорт: [UsersModule], 공급자
  : [AuthService],
})
내보내기 클래스 AuthModule {}

@@switch
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'../users/users.module'에서 { UsersModule }을 가져옵니다;

모듈({
  импорт: [UsersModule], 공급자
  : [AuthService],
})
내보내기 클래스 AuthModule {}

```

패스포트 로컬 구현

이제 Passport 로컬 인증 전략을 구현할 수 있습니다. 다음과 같은 파일을 만듭니다.

`local.strategy.ts`를 열고 다음 코드를 추가합니다:

```
@@파일명(auth/local.strategy)

'passport-local'에서 { Strategy }를 가져옵니다;
'@nestjs/passport'에서 { PassportStrategy }를 가져옵니다;
'@nestjs/common'에서 { Injectable, UnauthorizedException }을 임포트하고,
'./auth.service'에서 { AuthService }를 임포트합니다;
```

```

@Injectable()
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(private authService: AuthService) {
    super();
  }

  비동기 유효성 검사(사용자 이름: 문자열, 비밀번호: 문자열): Promise<any> { const
    user = await this.authService.validateUser(username, password); if
    (!user) {
      새로운 UnauthorizedException()을 던집니다;
    }
    사용자를 반환합니다;
  }
}
}
@switch
'passport-local'에서 { Strategy }를 가져옵니다;
'@nestjs/passport'에서 { PassportStrategy }를 가져옵니다;
'@nestjs/common'에서 { Injectable, UnauthorizedException, Dependencies }를 가져옵
니다;
'./auth.service'에서 { AuthService }를 가져옵니다;

주입 가능() @종속성(AuthService)
export class LocalStrategy extends PassportStrategy(Strategy) {
  constructor(authService) {
    super();
    this.authService = authService;
  }

  async validate(username, password) {
    const user = await this.authService.validateUser(username, password);
    if (!user) {
      새로운 UnauthorizedException()을 던집니다;
    }
    사용자를 반환합니다;
  }
}
}

```

모든 패스포트 전략에 대해 앞서 설명한 레시피를 따랐습니다. 패스포트-로컬 사용 사례에서는 구성 옵션이 없으므로 생성자는 옵션 객체 없이 단순히 `super()`를 호출합니다.

정보 힌트 `super()` 호출에 옵션 객체를 전달하여 패스포트 전략의 동작을 사용자 정의할 수 있습니다. 이 예제에서 패스포트-로컬 전략은 기본적으로 요청 본문에서 사용자 이름과 비밀번호라는 속성을 기대합니다. 옵션 객체를 전달하여 다른 속성 이름을 지정할 수 있습니다(예: `super({ 'usernameField': 'email' })`). 자세한 내용은 [Passport 문서](#)를 참조하세요.

또한 `validate()` 메서드도 구현했습니다. 각 전략에 대해 Passport는 적절한 전략별 매개변수 집합을 사용하여 검증 함수(`@nestjs/passport`에서 `validate()` 메서드로 구현됨)를 호출합니다. 로컬 전략의 경우, Passport는 다음과 같은 서명을 가진 `validate()` 메서드를 기대합니다: `validate(username: 문자열, password: 문자열): any`.

대부분의 유효성 검사 작업은 `AuthService`에서 수행되므로(`UsersService`의 도움으로) 이 메서드는 매우 간단합니다. 모든 Passport 전략의 `유효성 검사()` 메서드는 자격 증명에 표시되는 세부 사항만 다를 뿐 비슷한 패턴을 따릅니다. 사용자를 찾고 자격 증명에 유효하면 사용자를 반환하여 Passport가 작업(예: 요청 객체에서 사용자 속성 만들기)을 완료하고 요청 처리 파이프라인을 계속할 수 있습니다. 사용자를 찾을 수 없으면 예외를 발생시키고 `예외 계층에서` 처리하도록 합니다.

일반적으로 각 전략의 유효성 검사() 메서드에서 유일하게 중요한 차이점은 사용자가 존재하고 유효한지 확인하는 방법입니다. 예를 들어, JWT 전략에서는 요구 사항에 따라 디코딩된 토큰에 포함된 `userId`가 사용자 데이터베이스의 레코드와 일치하는지 또는 해지된 토큰 목록과 일치하는지 평가할 수 있습니다. 따라서 전략별 유효성 검사를 하위 분류하고 구현하는 이러한 패턴은 일관성 있고 우아하며 확장 가능합니다.

방금 정의한 패스포트 기능을 사용하도록 `AuthModule`을 구성해야 합니다. 업데이트 `auth.module.ts`를 다음과 같이 수정합니다:

```

@@파일명(auth/auth.module)

'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'../users/users.module'에서 { UsersModule }을 임포트하고,
'@nestjs/passport'에서 { PassportModule }을 임포트하고,
'./local.strategy'에서 { LocalStrategy }를 임포트합니다;

```

```

모듈({
  임포트합니다: [UsersModule, PassportModule], 공
  급자: [AuthService, LocalStrategy],
})
내보내기 클래스 AuthModule {}

@@switch
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'../users/users.module'에서 { UsersModule }을 임포트하고,
'@nestjs/passport'에서 { PassportModule }을 임포트하고,
'./local.strategy'에서 { LocalStrategy }를 임포트합니다;

```

```

모듈({
  임포트합니다: [사용자 모듈, 패스포트 모듈], 공급자:
  [AuthService, LocalStrategy],
})
내보내기 클래스 AuthModule {}

```

내장형 여권 가드

가드 챕터에서는 가드의 주요 기능인 요청이 라우트 핸들러에 의해 처리될지 여부를 결정하는 기능에 대해 설명합니다. 이는 여전히 유효하며 곧 이 표준 기능을 사용하게 될 것입니다.

그러나 `@nestjs/passport` 모듈을 사용하는 맥락에서 처음에는 혼란스러울 수 있는 약간의 새로운 주름도 소개할 것이므로 지금부터 이에 대해 논의해 보겠습니다. 인증 관점에서 앱이 두 가지 상태로 존재할 수 있다고 가정해 보겠습니다:

사용자/클라이언트가 로그인되지 않음(인증되지 않음)

사용자/클라이언트가 로그인(인증됨)되었습니다.

첫 번째 경우(사용자가 로그인하지 않은 경우)에는 두 가지 기능을 수행해야 합니다:

- 인증되지 않은 사용자가 액세스할 수 있는 경로를 제한합니다(즉, 제한된 경로에 대한 액세스를 거부합니다). 이 기능을 처리하기 위해 보호된 경로에 가드를 배치하여 익숙한 기능인 가드를 사용할 것입니다. 예상할 수 있듯이 이 가드에 유효한 JWT가 있는지 확인할 것이므로 나중에 JWT를 성공적으로 발급한 후 이 가드에 대해 작업할 것입니다.
- 이전에 인증되지 않은 사용자가 로그인을 시도할 때 인증 단계 자체를 시작합니다. 이 단계는 유효한 사용자에게 JWT를 발급하는 단계입니다. 잠시 생각해 보면 인증을 시작하려면 사용자 이름/비밀번호 자격 증명을 **POST**해야 한다는 것을 알 수 있으므로 **POST /auth/login** 경로를 사용하여 처리할 수 있습니다. 이 경우 해당 경로에서 정확히 어떻게 패스포트-로컬 전략을 호출할 수 있을지에 대한 의문이 생깁니다.

답은 간단합니다. 약간 다른 유형의 가드를 사용하면 됩니다. 이 작업을 수행하는 내장된 가드를

@nestjs/passport 모듈에서 제공합니다. 이 가드는 패스포트 전략을 호출하고 위에서 설명한 단계(자격 증명 검색, 확인 함수 실행, **사용자** 속성 생성 등)를 시작합니다.

위에 열거한 두 번째 경우(로그인한 사용자)는 로그인한 사용자가 보호된 경로에 액세스할 수 있도록 이미 설명한 표준 유형의 가드에 의존하기만 하면 됩니다.

로그인 경로

이제 전략이 수립되었으므로 **베어본/인증/로그인** 경로를 구현하고 기본 제공 Guard를 적용하여 여권-로컬 플로우를 시작할 수 있습니다.

app.controller.ts 파일을 열고 내용을 다음과 같이 바꿉니다:

@@파일명 (앱.컨트롤러)

'@nestjs/common'에서 { Controller, Request, Post, UseGuards }를 가져오고,
'@nestjs/passport'에서 { AuthGuard }를 가져옵니다;

컨트롤러()

```
export class AppController {  
  @UseGuards(AuthGuard('local'))  
  Post('auth/login')  
  async login(@Request() req) {  
    return req.user;  
  }  
}
```

@@switch

'@nestjs/common'에서 { Controller, Bind, Request, Post, UseGuards }를 임포
트합니다;
'@nestjs/passport'에서 { AuthGuard }를 임포트합니다;

컨트롤러()

```
export class AppController {  
  @UseGuards(AuthGuard('local'))
```

```

    Post('auth/login')
    @Bind(Request())
    async login(req) {
        req.user를 반환합니다;
    }
}

```

사용가드(`AuthGuard('local')`)를 사용하면 패스포트-로컬 전략을 확장할 때 `@nestjs/passport`가 자동으로 프로비저닝한 `AuthGuard`를 사용하고 있습니다. 자세히 살펴보겠습니다. 패스포트 로컬 전략의 기본 이름은 `'local'`입니다. 이 이름을 `@UseGuards()` 데코레이터에서 참조하여 패스포트-로컬 패키지가 제공하는 코드와 연결합니다. 이는 앱에 여러 개의 패스포트 전략이 있는 경우 호출할 전략을 명확히 하기 위해 사용됩니다(각 전략은 전략별 `AuthGuard`를 제공할 수 있음). 지금까지는 이러한 전략이 하나뿐이지만 곧 두 번째 전략을 추가할 예정이므로 명확하게 구분하기 위해 필요합니다.

경로를 테스트하기 위해 지금은 `/auth/login` 경로에서 단순히 사용자를 반환하도록 하겠습니다. 이를 통해 또 다른 Passport 기능을 시연할 수 있습니다: Passport는 `validate()` 메서드에서 반환한 값을 기반으로 사용자 객체를 자동으로 생성하고 이를 요청 객체에 `req.user`로 할당합니다. 나중에는 이 대신 JWT를 생성하고 반환하는 코드로 대체하겠습니다.

API 경로이므로 일반적으로 사용 가능한 `cURL` 라이브러리를 사용하여 테스트해 보겠습니다. `UserService`에 하드코딩된 모든 사용자 객체로 테스트할 수 있습니다.

```

인증/로그인에 $ # POST 보내기
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
$ # result -> {"userId":1,"username":"john"}

```

이 방법은 작동하지만 전략 이름을 `AuthGuard()`에 직접 전달하면 코드베이스에 마법의 문자열이 생깁니다. 대신 아래와 같이 자체 클래스를 생성하는 것이 좋습니다:

```

@@파일명(auth/local-auth.guard)

'@nestjs/common'에서 { Injectable }을 임포트하고,
'@nestjs/passport'에서 { AuthGuard }를 임포트합니다
;

@Injectable()
export class LocalAuthGuard extends AuthGuard('local') {}

```

이제 `/auth/login` 경로 핸들러를 업데이트하고 대신 `LocalAuthGuard`를 사용할 수 있습니다:

```
사용가드(LocalAuthGuard)
@Post('auth/login')
async login(@Request() req) {
  return req.user;
}
```

JWT 기능

이제 인증 시스템의 JWT 부분으로 넘어갈 준비가 되었습니다. 요구 사항을 검토하고 구체화해 보겠습니다:

- 사용자가 사용자 이름/비밀번호로 인증할 수 있도록 허용하여 보호된 API 엔드포인트에 대한 후속 호출에서 사용할 수 있도록 JWT를 반환합니다. 이 요구 사항을 충족하기 위한 작업은 순조롭게 진행 중입니다. 이를 완료하려면 JWT를 발급하는 코드를 작성해야 합니다.
- 무기명 토큰으로 유효한 JWT의 존재를 기반으로 보호되는 API 경로 만들기 JWT 요구 사항을 지원하

기 위해 몇 가지 패키지를 설치해야 합니다.

```
npm install --save @nestjs/jwt passport-jwt
$ npm install --save-dev @types/passport-jwt
```

`nestjs/jwt` 패키지(자세한 내용은 [여기를](#) 참조하세요)은 JWT 조작에 도움이 되는 유틸리티 패키지입니다. `passport-jwt` 패키지는 JWT 전략을 구현하는 Passport 패키지이며 `@types/passport-jwt`는 TypeScript 유형 정의를 제공합니다.

`POST /auth/login` 요청이 어떻게 처리되는지 자세히 살펴봅시다. 패스포트-로컬 전략에서 제공하는 내장 `AuthGuard`를 사용하여 경로를 꾸몄습니다. 즉

- . 경로 핸들러는 사용자가 유효성이 검사된 경우에만 호출됩니다.
- . `req` 매개변수에는 `사용자` 속성이 포함됩니다(여권-로컬 인증 흐름 중에 Passport에 의해 채워짐).

이를 염두에 두고 이제 실제 JWT를 생성하고 이 경로를 통해 반환할 수 있습니다. 서비스를 깔끔하게 모듈화하기 위해 `authService`에서 JWT 생성을 처리하겠습니다. `auth` 폴더에서 `auth.service.ts` 파일을 열고 `login()` 메서드를 추가한 후 그림과 같이 `JwtService`를 가져옵니다:

```
@@파일명(auth/auth.service)

'@nestjs/common'에서 { Injectable }을 imports합니다;
'../users/users.service'에서 { UsersService }를 imports하고,
'@nestjs/jwt'에서 { JwtService }를 imports합니다;

@Injectable()
내보내기 클래스 AuthService { 생
  성자(
    개인 사용자 서비스: UsersService, 비공개
    jwtService: JwtService
  ) {}

  async validateUser(사용자 이름: 문자열, 패스: 문자열): Promise<any> { const
    user = await this.usersService.findOne(username);
    if (user && user.password === pass) {
      const { password, ...result } = user; 반
      환 결과;
    }
    널을 반환합니다;
```

```

    }

    async login(user: any) {
      const payload = { username: user.username, sub: user.userId };
      return {
        액세스_토큰: this.jwtService.sign(페이로드),
      };
    }
  }

  @@switch
  '@nestjs/common'에서 { Injectable, Dependencies }를 가져오고,
  '../users/users.service'에서 { UsersService }를 가져오고,
  '@nestjs/jwt'에서 { JwtService }를 가져오세요;

  @Dependencies(UsersService, JwtService)
  @Injectable()
  export class AuthService {
    constructor(usersService, jwtService) {
      this.usersService = usersService;
      this.jwtService = jwtService;
    }

    async validateUser(username, pass) {
      const user = await this.usersService.findOne(username);
      if (user && user.password === pass) {
        const { password, ...result } = 사용자;

        반환 결과;
      }
      널을 반환합니다;
    }

    async login(user) {
      const payload = { username: user.username, sub: user.userId };
      return {
        access_token: this.jwtService.sign(payload),
      };
    }
  }
}

```

사용자 객체 속성의 하위 집합에서 JWT를 생성하는 `sign()` 함수를 제공하는 `@nestjs/jwt` 라이브러리를 사용하고 있으며, 이 라이브러리는 단일 `access_token` 속성을 가진 간단한 객체로 반환합니다. 참고: JWT 표준과 일관성을 유지하기 위해 `sub`라는 속성 이름을 선택하여 `userId` 값을 보유합니다. 인증 서비스에 `JwtService` 공급자를 삽입하는 것을 잊지 마세요.

이제 새 종속성을 가져오고 `JwtModule`을 구성하기 위해 `AuthModule`을 업데이트해야 합니다. 먼저 `auth` 폴더

에 `constants.ts`를 생성하고 다음 코드를 추가합니다:

```
@@파일명(auth/constants)  
export const jwtConstants = {
```



```
비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게 보  
관하세요.',  
};  
@@switch  
export const jwtConstants = {  
  비밀: '이 값을 사용하지 마세요. 대신 복잡한 비밀을 만들어 소스 코드 외부에 안전하게 보  
관하세요.',  
};
```

이 키를 사용하여 JWT 서명 및 확인 단계 간에 키를 공유할 것입니다.

경고 경고 이 키를 공개적으로 노출하지 마세요. 여기서는 코드가 수행하는 작업을 명확히 하기 위해 공개했지만, 프로덕션 시스템에서는 시크릿 볼트, 환경 변수 또는 구성 서비스 등의 적절한 조치를 사용하여 이 키를 보호해야 합니다.

이제 인증 폴더에서 `auth.module.ts`를 열고 다음과 같이 업데이트합니다:

```

@@파일명(auth/auth.module)

'@nestjs/common'에서 { Module }을 가져오고,

'./auth.service'에서 { AuthService }를 가져옵니다
;

'./local.strategy'에서 { LocalStrategy }를 임포트하고,

'../users/users.module'에서 { UsersModule }을 임포트하고,

'@nestjs/passport'에서 { PassportModule }을 임포트하고,

'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { jwtConstants } './constants'에서 임포트합니다;

```

```

모듈({ import: [
  UsersModule,
  PassportModule,
  JwtModule.register({
    비밀: jwtConstants.secret,
    signOptions: { expiresIn: '60s' },
  }),
],
  공급자: [AuthService, LocalStrategy], 내보내기:
  [AuthService],
})

```

```

내보내기 클래스 AuthModule {}

```

```

@@switch

'@nestjs/common'에서 { Module }을 가져오고,

'./auth.service'에서 { AuthService }를 가져옵니다
;

'./local.strategy'에서 { LocalStrategy }를 임포트하고,

'../users/users.module'에서 { UsersModule }을 임포트하고,

'@nestjs/passport'에서 { PassportModule }을 임포트하고,

'@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { jwtConstants } './constants'에서 임포트합니다;

```

```

모듈({ import: [
  UsersModule,

```

```
PassportModule,
JwtModule.register({
  비밀: jwtConstants.secret,
  signOptions: { expiresIn: '60s' },
}),
],
공급자: [AuthService, LocalStrategy], 내보내기:
[AuthService],
})
내보내기 클래스 AuthModule {}
```

구성 객체를 전달하여 `register()`를 사용하여 `JwtModule`을 구성합니다. Nest `JwtModule`에 대한 자세한 내용은 [여기를](#), 사용 가능한 구성 옵션에 대한 자세한 내용은 [여기를](#) 참조하세요.

이제 `/auth/login` 경로를 업데이트하여 JWT를 반환할 수 있습니다.

@@파일명 (앱.컨트롤러)

```
'@nestjs/common'에서 { Controller, Request, Post, UseGuards }를 임포트하고,
'./auth/local-auth.guard'에서 { LocalAuthGuard }를 임포트합니다;
'./auth/auth.service'에서 { AuthService }를 가져옵니다;
```

컨트롤러()

```
내보내기 클래스 AppController {
  constructor(private authService: AuthService) {}
```

사용가드(LocalAuthGuard)

```
@Post('auth/login')
async login(@Request() req) {
  this.authService.login(req.user)을 반환합니다;
}
```

}

@@switch

```
'@nestjs/common'에서 { Controller, Bind, Request, Post, UseGuards }를 임포
트합니다;
```

```
'./auth/local-auth.guard'에서 { LocalAuthGuard }를 가져오고,
```

```
'./auth/auth.service'에서 { AuthService }를 가져옵니다;
```

컨트롤러()

```
내보내기 클래스 AppController {
  constructor(private authService: AuthService) {}
```

UseGuards(LocalAuthGuard)

```
@Post('auth/login')
@Bind(Request())
async login(req) {
  this.authService.login(req.user)을 반환합니다;
}
```

}

계속해서 cURL을 사용하여 경로를 다시 테스트해 보겠습니다. `UsersService`에 하드코딩된 모든 사용자 객체를 사용하여 테스트할 수 있습니다.

```
인증/로그인에 $ # POST 보내기
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
$ # 결과 -> {"access_token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9..."}
```

참고: 위 JWT는 잘렸습니다.

패스포트 JWT 구현하기

이제 마지막 요구 사항인 요청에 유효한 JWT가 있어야 엔드포인트를 보호할 수 있습니다. 여기서도 Passport가 도움이 될 수 있습니다. JSON 웹 토큰으로 RESTful 엔드포인트를 보호하기 위한 [passport-jwt](#) 전략을 제공합니다.

먼저 인증 폴더에 `jwt.strategy.ts`라는 파일을 만들고 다음 코드를 추가합니다:

```

@@파일명(auth/jwt.strategy)

'passport-jwt'에서 { ExtractJwt, Strategy }를 임포트하
고, '@nestjs/passport'에서 { PassportStrategy }를 임포
트하고, '@nestjs/common'에서 { Injectable }을 임포트합니
다;
import { jwtConstants } from './constants'에서 임포트합니다;

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      비밀 또는 키: jwtConstants.secret,
    });
  }

  async validate(payload: any) {
    반환 { userId: payload.sub, username: payload.username };
  }
}

@@switch

'passport-jwt'에서 { ExtractJwt, Strategy }를 임포트하
고, '@nestjs/passport'에서 { PassportStrategy }를 임포
트하고, '@nestjs/common'에서 { Injectable }을 임포트합니
다;
import { jwtConstants } from './constants'에서 임포트합니다;

@Injectable()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor() {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: false,
      비밀 또는 키: jwtConstants.secret,
    });
  }
}

```

```

    });
  }

  async validate(payload) {
    반환 { userId: payload.sub, username: payload.username };
  }
}

```

JwtStrategy에서는 모든 패스포트 전략에 대해 앞서 설명한 것과 동일한 레시피를 따랐습니다. 이 전략은 약간의 초기화가 필요하므로 `super()` 호출에서 옵션 객체를 전달하여 초기화합니다. 사용 가능한 옵션에 대한 자세한 내용은 [여기에서](#) 확인할 수 있습니다. 저희의 경우 옵션은 다음과 같습니다:

- `jwtFromRequest`: 요청에서 JWT를 추출하는 방법을 제공합니다. 저희는 API 요청의 인증 헤더에 무기명 토큰을 제공하는 표준 방식을 사용합니다. 다른 옵션은 [여기에](#) 설명되어 있습니다.
- `무시 만료`: 명확히 하기 위해 기본값인 `거짓` 설정을 선택했는데, 이 설정은 JWT가 만료되지 않았는지 확인하는 책임을 패스포트 모듈에 위임합니다. 즉, 만료된 JWT가 경로에 제공되면 요청이 거부되고 `401 권한 없음` 응답이 전송됩니다. Passport는 이 작업을 자동으로 처리해 줍니다.
- `secretOrKey`: 저희는 토큰 서명을 위해 대칭형 비밀을 제공하는 편리한 옵션을 사용하고 있습니다. PEM 인코딩된 공개 키와 같은 다른 옵션이 프로덕션 앱에 더 적합할 수 있습니다(자세한 내용은 [여기를](#) 참조하세요). 어떤 경우든 앞서 주의한 대로 이 비밀을 공개적으로 노출하지 마세요.

`유효성 검사()` 메서드에 대해 설명할 필요가 있습니다. jwt 전략의 경우, Passport는 먼저 JWT의 서명을 확인하고 JSON을 디코딩합니다. 그런 다음 디코딩된 JSON을 단일 매개변수로 전달하는 `validate()` 메서드를 호출합니다. JWT 서명이 작동하는 방식에 따라 이전에 서명하고 유효한 사용자에게 발급한 유효한 토큰을 수신하고 있다는 것을 보장할 수 있습니다.

이 모든 작업의 결과로 `validate()` 콜백에 대한 응답은 간단합니다. `userId` 및 `사용자 이름` 프로퍼티가 포함된 객체를 반환하기만 하면 됩니다. 다시 한 번 기억해 두세요. Passport는 `validate()` 메서드의 반환값을 기반으로 `사용자` 객체를 빌드하고 이를 `요청` 객체에 프로퍼티로 첨부합니다.

또한 이 접근 방식은 프로세스에 다른 비즈니스 로직을 삽입할 수 있는 여지('후크')를 남긴다는 점도 지적할 가치가 있습니다. 예를 들어, `유효성 검사()` 메서드에서 데이터베이스 조회를 수행하여 사용자에 대한 더 많은 정보를 추출하여 `요청에서` 더 풍부한 `사용자` 객체를 사용할 수 있게 할 수 있습니다. 또한 해지된 토큰 목록에서 `userId`를 조회하여 토큰 해지를 수행하는 등 추가적인 토큰 유효성 검사를 수행할 수도 있습니다. 여기 샘플 코드에서 구현한 모델은 빠른 "상태 비저장형 JWT" 모델로, 각 API 호출은 유효한 JWT의 존재 여부에 따라 즉시

승인되며 요청자에 대한 약간의 정보(사용자 ID 및 사용자 이름)를 요청 파이프라인에서 사용할 수 있습니다.

AuthModule에 새 JwtStrategy를 공급자로 추가합니다:

```
@@파일명(auth/auth.module)
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'./local.strategy'에서 { LocalStrategy }를 가져옵니다;
```



```

'./jwt.strategy'에서 { JwtStrategy }를 가져옵니다;
'../users/users.module'에서 { UsersModule }을 임포트하고
, '@nestjs/passport'에서 { PassportModule }을 임포트하고
, '@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { jwtConstants } from './constants'에서 임포트합니다;

모듈({ import:
  [
    UsersModule,
    PassportModule,
    JwtModule.register({
      비밀: jwtConstants.secret, signOptions:
        { expiresIn: '60s' },
    }),
  ],
  공급자: [AuthService, LocalStrategy, JwtStrategy], 내보내기:
    [AuthService],
})
내보내기 클래스 AuthModule {}
@@switch
'@nestjs/common'에서 { Module }을 가져오고,
'./auth.service'에서 { AuthService }를 가져옵니다
;
'./local.strategy'에서 { LocalStrategy }를 가져오고,
'./jwt.strategy'에서 { JwtStrategy }를 가져옵니다;
'../users/users.module'에서 { UsersModule }을 임포트하고
, '@nestjs/passport'에서 { PassportModule }을 임포트하고
, '@nestjs/jwt'에서 { JwtModule }을 임포트합니다;
import { jwtConstants } from './constants'에서 임포트합니다;

모듈({ import:
  [
    UsersModule,
    PassportModule,
    JwtModule.register({
      비밀: jwtConstants.secret, signOptions:
        { expiresIn: '60s' },
    }),
  ],
  공급자: [AuthService, LocalStrategy, JwtStrategy], 내보내기:
    [AuthService],
})

```

내보내기 클래스 AuthModule {}

JWT에 서명할 때 사용한 것과 동일한 비밀번호를 가져옴으로써 Passport에서 수행하는 확인 단계와 AuthService에서 수행하는 서명 단계가 공통 비밀번호를 사용하도록 합니다.

마지막으로, 기본 제공 AuthGuard를 확장하는 JwtAuthGuard 클래스를 정의합니다:

```
@@파일명(auth/jwt-auth.guard)
'@nestjs/common'에서 { Injectable }을 임포트하고,
'@nestjs/passport'에서 { AuthGuard }를 임포트합니다
;
```

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {}
```

보호 경로 및 JWT 전략 가드 구현

이제 보호 경로와 관련 가드를 구현할 수 있습니다. `app.controller.ts` 파

일을 열고 아래와 같이 업데이트합니다:

```
@@파일명 (앱.컨트롤러)

'@nestjs/common'에서 { Controller, Get, Request, Post, UseGuards
}를 임포트합니다;

'./auth/jwt-auth.guard'에서 { JwtAuthGuard }를 가져오고,
'./auth/local-auth.guard'에서 { LocalAuthGuard }를 가져오고,
'./auth/auth.service'에서 { AuthService }를 가져오세요;

컨트롤러()

내보내기 클래스 AppController {
  constructor(private authService: AuthService) {}

  사용자가드(LocalAuthGuard)
  @Post('auth/login')
  async login(@Request() req) {
    this.authService.login(req.user)을 반환합니다;
  }

  UseGuards(JwtAuthGuard)
  @Get('profile')
  getProfile(@Request() req) {
    req.user를 반환합니다;
  }
}

@@switch
'@nestjs/common'에서 { 컨트롤러, 종속성, 바인드, 가져오기, 요청, 게시, 사용자가드 }를 임
포트합니다;

'./auth/jwt-auth.guard'에서 { JwtAuthGuard }를 가져오고,
'./auth/local-auth.guard'에서 { LocalAuthGuard }를 가져오고,
'./auth/auth.service'에서 { AuthService }를 가져오세요;
```

```
종속성(AuthService)
@Controller()
export class AppController {
  constructor(authService) {
    this.authService = authService;
  }

  UseGuards(LocalAuthGuard)
  @Post('auth/login')
  @Bind(Request())
  async login(req) {
```

```

    this.authService.login(req.user)을 반환합니다;
  }

  UseGuards(JwtAuthGuard)
  @Get('profile')
  @Bind(Request())
  getProfile(req) {
    req.user를 반환합니다;
  }
}

```

다시 한 번, passport-jwt 모듈을 구성할 때 @nestjs/passport 모듈이 자동으로 프로비저닝한 AuthGuard를 적용하고 있습니다. 이 가드는 기본 이름인 jwt로 참조됩니다. GET /profile 경로가 호출되면 가드가 자동으로 passport-jwt 사용자 지정 구성 전략을 호출하고 JWT의 유효성을 검사한 후 사용자 속성을 요청 개체에 할당합니다.

앱이 실행 중인지 확인하고 cURL을 사용하여 경로를 테스트합니다.

```

$ # GET /profile
curl http://localhost:3000/profile
결과 -> {"statusCode":401,"message":"승인되지 않음"}

$ # POST /auth/login
curl -X POST http://localhost:3000/auth/login -d '{"username": "john",
"password": "changeme"}' -H "Content-Type: application/json"
$ # 결과 ->
{"access_token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm... }

이전 단계에서 무기명 코드로 반환된 access_token을 사용하여 /profile을 GET합니다.
curl http://localhost:3000/profile -H "인증: 무기명
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2Vybm..."
$ # result -> {"userId":1,"username":"john"}

```

인증 모듈에서 JWT의 만료 시간을 60초로 구성했습니다. 이는 너무 짧은 만료일 수 있으며 토큰 만료 및 새로 고침에 대한 세부 사항을 다루는 것은 이 문서의 범위를 벗어납니다. 하지만 JWT의 중요한 특성과 패스포트-JWT 전략을 설명하기 위해 이 방법을 선택했습니다. 인증 후 60초를 기다렸다가 GET /프로필 요청을 시도하면 401 권한 없음 응답을 받게 됩니다. 이는 Passport가 자동으로 JWT의 만료 시간을 확인하므로 애플리케이션에서 직접 확인해야 하는 수고를 덜어주기 때문입니다.

이제 JWT 인증 구현이 완료되었습니다. 이제 자바스크립트 클라이언트(예: Angular/React/Vue) 및 기타 자바스크립트 앱이 트위터 API 서버와 안전하게 인증하고 통신할 수 있습니다.

가드 확장

대부분의 경우 제공된 `AuthGuard` 클래스를 사용하는 것으로 충분합니다. 그러나 기본 오류 처리 또는 인증 로직을 간단히 확장하려는 사용 사례가 있을 수 있습니다. 이를 위해 다음을 확장할 수 있습니다.

를 사용하여 내장 클래스와 하위 클래스 내의 메서드를 재정의할 수 있습니다.

```
import {
  ExecutionContext,
  Injectable,
  UnauthorizedException,
} from '@nestjs/common'에서 가져옵니다;
'@nestjs/passport'에서 { AuthGuard }를 임포트합니다;

@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  canActivate(context: ExecutionContext) {
    // 여기에 사용자 지정 인증 로직 추가
    // 예를 들어, super.logIn(request)를 호출하여 세션을 설정합니다. 반환
    super.canActivate(context);
  }

  handleRequest(err, user, info) {
    // "info" 또는 "err" 인자를 기반으로 예외를 던질 수 있습니다.
    if (err || !user) {
      throw err || new UnauthorizedException();
    }
    사용자를 반환합니다;
  }
}
```

기본 오류 처리 및 인증 로직을 확장하는 것 외에도 인증이 일련의 전략을 거치도록 허용할 수 있습니다. 첫 번째 전략이 성공, 리디렉션 또는 오류를 일으키면 체인이 중단됩니다.

인증 실패는 각 전략을 통해 연쇄적으로 진행되며, 모든 전략이 실패하면 최종적으로 인증이 실패합니다.

```
export class JwtAuthGuard extends AuthGuard(['strategy_jwt_1',
'strategy_jwt_2', '...']) { ... }
```

전 세계적으로 인증 사용

대부분의 엔드포인트를 기본적으로 보호해야 하는 경우, 인증 가드를 **전역 가드**로 등록하고 각 컨트롤러 위에 `@UseGuards()` 데코레이터를 사용하는 대신 어떤 경로를 공개해야 하는지 플래그를 지정하면 됩니다.

먼저, 모듈에 관계없이 다음 구성을 사용하여 `JwtAuthGuard`를 전역 가드로 등록합니다:

```
제공자: [  
  {  
    제공: APP_GUARD,  
    useClass: JwtAuthGuard,
```



```
  },
],
```

이렇게 하면 Nest는 자동으로 모든 엔드포인트에 `JwtAuthGuard`를 바인딩합니다.

이제 경로를 공개로 선언하는 메커니즘을 제공해야 합니다. 이를 위해 `SetMetadata` 데코레이터 팩토리 함수를 사용하여 사용자 정의 데코레이터를 만들 수 있습니다.

```
import { SetMetadata } from '@nestjs/common';

export const IS_PUBLIC_KEY = 'isPublic';
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

위 파일에서 두 개의 상수를 내보냈습니다. 하나는 `IS_PUBLIC_KEY`라는 메타데이터 키이고, 다른 하나는 `Public`이라고 부를 새 데코레이터 자체입니다(프로젝트에 맞는 다른 이름을 지정할 수 있습니다).

이제 사용자 정의 `@Public()` 데코레이터가 생겼으므로 다음과 같이 모든 메서드를 데코레이션하는 데 사용할 수 있습니다:

```
@Public()
@Get()
findAll() {
  반환 [];
}
```

마지막으로, `"isPublic"` 메타데이터가 발견될 때 `참`을 반환하도록 `JwtAuthGuard`가 필요합니다. 이를 위해 `Reflector` 클래스를 사용하겠습니다(자세한 내용은 [여기를](#) 참조하세요).

```
@Injectable()
export class JwtAuthGuard extends AuthGuard('jwt') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {
    const isPublic = this.reflector.getAllAndOverride<boolean>(
      IS_PUBLIC_KEY, [
        context.getHandler(),
        context.getClass(),
      ]);
    if (isPublic) { 참
      을 반환합니다;
    }
    반환 super.canActivate(context);
  }
}
```

요청 범위 전략

패스포트 API는 라이브러리의 글로벌 인스턴스에 전략을 등록하는 것을 기반으로 합니다. 따라서 전략은 요청 종속 옵션을 갖거나 요청별로 동적으로 인스턴스화되도록 설계되지 않았습니다([요청 범위](#) 제공자에 대한 자세한 내용을 참조하세요). 전략을 요청 범위로 구성하면 Nest는 특정 경로에 묶이지 않기 때문에 전략을 인스턴스화하지 않습니다. 요청별로 어떤 "요청 범위" 전략을 실행해야 하는지 물리적으로 결정할 수 있는 방법은 없습니다.

하지만 전략 내에서 요청 범위가 지정된 공급자를 동적으로 해결할 수 있는 방법이 있습니다. 이를 위해 [모듈 참조](#) 기능을 활용합니다.

먼저 `local.strategy.ts` 파일을 열고 일반적인 방법으로 `ModuleRef`를 삽입합니다:

```
constructor(private moduleRef: ModuleRef) {  
  super({  
    passReqToCallback: true,  
  });  
}
```

정보 힌트 `ModuleRef` 클래스는 `@nestjs/core` 패키지에서 가져옵니다.

위와 같이 `passReqToCallback` 구성 속성을 `true`로 설정해야 합니다.

다음 단계에서는 새 컨텍스트 식별자를 생성하는 대신 요청 인스턴스를 사용하여 현재 컨텍스트 식별자를 가져옵니다([여기에서](#) 요청 컨텍스트에 대해 자세히 읽어보세요).

이제 `LocalStrategy` 클래스의 `validate()` 메서드 내에서 `ContextIdFactory` 클래스의 `getByRequest()` 메서드를 사용하여 요청 객체를 기반으로 컨텍스트 ID를 생성한 다음 이를 `resolve()` 호출에 전달합니다 :

```
비동기 유효성 검사(요청: 요청, 사용자명: 문자열, 비밀번호: 문자열,  
자열,  
) {  
  const contextId = ContextIdFactory.getByRequest(request);  
  // "AuthService"는 요청 범위가 지정된 제공자입니다.  
  const authService = await this.moduleRef.resolve(AuthService,  
contextId);  
  ...  
}
```

위의 예제에서 `resolve()` 메서드는 `AuthService` 공급자의 요청 범위 인스턴스를 비동기적으로 반환합니다 (AuthService가 요청 범위 공급자로 표시되어 있다고 가정했습니다).

여권 사용자 지정

모든 표준 패스포트 사용자 지정 옵션은 `등록()` 메서드를 사용하여 같은 방식으로 전달할 수 있습니다. 사용 가능한 옵션은 구현 중인 전략에 따라 다릅니다. 예를 들어

```
PassportModule.register({ 세션: true });
```

생성자에 옵션 객체를 전달하여 전략을 구성할 수도 있습니다. 로컬 전략의 경우 예를 들어 다음과 같이 전달할 수 있습니다:

```
constructor(private authService: AuthService) {  
  super({  
    usernameField: '이메일',  
    passwordField: '비밀번호',  
  });  
}
```

숙소 이름은 공식 [패스포트 웹사이트](#)에서 확인하세요. 네이밍 전

략

전략을 구현할 때 `PassportStrategy` 함수에 두 번째 인수를 전달하여 전략의 이름을 지정할 수 있습니다. 이렇게 하지 않으면 각 전략에 기본 이름이 지정됩니다(예: jwt-전략의 경우 'jwt'):

```
내보내기 클래스 JwtStrategy는 PassportStrategy를 확장합니다(strategy, 'myjwt').
```

그런 다음 `@UseGuards(AuthGuard('myjwt'))` 같은 데코레이터를 통해 이를 참조합

니다. GraphQL

[GraphQL](#)과 함께 `AuthGuard`를 사용하려면 기본 제공 `AuthGuard` 클래스를 확장하고 `getRequest()` 메서드를 재정의하세요.

```
@Injectable()
export class GqlAuthGuard extends AuthGuard('jwt') {
  getRequest(context: ExecutionContext) {
    const ctx = GqlExecutionContext.create(context);
    return ctx.getContext().req;
  }
}
```

그래프 쿼리 리졸버에서 현재 인증된 사용자를 가져오려면 `@CurrentUser()`를 정의하면 됩니다.

데코레이터:

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

export const CurrentUser = createParamDecorator( (데이
  터: 알 수 없음, 컨텍스트: 실행 컨텍스트) => {
    const ctx = GqlExecutionContext.create(context);
    return ctx.getContext().req.user;
  },
);
```

리졸버에서 위의 데코레이터를 사용하려면 쿼리 또는 변이의 매개 변수로 포함해야 합니다:

```
쿼리(반환 => 사용자) @사용가드(GqlAuthGuard)
whoAmI(@CurrentUser() 사용자: User) {
  this.userService.findById(user.id)를 반환합니다;
}
```

핫 리로드

애플리케이션의 부트스트랩 프로세스에 가장 큰 영향을 미치는 것은 TypeScript 컴파일입니다. 다행히도 **웹팩 HMR**(핫 모듈 교체)을 사용하면 변경 사항이 발생할 때마다 전체 프로젝트를 다시 컴파일할 필요가 없습니다. 따라서 애플리케이션을 인스턴스화하는 데 필요한 시간이 크게 줄어들고 반복 개발이 훨씬 쉬워집니다.

경고 **웹팩**은 에셋(예: `graphql` 파일)을 `dist` 폴더에 자동으로 복사하지 **않습니다**. 마찬가지로 **웹팩**은 글로벌 정적 경로(예: `TypeOrmModule`의 `엔티티` 속성)와 호환되지 않습니다.

CLI 사용

Nest CLI를 사용하는 경우 구성 프로세스는 매우 간단합니다. CLI는 **웹팩**을 래핑하여 **HotModuleReplacementPlugin**을 사용할 수 있도록 합니다.

설치

먼저 필요한 패키지를 설치합니다:

```
$ npm i --save-dev 웹팩-노드-외부 실행 스크립트 웹팩 플러그인 웹팩
```

정보 힌트 클래식 `안`이 아닌 `안 베리`를 사용하는 경우 **웹팩-노드-외부** 패키지를 **웹팩-node-외부** 대신 설치하세요.

구성

설치가 완료되면 애플리케이션의 루트 디렉터리에 `webpack-hmr.config.js` 파일을 생성합니다.


```
const nodeExternals = require('webpack-node-externals');
const { RunScriptWebpackPlugin } = require('run-script-webpack-plugin');

module.exports = 함수 (옵션, 웹팩) { 반환 {
  ...옵션,
  항목: ['webpack/hot/poll?100', options.entry], 외부: [
    nodeExternals({
      허용 목록: ['webpack/hot/poll?100'],
    }),
  ],
  플러그인: [
    ...options.plugins,
    새로운 웹팩.핫모듈교체플러그인(), 새로운 웹팩.위치
  ],
  무시플러그인({
    경로: [/\.js$/, /\.d\.ts$/],
  })
}
```

```

    }),
    새로운 런스크립트웹팩 플러그인({ 이름: options.output.filename, 자동 재시작:
false })),
  ],
};
};
};

```

정보 힌트 Yarn 베리(클래식 Yarn이 아님)의 경우, 외부 구성 속성의 `nodeExternals`를 사용하는 대신 `webpack-pnp-externals` 패키지의 `WebpackPnpExternals`를 사용하세요:

```
WebpackPnpExternals({{ '{' }} exclude: ['webpack/hot/poll? 100'] {{ '}' }}).
```

이 함수는 기본 웹팩 구성이 포함된 원본 객체를 첫 번째 인수로, Nest CLI에서 사용하는 기본 웹팩 패키지에 대한 참조를 두 번째 인수로 받습니다. 또한 이 함수는 `HotModuleReplacementPlugin`, `WatchIgnorePlugin` 및 `RunScriptWebpackPlugin` 플러그인을 사용하여 수정된 웹팩 구성을 반환합니다.

핫 모듈 교체

HMR을 활성화하려면 애플리케이션 입력 파일(`main.ts`)을 열고 다음 웹팩 관련 지침을 추가합니다:

```

선언하다 const module: any;

async 함수 bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}

부트스트랩();

```

실행 프로세스를 간소화하려면 `package.json` 파일에 스크립트를 추가하세요.

```
"start:dev": "nest build --webpack --webpackPath webpack-hmr.config.js --watch"
```

이제 명령줄을 열고 다음 명령을 실행하기만 하면 됩니다:

```
$ npm 실행 시작:dev
```

CLI 사용 안 함

Nest CLI를 사용하지 않는 경우 구성이 약간 더 복잡해집니다(수동 단계가 더 필요함).

설치

먼저 필요한 패키지를 설치합니다:

```
$ npm i --save-dev 웹팩 웹팩-cli 웹팩-노드-외부 ts-로더 실행 스크립트-웹팩-플러그인
```

정보 힌트 클래식 안이 아닌 안 베리를 사용하는 경우 **웹팩-노드-외부** 패키지를 **웹팩-node-외부** 대신 설치하세요.

구성

설치가 완료되면 애플리케이션의 루트 디렉터리에 **webpack.config.js** 파일을 생성합니다.

```
const webpack = require('webpack');
const path = require('path');
const nodeExternals = require('webpack-node-externals');
const { RunScriptWebpackPlugin } = require('run-script-webpack-plugin');

module.exports = {
  항목: ['webpack/hot/poll?100', './src/main.ts'], target:
  'node',
  외부: [
    nodeExternals({
      허용 목록: ['webpack/hot/poll?100'],
    }),
  ],
  모듈: { rules:
    [
      {
        test: /\.tsx?$/, 사
        용: 'ts-loader',
        제외합니다: /node_modules/,
      },
    ],
  },
  모드: '개발', 해결: {
    확장자: ['.tsx', '.ts', '.js'],
  },
  플러그인: [
    새로운 웹팩.핫모듈교체플러그인(),
    새로운 런스크립트웹팩 플러그인({ 이름: 'server.js', 자동재시작: false }),
  ],
  출력합니다: {
```

```
경로: path.join(__dirname, 'dist'),
파일명: 'server.js',
},
};
```

정보 힌트 Yarn 베리(클래식 Yarn이 아님)의 경우, 외부 구성 속성의 `nodeExternals`를 사용하는 대신 `webpack-pnp-externals` 패키지의 `WebpackPnpExternals`를 사용하세요:

```
WebpackPnpExternals({{ '{' }} exclude: ['webpack/hot/poll? 100'] {{ '}' }}}).
```

이 설정은 엔트리 파일의 위치, 컴파일된 파일을 저장하는 데 사용할 디렉토리, 소스 파일을 컴파일하는 데 사용할 로더의 종류 등 애플리케이션에 대한 몇 가지 필수 사항을 웹팩에 알려줍니다. 일반적으로 모든 옵션을 완전히 이해하지 못하더라도 이 파일을 그대로 사용할 수 있습니다.

핫 모듈 교체

HMR을 활성화하려면 애플리케이션 입력 파일(`main.ts`)을 열고 다음 웹팩 관련 지침을 추가합니다:

```
선언하다 const module: any;

async 함수 bootstrap() {
  const app = await NestFactory.create(AppModule);
  await app.listen(3000);

  if (module.hot) {
    module.hot.accept();
    module.hot.dispose(() => app.close());
  }
}

부트스트랩();
```

실행 프로세스를 간소화하려면 `package.json` 파일에 스크립트를 추가하세요.

```
"start:dev": "webpack --config webpack.config.js --watch"
```

이제 명령줄을 열고 다음 명령을 실행하기만 하면 됩니다:

```
$ npm 실행 시작:dev
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

MikroORM

이 레시피는 Nest에서 MikroORM을 시작하는 사용자를 돕기 위해 마련되었습니다. MikroORM은 데이터 매퍼, 작업 단위 및 ID 맵 패턴을 기반으로 하는 Node.js용 TypeScript ORM입니다. TypeORM을 대체할 수 있는 훌륭한 대안이며 TypeORM에서 마이그레이션하는 것은 매우 쉽습니다. MikroORM에 대한 전체 문서는 [여기에서](#) 확인할 수 있습니다.

정보 @mikro-orm/nestjs는 타사 패키지이며 NestJS 코어 팀에서 관리하지 않습니다. 라이브러리와 관련된 문제가 발견되면 [해당 리포지토리에](#) 보고해 주세요.

설치

Nest에 MikroORM을 통합하는 가장 쉬운 방법은 @mikro-orm/nestjs [모듈](#)을 사용하는 것입니다. Nest, MikroORM 및 기본 드라이버 옆에 설치하기만 하면 됩니다:

```
$ npm i @mikro-orm/core @mikro-orm/nestjs @mikro-orm/mysql # for
mysql/mariadb
```

MikroORM은 [포스트그레스](#), [sqlite](#), [몽고도](#) 지원합니다. 모든 드라이버에 대한 [공식 문서](#)를 참조하세요.

설치 프로세스가 완료되면 MikroOrmModule을 루트 앱모듈로 가져올 수 있습니다.

```
모듈({ import: [
  MikroOrmModule.forRoot({
    entities: ['./dist/entities'],
    entitiesTs: ['./src/entities'],
    dbName: 'my-db-name.sqlite3',
    type: 'sqlite',
  }),
],
컨트롤러: [AppController], 공급자: [
  앱서비스],
})
내보내기 클래스 AppModule {}
```

forRoot() 메서드는 MikroORM 패키지의 init()과 동일한 구성 객체를 받습니다. 전체 구성 설명서는 [이 페이지](#)에서 확인하세요.

또는 구성 파일 `mikro-orm.config.ts`를 생성하여 [CLI를 구성한](#) 다음 인자 없이 `forRoot()`를 호출할 수 있습니다. 트리 셰이킹을 사용하는 빌드 툴을 사용하는 경우에는 이 방법이 작동하지 않습니다.

```
모듈({ import: [  
  MikroOrmModule.forRoot(),
```

```
    ],
    ...
  })
  내보내기 클래스 AppModule {}
```

그 후에는 다른 곳에서 모듈을 임포트하지 않고도 전체 프로젝트에 `EntityManager`를 삽입할 수 있습니다.

```
'@mikro-orm/core'에서 { MikroORM }을 가져옵니다;
// 드라이버 패키지에서 EntityManager를 가져오거나 '@mikro-orm/knex'에서 {
EntityManager }를 가져옵니다;

@Injectable()
내보내기 클래스 MyService { 생성
  자(
    비공개 읽기 전용ORM: 비공개 읽기 전용 em:
    EntityManager,
  ) {}
```

정보 `EntityManager`는 `@mikro-orm/driver` 패키지에서 가져오는데, 여기서 driver는 `mysql`, `sqlite`, `postgres` 또는 사용 중인 드라이버입니다. 종속성으로 `@mikro-orm/knex`가 설치되어 있는 경우, 거기에서 `EntityManager`를 가져올 수도 있습니다.

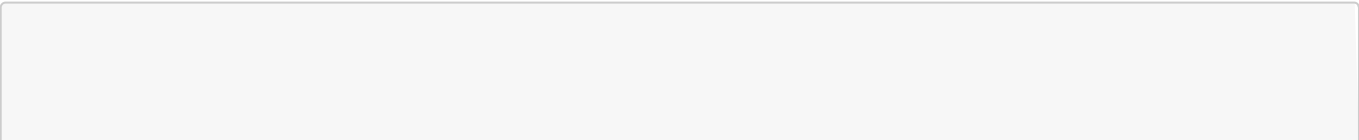
리포지토리

MikroORM은 리포지토리 디자인 패턴을 지원합니다. 모든 엔티티에 대해 리포지토리를 만들 수 있습니다. [여기에서](#) 리포지토리에 대한 전체 문서를 읽어보세요. 현재 범위에 등록할 리포지토리를 정의하려면 `forFeature()` 메서드를 사용할 수 있습니다. 예를 들면 다음과 같습니다:

정보 기본 엔티티는 리포지토리가 없으므로 `forFeature()`를 통해 등록해서는 안 됩니다. 반면에 기본 엔티티는 `forRoot()`(또는 일반적으로 ORM 구성)의 목록에 포함되어야 합니다.

```
// photo.module.ts
@Module({
  imports: [MikroOrmModule.forFeature([Photo])],
  providers: [PhotoService],
  controllers: [PhotoController],
})
내보내기 클래스 PhotoModule {}
@Module({
```

를 생성하고 루트 AppModule로 가져옵니다:



```

    imports: [MikroOrmModule.forRoot(...), PhotoModule],
  })
  내보내기 클래스 AppModule {}

```

이런 식으로 `@InjectRepository()` 메서드를 사용하여 `PhotoService`에 `PhotoRepository`를 주입할 수 있습니다.

데코레이터:

```

@Inject()
내보내기 클래스 PhotoService {
  생성자(
    인젝트 리포지토리(사진)
    비공개 읽기 전용 사진 저장소: 엔티티저장소<사진>,
  ) {}
}

```

사용자 지정 리포지토리 사용

사용자 정의 리포지토리를 사용할 때 리포지토리의 이름을 `getRepositoryToken()` 메서드와 같은 방식으로 지정하면 `@InjectRepository()` 데코레이터를 사용하지 않아도 됩니다:

```

export const getRepositoryToken = <T>(엔티티: 엔티티이름<T>) => =>
  `${Utils.className(entity)}Repository`;

```

즉, 리포지토리의 이름을 엔티티의 이름과 동일하게 지정하고 `Repository` 접미사를 추가하면 리포지토리가 Nest DI 컨테이너에 자동으로 등록됩니다.

```

// `**./author.entity.ts**`
@Entity()
내보내기 클래스 Author {
  // 에서 추론할 수 있도록 `em.getRepository()` [EntityRepositoryType]?
  AuthorRepository;
}

// `**./author.repository.ts**`
@Repository(저자)
내보내기 클래스 AuthorRepository extends EntityRepository<Author> {
  @Inject() 지정 메소드...
}

내보내기 클래스 MyService {

```

사용자 지정 리포지토리 이름이 `getRepositoryToken()`이 반환하는 이름과 동일하므로 더 이상 `@InjectRepository()` 데코레이터가 필요하지 않습니다:

```

    생성자(비공개 읽기 전용 저장소: AuthorRepository) {}
}

```

엔티티 자동 로드

정보 자동로드 엔티티 옵션이 v4.1.0에 추가되었습니다.

연결 옵션의 엔티티 배열에 엔티티를 수동으로 추가하는 작업은 번거로울 수 있습니다. 또한 루트 모듈에서 엔티티를 참조하면 애플리케이션 도메인 경계가 깨지고 애플리케이션의 다른 부분으로 구현 세부 정보가 유출될 수 있습니다. 이 문제를 해결하기 위해 정적 글로벌 경로를 사용할 수 있습니다.

그러나 글로벌 경로는 웹팩에서 지원되지 않으므로 모노레포 내에서 애플리케이션을 빌드하는 경우 사용할 수 없습니다. 이 문제를 해결하기 위해 대체 솔루션이 제공됩니다. 엔티티를 자동으로 로드하려면 아래 그림과 같이 구성 객체(`forRoot()` 메서드에 전달됨)의 `autoLoadEntities` 속성을 `true`로 설정합니다:

```

모듈({ import: [
  MikroOrmModule.forRoot({
    ...
    autoLoadEntities: true,
  }),
],
})
내보내기 클래스 AppModule {}

```

이 옵션을 지정하면 `forFeature()` 메서드를 통해 등록된 모든 엔티티가 구성 객체의 엔티티 배열에 자동으로 추가됩니다.

정보 정보 `forFeature()` 메서드를 통해 등록되지 않고 (관계를 통해서만) 엔티티에서 참조되는 엔티티는 자동 로드 엔티티 설정을 통해 포함되지 않습니다.

정보 정보 자동 로드 엔티티를 사용해도 MikroORM CLI에는 영향을 미치지 않습니다. 여전히 전체 엔티티 목록이 포함된 CLI 구성이 필요하기 때문입니다. 반면에 CLI가 웹팩을 거치지 않으므로 글로벌을 사용할 수 있습니다.

직렬화

경고 참고 MikroORM은 더 나은 유형 안전성을 제공하기 위해 모든 단일 엔티티 관계를 참조<T> 또는 컬렉션<T> 객체로 래핑합니다. 이렇게 하면 Nest의 내장 직렬화기가 래핑된 관계에 대해 블라인드 처리됩니다. 다시 말해, HTTP 또는 웹소켓 핸들러에서 MikroORM 엔티티를 반환하는 경우, 해당 엔티티의 모든 관계가 직렬화되지 않습니다.

다행히도 MikroORM은 다음을 대신하여 사용할 수 있는 직렬화 API를 제공합니다.

`ClassSerializerInterceptor`.

```

엔티티()

내보내기 클래스 Book {
    @Property({ hidden: true }) // 클래스 트랜스포머와 동일합니다.
    `@제외`
    숨겨진 필드 = Date.now();

    @Property({ persist: false }) // 클래스 트랜스포머와 유사합니다.
    `@Expose()` . 메모리에만 존재하며 직렬화됩니다. count?: 숫자;

    @ManyToOne({
        serializer: (value) => value.name,
        serializedName: '작성자 이름',
    }) // 클래스 트랜스포머의 `@Transform()` 작성자와 동일합니다:
    Author;
}

```

대기열의 범위 지정 핸들러 요청

v4.1.0에 `@UseRequestContext()` 데코레이터가 추가되었습니다.

문서에서 언급했듯이 각 요청에 대해 깨끗한 상태가 필요합니다. 이 작업은 미들웨어를 통해 등록된 `RequestContext` 헬퍼 덕분에 자동으로 처리됩니다.

하지만 미들웨어는 일반 HTTP 요청 처리에 대해서만 실행되는데, 그 외의 요청 범위 메서드가 필요하다면 어떻게 해야 할까요? 한 가지 예로 큐 핸들러나 예약된 작업을 들 수 있습니다.

사용 요청 컨텍스트() 데코레이터를 사용할 수 있습니다. 이 데코레이터를 사용하려면 먼저 현재 컨텍스트에 `MikroORM` 인스턴스를 주입해야 하며, 그 다음 이 인스턴스를 사용하여 컨텍스트를 생성합니다. 내부적으로 데코레이터는 메서드에 대한 새 요청 컨텍스트를 등록하고 컨텍스트 내에서 메서드를 실행합니다.

```

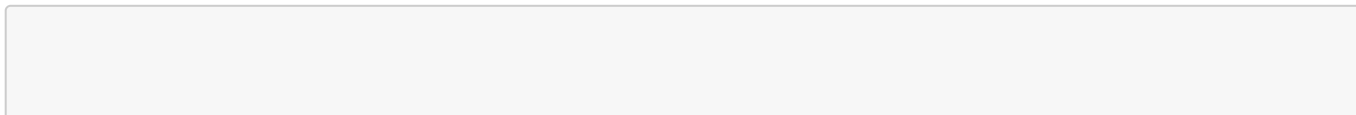
@Injectable()
내보내기 클래스 MyService {
    constructor(private readonly orm: MikroORM) {}

    @UseRequestContext()
    async doSomething() {
        // 별도의 컨텍스트에서 실행됩니다.
    }
}

```


요청 컨텍스트에 **AsyncLocalStorage** 사용

기본적으로 도메인 API는 **RequestContext** 헬퍼에서 사용됩니다. 최신 노드 버전을 사용 중이라면 **@mikro-orm/core@4.0.3** 이후 새로운 **AsyncLocalStorage**도 사용할 수 있습니다:



```
// 새 (글로벌) 스토리지 인스턴스 생성
const storage = new AsyncLocalStorage<EntityManager>();

모듈({ import: [
  MikroOrmModule.forRoot({
    // ...
    registerRequestContext: false, // 자동 미들웨어 비활성화 context: () =>
    storage.getStore(), // AsyncLocalStorage 사용
인스턴스
  }),
],
컨트롤러: [AppController], 공급자: [
  앱서비스],
})
내보내기 클래스 AppModule {}

// 요청 컨텍스트 미들웨어 등록
const app = await NestFactory.create(AppModule, { ... });
const orm = app.get(MikroORM);

app.use((req, res, next) => {
  storage.run(orm.em.fork(true, true), next);
});
```

테스트

`mikro-orm/nestjs` 패키지는 리포지토리를 모킹할 수 있도록 주어진 엔티티를 기반으로 준비된 토큰을 반환하는 `getRepositoryToken()` 함수를 노출합니다.

```
모듈({ providers:
  [
    포토서비스,
    {
      제공: getRepositoryToken(사진), 사용값
      : 모의 저장소,
    },
  ],
})
내보내기 클래스 PhotoModule {}
```

예

MikroORM을 사용한 NestJS의 실제 예제는 [여기에서](#) 확인할 수 있습니다.

SQL(TypeORM)

이 장은 TypeScript에만 적용됩니다.

경고 이 글에서는 사용자 정의 공급자 메커니즘을 사용하여 TypeORM 패키지를 기반으로 **데이터베이스 모듈**을 처음부터 만드는 방법을 배웁니다. 결과적으로 이 솔루션에는 바로 사용할 수 있고 즉시 사용 가능한 전용 **@nestjs/typeorm** 패키지를 사용하면 생략할 수 있는 많은 오버헤드가 포함되어 있습니다. 자세한 내용은 [여기를](#) 참조하세요.

TypeORM은 node.js 세계에서 사용할 수 있는 가장 성숙한 객체 관계형 매퍼(ORM)입니다. TypeScript로 작성되었기 때문에 Nest 프레임워크와 매우 잘 작동합니다.

시작하기

이 라이브러리로 모험을 시작하려면 필요한 모든 종속 요소를 설치해야 합니다:

```
$ npm install --save typeorm mysql2
```

가장 먼저 해야 할 일은 **typeorm** 패키지에서 가져온 새로운 **DataSource().initialize()** 클래스를 사용하여 데이터베이스와의 연결을 설정하는 것입니다. **초기화()** 함수는 **프로미스**를 반환하므로 **비동기 프로바이더**를 만들어야 합니다.

```

@@파일명(database.providers) 'typeorm'
에서 { DataSource }를 가져옵니다;

export const databaseProviders = [
  {
    제공: 'DATA_SOURCE',
    useFactory: async () => {
      const dataSource = new DataSource({
        type: 'mysql',
        호스트: 'localhost',
        포트: 3306, 사용자 이
        름: 'root', 비밀번호:
        'root', 데이터베이스:
        'test', entities:
        [
          __dirname + '/../**/*.entity{.ts,.js}',
        ],
        동기화: true,
      });

      데이터소스 초기화()를 반환합니다;
    },
  },
];

```

경고 동기화 설정: `true`는 프로덕션 환경에서 사용해서는 안 됩니다. 그렇지 않으면 프로덕션 데이터가 손실될 수 있습니다.

정보 힌트 모범 사례에 따라 사용자 정의 공급자를 분리된 파일에 선언했습니다.

*`.providers.ts` 접미사.

그런 다음 나머지 애플리케이션에서 액세스할 수 있도록 이러한 공급자를 내보내야 합니다.

```
@@파일명 (데이터베이스.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'./database.providers'에서 { databaseProviders }를 가져옵니다;

모듈 ({
  공급자: [...데이터베이스 공급자], 내보내기:
  [...databaseProviders],
})
데이터베이스 모듈 클래스 {} 내보내기
```

이제 `@Inject()` 데코레이터를 사용하여 `DATA_SOURCE` 객체를 주입할 수 있습니다. `DATA_SOURCE` 비동기 프로바이더에 의존하는 각 클래스는 `프로미스`가 해결될 때까지 기다립니다.

리포지토리 패턴

`TypeORM`은 리포지토리 디자인 패턴을 지원하므로 각 엔티티에는 자체 리포지토리가 있습니다. 이러한 리포지토리는 데이터베이스 연결에서 얻을 수 있습니다.

하지만 먼저 엔티티가 하나 이상 필요합니다. 공식 문서에 있는 `사진` 엔티티를 재사용하겠습니다.

@@파일명 (사진.엔티티)

'typeorm'에서 { Entity, Column, PrimaryGeneratedColumn }을 가져옵니다;

엔티티()

```
export class Photo {  
  @PrimaryGeneratedColumn()  
  id: number;
```

```
  @Column({ length: 500 })  
  name: 문자열;
```

열('text') 설명: 문자열
입니다;

```
  @Column() 파일명:  
  문자열;
```

열('int') 보기:

숫자;

```
  @Column()
```

```
isPublished: 부울입니다;
}
```

사진 엔티티는 사진 디렉터리에 속합니다. 이 디렉토리는 `PhotoModule`을 나타냅니다. 이제 리포지토리 공급자를 만들어 보겠습니다:

```
@@파일명 (사진.제공자)
'typeorm'에서 { DataSource } 가져오기; './사진.entity'에서 { Photo } 가져오기;

export const photoProviders = [
  {
    제공: '사진_저장소',
    사용 팩토리: (데이터 소스: 데이터 소스) => 데이터 소스.get 리포
지토리(사진),
    주입합니다: ['data_source'],
  },
];
```

경고 경고 실제 애플리케이션에서는 마법의 문자열을 피해야 합니다. 둘 다

사진 저장소 및 데이터 소스는 별도의 `constants.ts` 파일에 보관해야 합니다.

이제 `@Inject()` 데코레이터를 사용하여 리포지토리<사진>을 `PhotoService`에 삽입할 수 있습니다:

```
@@파일명 (사진.서비스)
'@nestjs/common'에서 { Injectable, Inject }를 가져오고,
'typeorm'에서 { Repository }를 가져옵니다;
'./사진.entity'에서 { 사진 }을 가져옵니다;

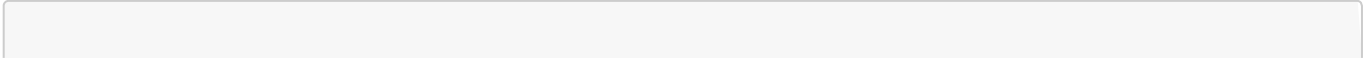
@Injectable()
내보내기 클래스 PhotoService { 생성자(
  @Inject('PHOTO_REPOSITORY')
  비공개 사진 저장소: 리포지토리<사진>,
) {}

비동기 findAll(): Promise<Photo[]> {
  return this.photoRepository.find();
}
}
```

데이터베이스 연결은 비동기식이지만 Nest는 이 프로세스를 최종 사용자에게 완전히 보이지 않게 합니다.

PhotoRepository는 데이터베이스 연결을 기다리고 있으며, PhotoService는 리포지토리를 사용할 준비가 될 때까지 지연됩니다. 각 클래스가 인스턴스화되면 전체 애플리케이션이 시작될 수 있습니다.

다음은 최종 포토모듈입니다:



@@파일명 (사진. 모듈)

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'../database/database.module'에서 { DatabaseModule } 가져오기, './
사진_제공자'에서 { photoProviders } 가져오기;
'./사진.서비스'에서 { PhotoService }를 가져옵니다;
```

모듈({

 임포트: [데이터베이스 모듈], 공급

 자: [

 ...사진_제공자, 사진_서비스,

],

})

내보내기 클래스 PhotoModule {}

정보 힌트 포토모듈을 루트 앱모듈로 가져오는 것을 잊지 마세요.

몽고DB(몽구스)

경고 이 문서에서는 사용자 정의 컴포넌트를 사용하여 몽구스 패키지를 기반으로 데이터베이스 모듈을 처음부터 만드는 방법을 알아봅니다. 결과적으로 이 솔루션에는 바로 사용할 수 있고 즉시 사용 가능한 전용 `@nestjs/mongoose` 패키지를 사용하면 생략할 수 있는 많은 오버헤드가 포함되어 있습니다. 자세한 내용은 [여기를](#) 참조하세요.

몽구스는 가장 널리 사용되는 MongoDB 객체 모델링 도구입니다.

시작하기

이 라이브러리로 모험을 시작하려면 필요한 모든 종속 요소를 설치해야 합니다:

```
$ npm install --save mongoose
```

가장 먼저 해야 할 일은 `connect()` 함수를 사용하여 데이터베이스와의 연결을 설정하는 것입니다. `connect()` 함수는 `Promise`를 반환하므로 비동기 프로바이더를 만들어야 합니다.

```
@@파일명 (데이터베이스.제공자) '몽구스'에서 *
를 몽구스로 가져옵니다;

export const databaseProviders = [
  {
    제공: '데이터베이스_연결',
    useFactory: (): Promise<유형 몽구스> =>
      mongoose.connect('mongodb://localhost/nest'),
  },
];
@@switch
'몽구스'에서 *를 몽구스로 가져옵니다;

export const databaseProviders = [
  {
    제공: '데이터베이스_연결',
    사용 팩토리: () => 몽구스.연결('mongodb://localhost/nest'),
```

정보 힌트 모범 사례에 따라 사용자 정의 공급자를 분리된 파일에 선언했습니다.

`*.providers.ts` 접미사.

그런 다음 이러한 공급자를 내보내서 애플리케이션의 나머지 부분에서 액세스할 수 있도록 해야 합니다.

`@@파일명 (데이터베이스.모듈)`

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

`'./database.providers'에서 { databaseProviders }를 가져옵니다;`

```

모듈({
  제공자: [...데이터베이스 제공자], 내보내기:
  [...databaseProviders],
})
데이터베이스 모듈 클래스 {} 내보내기

```

이제 `@Inject()` 데코레이터를 사용하여 `Connection` 객체를 주입할 수 있습니다. `Connection` 비동기 프로바이더에 종속되는 각 클래스는 `프로미스`가 해결될 때까지 기다립니다.

모델 주입

몽구스에서는 모든 것이 [스키마에서](#) 파생됩니다. `CatSchema`를 정의해 보겠습니다:

```

@@파일명(schemas/cat.schema)

'mongoose'에서 mongoose로 *를 가져옵니다;

export const CatSchema = new mongoose.Schema({
  name: String,
  나이: 숫자, 품종
  : 문자열,
});

```

`CatSchema`는 `cats` 디렉터리에 속합니다. 이 디렉토리는 `CatsModule`을 나타냅니다. 이제 모

델 프로바이더를 생성할 차례입니다:

```
@@파일명(cats.providers)
'몽구스'에서 { Connection }을 가져옵니다;
'./schemas/cat.schema'에서 { CatSchema }를 가져옵니다;

export const catsProviders = [
  {
    제공: 'CAT_MODEL',
    사용 팩토리: (연결: 연결) => 연결.모델('Cat', CatSchema),
    주입합니다: ['데이터베이스_연결'],
  },
];
@@switch
'./schemas/cat.schema'에서 { CatSchema }를 가져옵니다;

export const catsProviders = [
  {
    제공: 'CAT_MODEL',
    useFactory: (connection) => connection.model('Cat', CatSchema),
    inject: ['database_connection'],
  },
];
```

경고 경고 실제 애플리케이션에서는 매직 문자열을 피해야 합니다. `CAT_MODEL` 및 `DATABASE_CONNECTION`은 별도의 `constants.ts` 파일에 보관해야 합니다.

이제 `@Inject()` 데코레이터를 사용하여 `CAT_MODEL`을 `CatsService`에 주입할 수 있습니다:

```

@@파일명(cats.service)
'몽구스'에서 { 모델 }을 가져옵니다;

'@nestjs/common'에서 { Injectable, Inject }를 임포트하고
, './interfaces/cat.interface'에서 { Cat }을 임포트하고,
'./dto/create-cat.dto'에서 { CreateCatDto}를 임포트합니다
;

@Injectable()
내보내기 클래스 CatsService { 생성
  자(
    @Inject('CAT_MODEL')
    private catModel: Model<Cat>,
  ) {}

  async create(createCatDto: CreateCatDto): Promise<Cat> {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }

  비동기 findAll(): Promise<Cat[]> {
    return this.catModel.find().exec();
  }
}
@@switch
'@nestjs/common'에서 { 주입 가능, 종속성 }을 가져옵니다;

주입 가능()
@Dependencies('CAT_MODEL')
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
  }

  async create(createCatDto) {
    const createdCat = new this.catModel(createCatDto);
    return createdCat.save();
  }
}

```

```

async findAll() {

```

```
    이.catModel.find().exec()를 반환합니다;  
  }  
}
```

위의 예에서는 **Cat** 인터페이스를 사용했습니다. 이 인터페이스는 몽구스 패키지에서 **문서**를 확장합니다:

```
'몽구스'에서 { 문서 }를 가져옵니다;

내보내기 인터페이스 Cat 확장 문서 { 읽기 전용 이
    림: 문자열;
    읽기 전용 나이: 숫자; 읽기 전용
    품종: 문자열;
}
```

데이터베이스 연결은 비동기식이지만 Nest는 이 프로세스를 최종 사용자에게 완전히 보이지 않게 합니다.

`CatModel` 클래스는 데이터베이스 연결을 기다리고 있으며, `CatsService`는 모델을 사용할 준비가 될 때까지 지연됩니다. 각 클래스가 인스턴스화되면 전체 애플리케이션이 시작될 수 있습니다.

다음은 최종 `CatsModule`입니다:

```
@@파일명(cats.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'./cats.controller'에서 { CatsController }를 임포트하
고, './cats.service'에서 { CatsService }를 임포트하고,
'./cats.providers'에서 { catsProviders }를 임포트합니다
;
'../database/database.module'에서 { DatabaseModule }을 가져옵니다;

모듈({
    импорт: [데이터베이스 모듈], 컨트롤러:
    [CatsController], 제공자: [
        CatsService,
        ...고양이제공자,
```

정보 힌트 `CatsModule`을 루트 앱모듈로 가져오는 것을 잊지 마세요.

```
내보내기 클래스 CatsModule {}
```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

SQL(시퀀라이즈)

이 장은 TypeScript에만 적용됩니다.

경고 이 글에서는 사용자 정의 컴포넌트를 사용하여 Sequelize 패키지를 기반으로 `DatabaseModule`을 처음부터 새로 만드는 방법을 알아봅니다. 결과적으로 이 기법에는 많은 오버헤드가 포함되어 있지만, 바로 사용할 수 있는 전용 `@nestjsjs/sequelize` 패키지를 사용하면 피할 수 있습니다. 자세한 내용은 [여기](#)를 참조하세요.

[시퀀라이즈](#)는 바닐라 자바스크립트로 작성된 인기 있는 객체 관계형 매퍼(ORM)이지만, 기본 시퀀라이즈에 데코레이터 세트와 기타 추가 기능을 제공하는 [시퀀라이즈](#) 타입스크립트 타입스크립트 래퍼가 있습니다.

시작하기

이 라이브러리로 모험을 시작하려면 다음 종속성을 설치해야 합니다:

```
$ npm install --save sequelize sequelize-typescript mysql2
$ npm install --save-dev @types/sequelize
```

가장 먼저 해야 할 일은 생성자에 옵션 객체를 전달하여 Sequelize 인스턴스를 생성하는 것입니다. 또한 모든 모델을 추가하고(`모델 경로` 속성을 사용하는 대안도 있습니다) 데이터베이스 테이블을 동기화(`()`)해야 합니다.

@@파일명 (데이터베이스.공급자)

'sequelize-typescript'에서 { Sequelize }를 임포트하고,
'../cats/cat.entity'에서 { Cat }을 임포트합니다;

```
export const databaseProviders = [
  {
    제공: 'SEQUELIZE',
    useFactory: async () => {
      const sequelize = new Sequelize({
        dialect: 'mysql',
        호스트: 'localhost',
        포트: 3306, 사용자명:
          'root', 비밀번호:
          'password', 데이터베이스: 'nest',
      });
      sequelize.addModels([Cat]);
      await sequelize.sync();
      return sequelize;
    },
  },
];
```

정보 힌트 모범 사례에 따라 사용자 정의 공급자를 분리된 파일에 선언했습니다.

*.providers.ts 접미사.

그런 다음 이러한 공급자를 내보내서 애플리케이션의 나머지 부분에서 액세스할 수 있도록 해야 합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./database.providers'에서 { databaseProviders }를 가져옵니다;

모듈({
  공급자: [...데이터베이스 공급자], 내보내기:
    [...databaseProviders],
})
```

데이터베이스 모듈 클래스 {} 내보내기

이제 `@Inject()` 데코레이터를 사용하여 `Sequelize` 객체를 주입할 수 있습니다. `Sequelize` 비동기 프로바이더에 종속되는 각 클래스는 `프로미스`가 해결될 때까지 기다립니다.

모델 주입

[시퀀라이즈에서](#) 모델은 데이터베이스의 테이블을 정의합니다. 이 클래스의 인스턴스는 데이터베이스 행을 나타냅니다. 먼저 엔티티가 하나 이상 필요합니다:

```
@파일명(cat.entity)
'sequence-typescript'에서 { 테이블, 열, 모델 }을 가져옵니다;

테이블
내보내기 클래스 Cat extends Model {
  @Column
  이름: 문자열;

  칼럼
  나이: 숫자;

  열 종류: 문자열;
}
```

`Cat` 엔티티는 `cats` 디렉토리에 속합니다. 이 디렉토리는 `CatsModule`을 나타냅니다. 이제 리포지토리 공급자를 생성할 차례입니다:

```
@@파일명(cats.providers)
'./cat.entity'에서 { Cat }을 가져옵니다;

export const catsProviders = [
  {
    제공: 'CATS_REPOSITORY',
    useValue: Cat,
```

```
},
];
```

경고 경고 실제 애플리케이션에서는 마법의 문자열을 피해야 합니다. 둘 다

CATS_REPOSITORY와 SEQUELIZE는 별도의 `constants.ts` 파일에 보관해야 합니다.

Sequelize에서는 정적 메서드를 사용하여 데이터를 조작하므로 여기에 별칭을 만들었습니다. 이제

`@Inject()` 데코레이터를 사용하여 `CATS_REPOSITORY`를 `CatsService`에 주입할 수 있습니다:

```
@파일명(cats.service)
'@nestjs/common'에서 { Injectable, Inject }를 임포트하
고, './dto/create-cat.dto'에서 { CreateCatDto }를 임포
트하고, './cat.entity'에서 { Cat }을 임포트합니다;

@Injectable()
내보내기 클래스 CatsService { 생
  성자(
    @Inject('CATS_REPOSITORY')
    비공개 고양이저장소: 고양이 유형
  ) {}

  비동기 findAll(): Promise<Cat[]> {
    this.catsRepository.findAll<Cat>()을 반환합니다;
  }
}
```

데이터베이스 연결은 비동기식이지만 Nest는 이 프로세스를 최종 사용자에게 완전히 보이지 않게 합니다.

`CATS_REPOSITORY` 프로바이더는 데이터베이스 연결을 기다리고 있으며, `CatsService`는 리포지토리를 사용할 준비가 될 때까지 지연됩니다. 각 클래스가 인스턴스화되면 전체 애플리케이션이 시작될 수 있습니다.

다음은 최종 `CatsModule`입니다:

```
@@파일명(cats.module)

'@nestjs/common'에서 { Module }을 가져옵니다;

'./cats.controller'에서 { CatsController }를 임포트하
고, './cats.service'에서 { CatsService }를 임포트하고,
'./cats.providers'에서 { catsProviders }를 임포트합니다
;
'../database/database.module'에서 { DatabaseModule }을 가져옵니다;

모듈({
  импорт: [데이터베이스 모듈], 컨트롤러:
    [CatsController], 제공자: [
      CatsService,
      ...고양이제공자,
    ],
})
내보내기 클래스 CatsModule {}
```

정보 힌트 `CatsModule`을 루트 앱모듈로 가져오는 것을 잊지 마세요.

라우터 모듈

정보 힌트 이 장은 HTTP 기반 애플리케이션에만 해당됩니다.

HTTP 애플리케이션(예: **REST** API)에서 핸들러의 경로 경로는 컨트롤러에 대해 선언된 (선택적) 접두사 (`@Controller` 데코레이터 내부)와 메서드의 데코레이터에 지정된 모든 경로(예: `@Get('users')`)를 연결하여 결정됩니다.) 이에 대한 자세한 내용은 [이 섹션에서](#) 확인할 수 있습니다.

또한 애플리케이션에 등록된 모든 경로에 [글로벌 접두사](#)를 정의하거나 [버전 관리](#)를 활성화할 수 있습니다.

또한 모듈 수준에서 접두사를 정의하는 것이(따라서 해당 모듈에 등록된 모든 컨트롤러에 대해) 유용할 수 있는 예지 케이스도 있습니다. 예를 들어 "대시보드"라는 애플리케이션의 특정 부분에서 사용되는 여러 가지 엔드포인트를 노출하는 **REST** 애플리케이션을 상상해 보세요. 이 경우 각 컨트롤러 내에서 `/dashboard` 접두사를 반복하는 대신 다음과 같이 [라우터](#) 모듈을 사용할 수 있습니다:

```
모듈({ import: [
  대시보드 모듈, 라우터 모듈.등록([
    {
      경로: '대시보드', 모듈: 대시
      보드모듈,
    },
  ]),
],
})
내보내기 클래스 AppModule {}
```

정보 힌트 [라우터모듈](#) 클래스는 `@nestjs/core` 패키지에서 내보냅니다.

또한 계층 구조를 정의할 수 있습니다. 즉, 각 모듈은 [하위](#) 모듈을 가질 수 있습니다. 자식 모듈은 부모의 접두사를 상속합니다. 다음 예제에서는 [관리 모듈](#) `DashboardModule` 및 `MetricsModule`의 부모 모듈로 등록하겠습니다.

```
모듈({ import: [
  AdminModule,
  DashboardModule,
  MetricsModule,
  RouterModule.register([[
    {
      경로: 'admin',
      module: 관리자 모듈, 아
    이들: [
      {
        경로: '대시보드', 모듈: 대시
        보드모듈,
      },
    ],
  ]],
```

```
    {
      경로: 'metrics',
      module: MetricsModule,
    },
  ],
},
1)
},
1)
],
});
```

정보 힌트 이 기능을 과도하게 사용하면 코드를 작성하기 어려울 수 있으므로 매우 신중하게 사용해야 합니다.
시간이 지나도 유지됩니다.

위의 예시에서 `DashboardModule` 내부에 등록된 모든 컨트롤러에는 추가

`/관리/대시보드` 접두사를 사용합니다(모듈은 경로를 위에서 아래로 - 재귀적으로 - 부모에서 자식으로 연결하므로). 마찬가지로, `MetricsModule` 내에 정의된 각 컨트롤러에는 모듈 수준 접두사 `/admin/metrics`가 추가됩니다.

건강 상태 확인(종료)

터미널 통합은 준비 상태/활력 상태 확인 기능을 제공합니다. 상태 점검은 복잡한 백엔드 설정에 있어 매우 중요합니다. 간단히 말해, 웹 개발 영역에서의 상태 점검은 일반적으로 `https://my-website.com/health/readiness` 같은 특수 주소로 구성됩니다. 서비스나 인프라의 구성 요소(예: Kubernetes)는 이 주소를 지속적으로 확인합니다. 이 주소에 대한 `GET` 요청에서 반환된 HTTP 상태 코드에 따라 서비스는 "비정상" 응답을 수신하면 조치를 취합니다. "정상" 또는 "비정상"의 정의는 제공하는 서비스 유형에 따라 다르므로 Terminus 통합은 일련의 상태 지표를 통해 지원합니다.

예를 들어, 웹 서버가 데이터를 저장하기 위해 MongoDB를 사용하는 경우, MongoDB가 여전히 가동 중인지 여부는 중요한 정보가 될 것입니다. 이 경우, `몽구스헬스 인디케이터`를 사용할 수 있습니다. 올바르게 구성한 경우(나중에 자세히 설명), 상태 확인 주소는 MongoDB가 실행 중인지 여부에 따라 정상 또는 비정상 HTTP 상태 코드를 반환합니다.

시작하기

nestjs/terminus를 시작하려면 필요한 종속성을 설치해야 합니다.

```
npm install --save @nestjs/terminus
```

상태 확인 설정

상태 확인은 상태 지표의 요약을 나타냅니다. 상태 점검은 서비스가 정상 상태인지 비정상 상태인지에 대한 점검을 실행합니다. 할당된 모든 상태 표시기가 실행 중이면 상태 확인이 긍정적입니다. 많은 애플리케이션에 유사한 상태 표시기가 필요하기 때문에 @nestjs/terminus에서는 다음과 같이 미리 정의된 표시기 집합을 제공합니다:

- `HttpHealthIndicator`
- `TypeOrmHealthIndicator`• `몽구스헬스인디케이터`
- `시퀀라이즈헬스인디케이터`• `마이 크로오름헬스인디케이터`• `프리즈마`

헬스인디케이터

- 마이크로서비스건강지표•

GRPCHealthIndicator

- MemoryHealthIndicator
- DiskHealthIndicator

첫 번째 상태 확인을 시작하기 위해 `HealthModule`을 생성하고 `TerminusModule`을 임포트해 보겠습니다.
를 가져오기 배열에 넣습니다.

정보 힌트 `Nest CLI`를 사용하여 모듈을 생성하려면 `$ nest g 모듈 상태`를 실행하기만 하면 됩니다.
명령을 사용합니다.

```

@@파일명 (health.module)

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

모듈 ({
  수입: [종단 모듈]
})

내보내기 클래스 HealthModule {}

```

[Nest CLI](#)를 사용하여 쉽게 설정할 수 있는 [컨트롤러](#)를 사용하여 상태 확인을 실행할 수 있습니다.

```
nest g 컨트롤러 상태
```

정보 애플리케이션에서 종료 후크를 활성화하는 것이 좋습니다. 종료 후크를 활성화하면 종료 통합이 이 수명 주기 이벤트를 사용합니다. 종료 후크에 대한 자세한 내용은 [여기를](#) 참조하세요.

HTTP 상태 확인

[nestjs/terminus](#)를 설치하고 [TerminusModule](#)을 임포트하고 새 컨트롤러를 생성했으면 상태 검사를 만들 준비가 된 것입니다.

[HTTPHealthIndicator](#)를 사용하려면 [@nestjs/axios](#) 패키지가 필요하므로 반드시 설치해야 합니다:

```
npm i --save @nestjs/axios axios
```

이제 [헬스 컨트롤러](#)를 설정할 수 있습니다:

```
@@파일명(health.controller)

'@nestjs/common'에서 { Controller, Get }을 가져옵니다;
'@nestjs/terminus'에서 { HealthCheckService, HttpHealthIndicator, HealthCheck
}를 임포트합니다;

컨트롤러('health')
내보내기 클래스 HealthController { 생성자(
  개인 건강: HealthCheckService, 비공개
  http: HttpHealthIndicator,
) {}

  Get()
  @HealthCheck()
  check() {
    return this.health.check([
      () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),
    ]);
  }
}
```

```

}
@@switch
'@nestjs/common'에서 { Controller, Dependencies, Get }를 임포트하고,
'@nestjs/terminus'에서 { HealthCheckService, HttpHealthIndicator,
HealthCheck }를 임포트합니다;

Controller('health') @Dependencies(HealthCheckService,
HttpHealthIndicator) export class HealthController {
  생성자( 개인 건강,
    개인 http,
  ) { }

  Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.http.pingCheck('nestjs-docs', 'https://docs.nestjs.com'),
    ])
  }
}
}

```

```

@@파일명(health.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/terminus'에서 { TerminusModule } 가져오기;
'@nestjs/axios'에서 { HttpModule } 가져오기;
'./health.controller'에서 { HealthController }를 가져옵니다;

모듈({
  임포트: [TerminusModule, HttpModule], 컨
  트롤러: [HealthController],
})
내보내기 클래스 HealthModule {}

@@switch
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/terminus'에서 { TerminusModule } 가져오기;
'@nestjs/axios'에서 { HttpModule } 가져오기;
'./health.controller'에서 { HealthController }를 가져옵니다;

```

```

모듈({
  임포트: [TerminusModule, HttpModule], 컨
  트롤러: [HealthController],
})
내보내기 클래스 HealthModule {}

```


이제 상태 확인이 <https://docs.nestjs.com> 주소로 GET 요청을 보냅니다. 해당 주소에서 정상 응답을 받으면 <http://localhost:3000/health> 경로에서 200 상태 코드가 포함된 다음 객체를 반환합니다.

```
{
  "상태": "ok", "info":
  {
    "nestjs-docs": {
      "상태": "up"
    }
  },
  "error": {},
  "details": {
    "nestjs-docs": {
      "상태": "up"
    }
  }
}
```

이 응답 객체의 인터페이스는 `@nestjs/terminus` 패키지에서 다음과 같이 액세스할 수 있습니다.

`HealthCheckResult` 인터페이스.

상태	상태 표시기가 실패하면 상태는 '오류'가 됩니다. NestJS 앱이 종료 중이지만 여전히 HTTP 요청을 수락하는 경우 상태 확인의 상태는 '종료 중'입니다.	'error' \ 'ok' \ 'shutting_down'
정보	각 상태 표시기의 정보를 포함하는 객체는 다음과 같습니다. 상태 '업', 즉 "건강"을 반환합니다.	객체
오류	각 상태 표시기의 정보를 포함하는 객체는 다음과 같습니다. 상태 '다운', 즉 "건강하지 않음"을 반환합니다.	object
세부 정보	각 상태 표시기의 모든 정보를 포함하는 객체	객체

특정 HTTP 응답 코드 확인

특정 경우에는 특정 기준을 확인하고 응답의 유효성을 검사해야 할 수도 있습니다. 예를 들어 `https://my-external-service.com` 가 응답 코드 204를 반환한다고 가정해 보겠습니다.

`HttpHealthIndicator.responseCheck`를 사용하면 해당 응답 코드를 구체적으로 확인하고 다른 모든 코드를 비정상적인 것으로 판단할 수 있습니다.

204 이외의 다른 응답 코드가 반환되는 경우 다음 예제는 비정상적인 응답입니다. 세 번째 매개 변수는 응답이 정상(`true`) 또는 비정상(`false`)으로 간주되는지 여부를 부울로 반환하는 함수(동기화 또는 비동기화)를 제공해

야 합니다.

```
@@파일명(health.controller)

// `HealthController` 클래스 내에서

Get()
@HealthCheck()
check() {
    return this.health.check([
        () =>])
}
```

```

        this.http.responseCheck(
            '내-외부-서비스',
            'https://my-external-service.com',
            (res) => res.status === 204,
        ),
    ];
}

```

TypeOrm 상태 표시기

Terminus는 상태 확인에 데이터베이스 검사를 추가하는 기능을 제공합니다. 이 상태 표시기를 시작하려면 [데이터베이스 챕터](#)를 확인하여 애플리케이션 내에서 데이터베이스 연결이 설정되어 있는지 확인해야 합니다.

정보 힌트 뒤에서 `TypeOrmHealthIndicator`는 데이터베이스가 아직 살아 있는지 확인하는 데 자주 사용되는 `SELECT 1`-SQL 명령을 실행하기만 하면 됩니다. Oracle 데이터베이스를 사용하는 경우

```

@@파일명(health.controller)
@Controller('health')
내보내기 클래스 HealthController { 생성자(
    개인 건강: HealthCheckService, 비공개
    db: TypeOrmHealthIndicator,
) {}

Get()
@HealthCheck()
check() {
    return this.health.check([
        () => this.db.pingCheck('database'),
    ]);
}
}

@@스위치
@Controller('health')
@Dependencies(HealthCheckService, TypeOrmHealthIndicator)
export class HealthController {
    생성자( 개인 건강,
        개인 DB,
    ) { }

    Get()
    @HealthCheck()
    healthCheck() {
        return this.health.check([
            () => this.db.pingCheck('database'),
        ]);
    }
}

```


데이터베이스에 연결할 수 있는 경우 이제 요청 시 다음과 같은 JSON 결과를 볼 수 있습니다.

`http://localhost:3000` 에 GET 요청을 보냅니다:

```
{
  "상태": "ok", "info":
  {
    "데이터베이스": {
      "상태": "up"
    }
  },
  "error": {},
  "details": {
    "데이터베이스": {
      "상태": "up"
    }
  }
}
```

앱에서 여러 데이터베이스를 사용하는 경우 각 연결을 `HealthController`에 삽입해야 합니다. 그런 다음 연결 참조를 `TypeOrmHealthIndicator`에 전달하기만 하면 됩니다.

```
@@파일명(health.controller)
@Controller('health')
내보내기 클래스 HealthController { 생성자(
  개인 건강: HealthCheckService, 비공개
  db: TypeOrmHealthIndicator,
  @InjectConnection('albumsConnection')
  비공개 albumsConnection: Connection,
  @InjectConnection()
  private defaultConnection: 연결,
) {}

Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.db.pingCheck('albums-database', { connection:
this.albumsConnection }),
    () => this.db.pingCheck('database', { connection:
this.defaultConnection }),
  ]);
}
}
```

디스크 상태 표시기

DiskHealthIndicator를 사용하면 사용 중인 스토리지의 양을 확인할 수 있습니다. 시작하려면

DiskHealthIndicator를 HealthController에 삽입하세요. 다음 예제에서는

경로의 스토리지 사용량(또는 Windows의 경우 C:\\ 사용 가능)을 확인합니다. 전체의 50%를 초과하는 경우 저장 공간이 부족하면 건강하지 않은 상태 확인으로 응답합니다.

```

@@파일명(health.controller)
@Controller('health')
내보내기 클래스 HealthController { 생성자(
    개인 읽기 전용 상태: HealthCheckService, 비공개 읽기 전
    용 디스크: DiskHealthIndicator,
) {}

Get()
@HealthCheck()
check() {
    return this.health.check([
        () => this.disk.checkStorage('storage', { 경로: '/', 임계값 퍼센트:
0.5 })),
    ]);
}
}

@@스위치
@Controller('health')
@Dependencies(HealthCheckService, DiskHealthIndicator)
export class HealthController {
    constructor(health, disk) {}

    Get()
    @HealthCheck()
    healthCheck() {
        return this.health.check([
            () => this.disk.checkStorage('storage', { 경로: '/', 임계값 퍼센트:
0.5 })),
        ]);
    }
}
}

```

DiskHealthIndicator.checkStorage 함수를 사용하면 고정된 공간도 확인할 수 있습니다. 다음 예는

/my-app/ 경로가 250GB를 초과하는 경우 건강하지 않은 것으로 간주합니다.


```
@@파일명(health.controller)

// `HealthController` 클래스 내에서

Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.disk.checkStorage('storage', {path : '/', threshold: 250
    *})
  ])
}
```

```
1024 * 1024 * 1024, })
    });
}
```

메모리 상태 표시기

프로세스가 특정 메모리 제한을 초과하지 않는지 확인하려면 **메모리 상태** 표시기를 사용할 수 있습니다. 다음 예제는 프로세스의 힙을 확인하는 데 사용할 수 있습니다.

정보 힙은 동적으로 할당된 메모리가 상주하는 메모리 부분(즉, malloc을 통해 할당된 메모리)입니다.

힙에서 할당된 메모리는 다음 중 하나가 발생할 때까지 할당된 상태로 유지됩니다:

- 메모리가 **비어** 있습니다.

프로그램이 종료됩니다.

```
@@파일명 (health.controller)
@Controller('health')
내보내기 클래스 HealthController { 생성자(
    개인 건강: HealthCheckService, 개인 메모리:
    MemoryHealthIndicator,
) {}

Get()
@HealthCheck()
check() {
    return this.health.check([
        () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
    ]);
}
}

@@스위치
@Controller('health')
@Dependencies(HealthCheckService, MemoryHealthIndicator)
export class HealthController {
    constructor(health, memory) {}

    Get()
    @HealthCheck()
    healthCheck() {
        return this.health.check([
            () => this.memory.checkHeap('memory_heap', 150 * 1024 * 1024),
        ])
    }
}
```

`MemoryHealthIndicator.checkRSS`를 사용하여 프로세스의 메모리 RSS를 확인할 수도 있습니다. 이 예제는 프로세스가 150MB를 초과하는 경우 비정상 응답 코드를 반환합니다.

할당되었습니다.

정보 힌트 RSS는 상주 세트 크기이며 해당 프로세스에 할당된 메모리와 RAM에 있는 메모리의 양을 표시하는 데 사용됩니다. 스왑아웃된 메모리는 포함되지 않습니다. 공유 라이브러리의 페이지가 실제로 메모리에 있는 한 공유 라이브러리의 메모리는 포함됩니다. 모든 스택 및 힙 메모리는 포함됩니다.

```
@@파일명(health.controller)

// `HealthController` 클래스 내에서

Get()
@HealthCheck()
check() {
  return this.health.check([
    () => this.memory.checkRSS('memory_rss', 150 * 1024 * 1024),
  ]);
}
```

사용자 지정 상태 표시기

경우에 따라 [@nestjs/terminus](#)에서 제공하는 사전 정의된 상태 표시기가 모든 상태 확인 요구 사항을 충족하지 못할 수 있습니다. 이 경우 필요에 따라 사용자 정의 상태 지표를 설정할 수 있습니다.

사용자 지정 지표를 나타낼 서비스를 만드는 것으로 시작해 보겠습니다. 지표가 어떻게 구조화되는지에 대한 기본적인 이해를 돕기 위해 `DogHealthIndicator` 예제를 만들어 보겠습니다. 이 서비스는 모든 `Dog` 객체의 유형이 `'goodboy'`인 경우 `'up'` 상태를 가져야 합니다. 이 조건이 충족되지 않으면 오류가 발생해야 합니다.

```
@@파일명(dog.health)

'@nestjs/common'에서 { Injectable }을 가져옵니다;
'@nestjs/terminus'에서 { HealthIndicator, HealthIndicatorResult,
HealthCheckError }를 가져옵니다;

내보내기 인터페이스 Dog {
  이름: 문자열;
  유형: 문자열;
}

@Injectable()
export class DogHealthIndicator extends HealthIndicator {
  private dogs: Dog[] = [
    { 이름: '피도', 유형: '굿보이' },
    { 이름: '렉스', 유형: '나쁜남자' },
  ];

  async isHealthy(키: 문자열): Promise<HealthIndicatorResult> { const
    badboys = this.dogs.filter(dog => dog.type === 'badboy'); const
    isHealthy = badboys.length === 0;
    const result = this.getStatus(key, isHealthy, { badboys:
```

```

badboys.length });

    if (isHealthy) {
        결과를 반환합니다
    }
    ;
    새로운 HealthCheckError('도그체크 실패', 결과)를 던집니다;
}
}
@@switch
'@nestjs/common'에서 { Injectable }을 가져옵니다;
'@godaddy/terminus'에서 { HealthCheckError }를 가져옵니다;

@Injectable()
export class DogHealthIndicator extends HealthIndicator {
    dogs = [
        { 이름: '피도', 유형: '굿보이' },
        { 이름: '렉스', 유형: '나쁜남자' },
    ];

    async isHealthy(key) {
        const badboys = this.dogs.filter(dog => dog.type === 'badboy');
        const isHealthy = badboys.length === 0;
        const result = this.getStatus(key, isHealthy, { badboys:
badboys.length });

        if (isHealthy) {
            결과를 반환합니다
        }
        ;
        새로운 HealthCheckError('도그체크 실패', 결과)를 던집니다;
    }
}
}

```

다음으로 해야 할 일은 상태 지표를 공급자로 등록하는 것입니다.

```
@@파일명(health.module)

'@nestjs/common'에서 { Module }을 가져옵니다;

'@nestjs/terminus'에서 { TerminusModule }을 임포트하고,

'./dog.health'에서 { DogHealthIndicator }를 임포트합니다;

모듈({
  컨트롤러: [HealthController], 임포트
  : [TerminusModule], 제공자:
  [DogHealthIndicator]
```

정보 힌트 실제 애플리케이션에서 DogHealthIndicator는 별도의 모듈(예: DogModule)에 제공되어야 하
내보내기 클래스 HealthModule { }
며, 이 모듈은 HealthModule에서 가져올 것입니다.

마지막 필수 단계는 필수 상태 확인 엔드포인트에 현재 사용 가능한 상태 표시기를 추가하는 것입니다. 이를 위해 `HealthController`로 돌아가서 `검사` 함수에 추가합니다.

```

@@파일명 (health.controller)

'@nestjs/terminus'에서 { HealthCheckService, HealthCheck }를 임포트하고
, '@nestjs/common'에서 { Injectable, Dependencies, Get }을 임포트하고,
'./dog.health'에서 { DogHealthIndicator }를 임포트합니다;

@Injectable()
내보내기 클래스 HealthController { 생성자(
  개인 건강: 건강검진 서비스,
  비공개 dogHealthIndicator: DogHealthIndicator
) {}

  Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.dogHealthIndicator.isHealthy('dog'),
    ])
  }
}

@@switch

'@nestjs/terminus'에서 { HealthCheckService, HealthCheck }를 임포트하고
, '@nestjs/common'에서 { Injectable, Get }을 임포트합니다;
'./dog.health'에서 { DogHealthIndicator }를 가져옵니다;

@Injectable()
@Dependencies(HealthCheckService, DogHealthIndicator)
export class HealthController {
  생성자( 개인 건강,
    개인 개건강지표
  ) {}

  Get()
  @HealthCheck()
  healthCheck() {
    return this.health.check([
      () => this.dogHealthIndicator.isHealthy('dog'),
    ])
  }
}

```


종결은 예를 들어 상태 검사에 실패한 경우와 같은 오류 메시지만 기록합니다. `TerminusModule.forRoot()` 메서드를 사용하면 오류를 기록하는 방법을 더 잘 제어할 수 있을 뿐만 아니라 로깅 자체를 완전히 인수할 수 있습니다.

이 섹션에서는 사용자 정의 로거 `TerminusLogger`를 만드는 방법을 안내해 드리겠습니다. 이 로거는 기본 제공 로거를 확장합니다. 따라서 로거의 어느 부분을 덮어쓸지 선택할 수 있습니다.

정보 NestJS의 사용자 정의 로거에 대해 자세히 알아보려면 [여기에서 자세히 읽어보세요](#).

```

@@파일명(terminus-logger.service)

'@nestjs/common'에서 { Injectable, Scope, ConsoleLogger }를 임포트합니다;

주사형({ 범위: Scope.TRANSIENT })
export class TerminusLogger extends ConsoleLogger {
  error(message: any, stack?: string, context?: string): void;
  error(message: any, ...optionalParams: any[]): void;
  error(
    메시지: 알 수 없음,
    스택?: 알 수 없음, 컨
    텍스트?: 알 수 없음,
    ...나머지: 알 수 없음[]
  ): void {
    // 오류 메시지를 기록하는 방법을 여기에 덮어씁니다.
  }
}

```

사용자 정의 로거를 생성한 후에는 해당 로거를

`TerminusModule.forRoot()`를 호출합니다.

```

@@파일명(health.module)
@Module({
  imports: [
    TerminusModule.forRoot({
      로거: TerminusLogger,
    }),
  ],
})

내보내기 클래스 HealthModule {}



```

오류 메시지를 포함하여 Terminus에서 오는 모든 로그 메시지를 완전히 차단하려면 Terminus를 이렇게 구성하세요.

```
@@파일명(health.module)
@Module({
  imports: [
    TerminusModule.forRoot({
      logger: false,
    }),
  ],
})

내보내기 클래스 HealthModule {}
```

터미널을 사용하면 로그에 상태 확인 오류를 표시하는 방법을 구성할 수 있습니다.

오류	설명	예제
로그		
스타일		
일		
json (기본값)	오류 발생 시 상태 검사 결과 요약을 JSON 객체로 출력합니다.	
예쁜	형식이 지정된 상자 내에 오류가 있는 경우 상태 검사 결과의 요약을 인쇄하고 성공/오류 결과를 강조 표시합니다.	

다음 코드조각에서와 같이 `errorLogStyle` 구성 옵션을 사용하여 로그 스타일을 변경할 수 있습니다.

```

@@파일명 (health.module)
@Module({
  imports: [
    TerminusModule.forRoot({
      errorLogStyle: 'pretty',
    }),
  ],
})
내보내기 클래스 HealthModule {}

```

더 많은 예제

더 많은 작업 예제는 [여기에서](#) 확인할 수 있습니다.

CQRS

간단한 **CRUD**(생성, 읽기, 업데이트 및 삭제) 애플리케이션의 흐름은 다음과 같이 설명할 수 있습니다:

- . 컨트롤러 계층은 HTTP 요청을 처리하고 서비스 계층에 작업을 위임합니다.
- . 서비스 계층은 대부분의 비즈니스 로직이 있는 곳입니다.
- . 서비스는 리포지토리/DAO를 사용하여 엔티티를 변경/유지합니다.
- . 엔티티는 값의 컨테이너 역할을 하며, 세터와 게터가 있습니다.

이 패턴은 일반적으로 중소규모 애플리케이션에는 충분하지만, 규모가 크고 복잡한 애플리케이션에는 최선의 선택이 아닐 수 있습니다. 이러한 경우에는 애플리케이션의 요구 사항에 따라 CQRS(명령 및 쿼리 책임 분리) 모델이 더 적합하고 확장성이 높을 수 있습니다. 이 모델의 장점은 다음과 같습니다:

- 관심사 분리. 이 모델은 읽기 작업과 쓰기 작업을 별도의 모델로 분리합니다.
- 확장성. 읽기 및 쓰기 작업을 독립적으로 확장할 수 있습니다.
- 유연성. 이 모델을 사용하면 읽기 및 쓰기 작업에 서로 다른 데이터 저장소를 사용할 수 있습니다.
- 성능. 이 모델을 사용하면 읽기 및 쓰기 작업에 최적화된 다양한 데이터 저장소를 사용할 수 있습니다.

이 모델을 용이하게 하기 위해 Nest는 경량 **CQRS 모듈**을 제공합니다. 이 장에서는 이 모듈을 사용하는 방법

을 설명합니다. 설치

먼저 필요한 패키지를 설치합니다:

```
npm install --save @nestjs/cqrs
```

명령

명령은 애플리케이션 상태를 변경하는 데 사용됩니다. 데이터 중심이 아닌 작업 기반이어야 합니다. 명령이 전송되면 해당 명령 핸들러가 처리합니다. 핸들러는 애플리케이션 상태를 업데이트할 책임이 있습니다.

```
@@파일명(heroes-game.service) @Injectable()
export class HeroesGameService {
  constructor(private commandBus: CommandBus) {}

  async killDragon(heroId: 문자열, killDragonDto: KillDragonDto) {
    return this.commandBus.execute(
      새로운 KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}
@@스위치
@Injectable()
```

```

@Dependencies(CommandBus) export
class HeroesGameService {
  constructor(commandBus) {
    this.commandBus = commandBus;
  }

  async killDragon(heroId, killDragonDto) {
    return this.commandBus.execute(
      새로운 KillDragonCommand(heroId, killDragonDto.dragonId)
    );
  }
}

```

위의 코드 스니펫에서는 `KillDragonCommand` 클래스를 인스턴스화하여 `CommandBus`의 `실행()` 메서드에 전달합니다. 이것이 데모 명령 클래스입니다:

```

@@filename(kill-dragon.command)
export class KillDragonCommand {
  생성자(
    공개 읽기 전용 heroId: 문자열, 공개 읽
    기 전용 dragonId: 문자열,
  ) {}
}
@@switch
export class KillDragonCommand {
  constructor(heroId, dragonId) {
    this.heroId = heroId;
    this.dragonId = dragonId;
  }
}

```

`CommandBus`는 명령 스트림을 나타냅니다. 적절한 핸들러에 명령을 디스패치하는 역할을 담당합니다. `실행()` 메서드는 핸들러가 반환한 값으로 확인되는 프로미스를 반환합니다.

`KillDragonCommand` 명령에 대한 핸들러를 만들어 보겠습니다.

```
@@파일명(kill-dragon.handler)
@CommandHandler(KillDragonCommand) 내보내기
클래스 KillDragonHandler는
ICommandHandler<KillDragonCommand>를 구현
합니다 {
    생성자(비공개 저장소: HeroRepository) {}

    async execute(command: KillDragonCommand) {
        const { heroId, dragonId } = command;
        const hero = this.repository.findOneById(+heroId);

        hero.killEnemy(dragonId);
        await this.repository.persist(hero);
    }
}
```



```

    }
  }
  @@스위치
  @CommandHandler(KillDragonCommand)
  @Dependencies(HeroRepository) 내보내
  기 클래스 KillDragonHandler {
    constructor(repository) {
      this.repository = repository;
    }

    async execute(command) {
      const { heroId, dragonId } = command;
      const hero = this.repository.findOneById(+heroId);

      hero.killEnemy(dragonId);
      await this.repository.persist(hero);
    }
  }
}

```

이 핸들러는 저장소에서 영웅 엔티티를 검색하고 `killEnemy()` 메서드를 호출한 다음 변경 사항을 유지합니다. `KillDragonHandler` 클래스는 `실행()` 메서드를 구현해야 하는 `ICommandHandler` 인터페이스를 구현합니다. `실행()` 메서드는 명령 객체를 인수로 받습니다.

쿼리

쿼리는 애플리케이션 상태에서 데이터를 검색하는 데 사용됩니다. 쿼리는 작업 기반이 아닌 데이터 중심이어야 합니다. 쿼리가 전송되면 해당 쿼리 핸들러가 처리합니다. 핸들러는 데이터를 검색할 책임이 있습니다.

쿼리버스는 명령버스와 동일한 패턴을 따릅니다. 쿼리 핸들러는

`IQueryHandler` 인터페이스에 `@QueryHandler()` 데코레이터를 사용하여 주석을 달

수 있습니다. 이벤트

이벤트는 애플리케이션 상태의 변경 사항을 애플리케이션의 다른 부분에 알리는 데 사용됩니다. 이벤트는 다음과 같습니다.

모델에 의해 디스패치되거나 이벤트버스를 사용하여 직접 디스패치됩니다. 이벤트가 디스패치되면 해당 이벤트 핸들러가 처리합니다. 그러면 핸들러는 예를 들어 읽기 모델을 업데이트할 수 있습니다.

데모를 위해 이벤트 클래스를 만들어 보겠습니다:

```
@@파일명(hero-killed-dragon.event) 내
보내기 클래스 HeroKilledDragonEvent {
    생성자(
        공개 읽기 전용 heroId: 문자열, 공개 읽
        기 전용 dragonId: 문자열,
    ) {}
}

@@switch
내보내기 클래스 HeroKilledDragonEvent {
```

```

    constructor(heroId, dragonId) {
      this.heroId = heroId;
      this.dragonId = dragonId;
    }
  }
}

```

이제 이벤트는 `EventBus.publish()` 메서드를 사용하여 직접 디스패치할 수도 있지만, 모델에서 디스패치할 수도 있습니다. `killEnemy()` 메서드가 호출될 때 `HeroKilledDragonEvent` 이벤트를 디스패치하도록 `Hero` 모델을 업데이트해 보겠습니다.

```

@@파일명(hero.model)
export class Hero extends AggregateRoot {
  constructor(private id: string) {
    super();
  }

  killEnemy(enemyId: 문자열) {
    // 비즈니스 로직
    this.apply(new HeroKilledDragonEvent(this.id, enemyId));
  }
}

@@switch
export class Hero extends AggregateRoot {
  constructor(id) {
    super();
    this.id = id;
  }

  killEnemy(enemyId) {
    // 비즈니스 로직
    this.apply(new HeroKilledDragonEvent(this.id, enemyId));
  }
}

```

`apply()` 메서드는 이벤트를 디스패치하는 데 사용됩니다. 이 메서드는 이벤트 객체를 인자로 받습니다. 하지만 우리 모델은 이벤트버스를 인식하지 못하기 때문에 모델과 연결해야 합니다. 이 작업은 `EventPublisher` 클래스를 사용하여 수행할 수 있습니다.

```
@@파일명(kill-dragon.handler)
@CommandHandler(KillDragonCommand) 내보내기

클래스 KillDragonHandler는
ICommandHandler<KillDragonCommand>를 구현
합니다 {
    생성자(
        비공개 저장소: 비공개 퍼블리셔인
        HeroRepository: 이벤트 퍼블리셔,
    ) {}

    async execute(command: KillDragonCommand) {
        const { heroId, dragonId } = command;
```

```

    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}

@@스위치 @CommandHandler(킬드래곤 커맨드)
@Dependencies(HeroRepository, EventPublisher)
export class KillDragonHandler {
  constructor(repository, publisher) {
    this.repository = repository;
    this.publisher = publisher;
  }

  async execute(command) {
    const { heroId, dragonId } = command;
    const hero = this.publisher.mergeObjectContext(
      await this.repository.findOneById(+heroId),
    );
    hero.killEnemy(dragonId);
    hero.commit();
  }
}

```

이벤트 퍼블리셔#병합 오브젝트 컨텍스트 메서드는 이벤트 퍼블리셔를 제공된 오브젝트에 병합하므로 이제 오브젝트가 이벤트 스트림에 이벤트를 게시할 수 있게 됩니다.

이 예제에서는 모델에서 `commit()` 메서드도 호출합니다. 이 메서드는 미결 이벤트를 디스패치하는 데 사용됩니다. 이벤트를 자동으로 디스패치하려면 `autoCommit` 속성을 `true`로 설정하면 됩니다:

```

export class Hero extends AggregateRoot {
  constructor(private id: string) {
    super();
    this.autoCommit = true;
  }
}

```

이벤트 퍼블리셔를 존재하지 않는 오브젝트가 아닌 클래스에 병합하려는 경우 이벤트 퍼블리셔 `#mergeClassContext` 메서드를 사용할 수 있습니다:

```

const HeroModel = this.publisher.mergeClassContext(Hero);
const hero = new HeroModel('id'); // <-- HeroModel은 클래스입니다

```

이제 `HeroModel` 클래스의 모든 인스턴스는 다음을 사용하지 않고도 이벤트를 게시할 수 있습니다.
메서드를 호출합니다.

또한 **이벤트버스를** 사용하여 수동으로 이벤트를 발생시킬 수도 있습니다:

```
this.eventBus.publish(new HeroKilledDragonEvent());
```

정보 힌트 **이벤트버스는** 인젝터블 클래스입니다.

각 이벤트에는 여러 이벤트 핸들러가 있을 수 있습니다.

```
@@파일명(영웅을 죽인 용.핸들러) @이벤트 핸들러(영웅을 죽인 용 이벤트)
export 클래스 HeroKilledDragonHandler 구현 IEventHandler<HeroKilledDragonEvent>
{
    생성자(비공개 저장소: HeroRepository) {}

    handle(event: HeroKilledDragonEvent) {
        // 비즈니스 로직
    }
}
```

정보 힌트 이벤트 핸들러를 사용하기 시작하면 기존 HTTP 웹 컨텍스트에서 벗어나게 된다는 점에 유의하세요.

- 명령 핸들러의 오류는 기본 제공 **예외 필터**로 여전히 포착할 수 있습니다.
- 이벤트 핸들러의 오류는 예외 필터로 포착할 수 없으므로 수동으로 처리해야 합니다. 간단한 **시도/잡기**,
- 보상 이벤트를 트리거하여 Sagas를 사용하거나 다른 해결 방법을 선택해야 합니다.
- **CommandHandlers**의 HTTP 응답은 여전히 클라이언트로 다시 전송할 수 있습니다.
- 이벤트 핸들러의 HTTP 응답은 불가능합니다. 클라이언트에 정보를 보내려면 **WebSocket**, **SSE** 또는 다른 솔루션을 사용할 수 있습니다.

사가

사가는 이벤트를 수신하고 새로운 명령을 트리거할 수 있는 장기 실행 프로세스입니다. 일반적으로 애플리케이션에서 복잡한 워크플로를 관리하는 데 사용됩니다. 예를 들어, 사용자가 가입할 때 사가는 **UserRegisteredEvent**를 수신하고 사용자에게 환영 이메일을 보낼 수 있습니다.

사가는 매우 강력한 기능입니다. 하나의 사가는 1...* 이벤트를 수신할 수 있습니다. **RxJS** 라이브러리를 사용하면 이벤트 스트림을 필터링, 매핑, 포크, 병합하여 정교한 워크플로를 만들 수 있습니다. 각 사가는 명령 인스턴스를 생성하

는 Observable을 반환합니다. 그런 다음 이 명령은 **CommandBus**에 의해 비동기적으로 전송됩니다.

HeroKilledDragonEvent를 듣고 영웅을 처치하는 사가를 만들어 보겠습니다.

DropAncientItemCommand 명령입니다.

```
@@파일명 (heroes-game.saga)
```

```
@Injectable()
```



```

내보내기 클래스 히어로즈게임사가 { @Saga()
  dragonKilled = (events$: Observable<any>): Observable<ICommand> => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}
@@스위치
@Injectable()
내보내기 클래스 히어로즈게임사가 { @Saga()
  dragonKilled = (events$) => {
    return events$.pipe(
      ofType(HeroKilledDragonEvent),
      map((event) => new DropAncientItemCommand(event.heroId,
fakeItemID)),
    );
  }
}

```

정보 힌트 `ofType` 연산자와 `@Saga()` 데코레이터는 [@nestjs/cqrs](https://github.com/nestjs/cqrs)에서 내보냅니다.

패키지입니다.

`사가()` 데코레이터는 메서드를 사가로 표시합니다. `events$` 인수는 모든 이벤트의 관찰 가능한 스트림입니다.

`ofType` 연산자는 지정된 이벤트 유형에 따라 스트림을 필터링합니다. `map` 연산자는 이벤트를 새 명령 인스턴스에 매핑합니다.

이 예제에서는 `HeroKilledDragonEvent`를 `DropAncientItemCommand` 명령에 매핑합니다. 그러면 `CommandBus`에 의해 `DropAncientItemCommand` 명령이 자동 발송됩니다.

설정

마무리하자면, 모든 명령 핸들러, 이벤트 핸들러, 사가를

히어로즈게임모듈:

@@파일명(heroes-game.module)

```
export const CommandHandlers = [KillDragonHandler,  
DropAncientItemHandler];  
export const EventHandlers = [HeroKilledDragonHandler,  
HeroFoundItemHandler];
```

모듈({

수입: [CqrsModule],

컨트롤러: [히어로즈게임컨트롤러], 공급자: [

히어로즈게임서비스, 히어로즈게임

사가,

...명령 핸들러,

```

        ...이벤트 핸들러, 영웅 저장소,
    ]
})
내보내기 클래스 히어로즈게임모듈 {}

```

처리되지 않은 예외

이벤트 핸들러는 비동기 방식으로 실행됩니다. 즉, 애플리케이션이 일관되지 않은 상태가 되는 것을 방지하기 위해 항상 모든 예외를 처리해야 합니다. 그러나 예외가 처리되지 않으면 이벤트버스는 `UnhandledExceptionInfo` 객체를 생성하고 이를 `UnhandledExceptionBus` 스트림으로 푸시합니다. 이 스트림은 처리되지 않은 예외를 처리하는 데 사용할 수 있는 `Observable`입니다.

```

private destroy$ = new Subject<void>();

constructor(private unhandledExceptionsBus: UnhandledExceptionBus) {
    this.unhandledExceptionsBus
        .pipe(takeUntil(this.destroy$))
        .subscribe((exceptionInfo) => {
            // 여기서 예외 처리

            // 예: 외부 서비스로 전송, 프로세스 종료 또는 새 이벤트 게시
        });
}

onModuleDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
}

```

예외를 필터링하려면 다음과 같이 `ofType` 연산자를 사용할 수 있습니다:

```

this.unhandledExceptionsBus.pipe(takeUntil(this.destroy$),
    UnhandledExceptionBus.ofType(TransactionNotAllowedException)).subscribe((e
xceptionInfo) => {
    // 여기서 예외 처리
});

```

여기서 `TransactionNotAllowedException`은 필터링하려는 예외입니다.

`UnhandledExceptionInfo` 객체에는 다음과 같은 프로퍼티가 포함되어 있습니다:

```
내보내기 인터페이스 UnhandledExceptionInfo<Cause = IEvent | ICommand,  
Exception = any> {  
    /**
```

```

    * 던져진 예외입니다.
    */
    예외: 예외;
    /**
    * 예외의 원인(이벤트 또는 명령 참조).
    */
    원인: 원인;
}

```

모든 이벤트 구독하기

`CommandBus`, `QueryBus`, `EventBus`는 모두 `Observable`입니다. 즉, 전체 스트림을 구독하고 예를 들어 모든 이벤트를 처리할 수 있습니다. 예를 들어 모든 이벤트를 콘솔에 기록하거나 이벤트 저장소에 저장할 수 있습니다.

```

private destroy$ = new Subject<void>();

constructor(private eventBus: EventBus) {
    this.eventBus
        .pipe(takeUntil(this.destroy$))
        .subscribe((event) => {
            // 데이터베이스에 이벤트 저장
        });
}

onModuleDestroy() {
    this.destroy$.next();
    this.destroy$.complete();
}

```

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

문서

Compodoc은 Angular 애플리케이션을 위한 문서화 도구입니다. Nest와 Angular는 유사한 프로젝트 및 코드 구조를 공유하므로 Compodoc은 Nest 애플리케이션에서도 작동합니다.

설정

기존 Nest 프로젝트 내에서 Compodoc을 설정하는 방법은 매우 간단합니다. 먼저 OS 터미널에서 다음 명령으로 개발 종속성을 추가합니다:

```
$ npm i -D @compodoc/compodoc
```

세대

다음 명령을 사용하여 프로젝트 문서를 생성합니다(`npx`를 지원하려면 npm 6이 필요합니다). 더 많은 옵션은 [공식 문서를](#) 참조하세요.

```
$ npx @compodoc/compodoc -p tsconfig.json -s
```

브라우저를 열고 <http://localhost:8080> 으로 이동합니다. 초기 Nest CLI 프로젝트가 표시됩니다:



기여하기

[여기에서](#) Compodoc 프로젝트에 참여하고 기여할 수 있습니다.

프리즈마

[프리즈마](#)는 Node.js 및 TypeScript용 [오픈소스](#) ORM입니다. 일반 SQL을 작성하거나 SQL 쿼리 빌더(예: [knex.js](#)) 또는 ORM(예: [TypeORM](#) 및 [Sequelize](#))과 같은 다른 데이터베이스 액세스 도구를 사용하는 대신에 사용됩니다. 프리즈마는 현재 PostgreSQL, MySQL, SQL Server, SQLite, MongoDB, CockroachDB([미리보기](#))를 지원합니다.

Prisma는 일반 JavaScript와 함께 사용할 수 있지만 TypeScript를 수용하며 TypeScript 에코시스템의 다른 ORM이 보장하는 수준을 뛰어넘는 유형 안전성을 제공합니다. Prisma와 TypeORM의 유형 안전 보장에 대한 심층적인 비교는 [여기에서](#) 확인할 수 있습니다.

정보 참고 Prisma의 작동 방식에 대한 간략한 개요를 확인하려면 [빠른 시작](#)을 따르거나 [문서에서 소개](#)를 읽어보세요. [prisma-examples](#) 리포지토리에 바로 실행할 수 있는 [REST](#) 및 [GraphQL](#) 예제도 있습니다.

시작하기

이 레시피에서는 NestJS와 Prisma를 처음부터 시작하는 방법을 배웁니다. 데이터베이스에서 데이터를 읽고 쓸 수 있는 REST API를 사용하여 샘플 NestJS 애플리케이션을 빌드할 것입니다.

이 가이드에서는 데이터베이스 서버를 설정하는 데 드는 오버헤드를 줄이기 위해 [SQLite](#) 데이터베이스를 사용합니다. PostgreSQL 또는 MySQL을 사용하더라도 이 가이드를 따를 수 있으며, 적절한 위치에서 이러한 데이터베이스 사용에 대한 추가 지침을 확인할 수 있습니다.

정보 참고 기존 프로젝트가 이미 있고 Prisma로 마이그레이션을 고려하고 있다면 [기존 프로젝트에 Prisma 추가](#) 가이드를 따르세요. TypeORM에서 마이그레이션하는 경우 [TypeORM에서 Prisma로 마이그레이션하기](#) 가이드를 참조하세요.

NestJS 프로젝트 생성

시작하려면 NestJS CLI를 설치하고 다음 명령을 사용하여 앱 스켈레톤을 생성합니다:

```
$ npm install -g @nestjs/cli
동지 새 헬로 프리즘
```

이 명령으로 생성된 프로젝트 파일에 대해 자세히 알아보려면 [첫 단계](#) 페이지를 참조하세요. 이제 `npm start`를 실행하여 애플리케이션을 시작할 수 있다는 점도 참고하세요. <http://localhost:3000/> 에서 실행되는 REST API는 현재 `src/app.controller.ts`에 구현된 단일 경로를 제공합니다. 이 가이드에서는 *사용자* 및 *글*에 대한 데이터를 저장하고 검색하는 추가 경로를 구현할 것입니다.

Prisma 설정

프로젝트에 개발 종속 요소로 Prisma CLI를 설치하여 시작하세요:

```
$ cd hello-prisma
$ npm 설치 프리즈마 --save-dev
```


다음 단계에서는 **프리즈마 CLI**를 활용하겠습니다. 모범 사례로, 접두사 앞에 **npx**를 붙여 로컬에서 CLI를 호출하는 것이 좋습니다:

```
엔엑스피 프리즈마
```

▶ 실을 사용하는 경우 확장

Yarn을 사용하는 경우 다음과 같이 Prisma CLI를 설치할 수 있습니다:

```
yarn 추가 프리즈마 --dev
```

설치가 완료되면 앞에 **yarn**을 붙여 호출할 수 있습니다:

```
원사 프리즈마
```

이제 Prisma CLI의 **init** 명령을 사용하여 초기 Prisma 설정을 생성합니다:

```
npx 프리즈마 초기화
```

이 명령은 다음 내용을 포함하는 새 **prisma** 디렉터리를 만듭니다:

- **schema.prisma**: 데이터베이스 연결을 지정하고 데이터베이스 스키마를 포함합니다.
- **.env**: 일반적으로 데이터베이스 자격 증명을 환경 변수 그룹에 저장하는 데 사용되는 **dotenv** 파일입니다.

데이터베이스 연결 설정

데이터베이스 연결은 **schema.prisma** 파일의 **데이터 소스** 블록에서 구성됩니다. 기본적으로 **포스트그레스큐엘**로 설정되어 있지만 이 가이드에서는 SQLite 데이터베이스를 사용하므로 **데이터 소스** 블록의 **공급자** 필드를 **sqlite**로 조정해야 합니다:

```
데이터 소스 db { 공급자 =  
  "sqlite"  
  url= env("DATABASE_URL")  
}  
  
제너레이터 클라이언트 {  
  공급자 = "prisma-client-js"  
}
```

이제 `.env`를 열고 `DATABASE_URL` 환경 변수를 조정하여 다음과 같이 표시되도록 합니다:

```
DATABASE_URL="file:./dev.db"
```

구성한 ConfigModule이 있는지 확인하세요. 그렇지 않으면 `.env`에서 `DATABASE_URL` 변수가 선택되지 않습니다.

SQLite 데이터베이스는 단순한 파일이므로 서버가 필요하지 않습니다. 따라서 호스트와 *포트로* 연결 URL을 구성하는 대신 로컬 파일(이 경우 `dev.db`)을 가리키기만 하면 됩니다. 이 파일은 다음 단계에서 생성됩니다.

► PostgreSQL 또는 MySQL을 사용하는 경우 확장하기

PostgreSQL 및 MySQL을 사용하는 경우 *데이터베이스 서버*를 가리키도록 연결 URL을 구성해야 합니다. 필요한 연결 URL 형식에 대한 자세한 내용은 [여기에서](#) 확인할 수 있습니다.

PostgreSQL

PostgreSQL을 사용하는 경우 다음과 같이 `schema.prisma` 및 `.env` 파일을 조정해야 합니다:

`schema.prisma`

```
데이터 소스 DB {
  공급자 = "postgresql"
  url= env("DATABASE_URL")
}

제너레이터 클라이언트 {
  공급자 = "prisma-client-js"
}
```

`.env`

```
DATABASE_URL="postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=SCHEMA"
```

모든 대문자로 된 자리 표시자의 철자를 데이터베이스 자격 증명으로 바꿉니다. `SCHEMA` 자리 표시자에 무엇을 입력해야 할지 잘 모르겠다면 기본값인 `public`을 입력하는 것이 가장 좋습니다:

```
DATABASE_URL="postgresql://USER:PASSWORD@HOST:PORT/DATABASE?schema=public"
```

PostgreSQL 데이터베이스를 설정하는 방법을 배우고 싶으시다면, [Heroku에서 무료 PostgreSQL 데이터베이스 설정 가이드](#)를 참조하세요.

MySQL

MySQL을 사용하는 경우 다음과 같이 `schema.prisma` 및 `.env` 파일을 조정해야 합니다:

schema.prisma

```

데이터 소스 db { 공급자
  = "mysql"
  url= env("DATABASE_URL")
}

제너레이터 클라이언트 {
  공급자 = "prisma-client-js"
}

```

.env

```
DATABASE_URL="mysql://USER:PASSWORD@HOST:PORT/DATABASE"
```

모든 대문자로 표기된 자리 표시자를 데이터베이스 자격 증명으로 바꿉니다. Prisma 마이그

레이션으로 두 개의 데이터베이스 테이블 만들기

이 섹션에서는 [Prisma 마이그레이션](#)을 사용하여 데이터베이스에 두 개의 새 테이블을 생성합니다. 프리즈마 마이그레이션

은 프리즈마 스키마에서 선언적 데이터 모델 정의를 위한 SQL 마이그레이션 파일을 생성합니다. 이러한 마이그레이션 파일은 완전히 사용자 정의할 수 있으므로 기본 데이터베이스의 추가 기능을 구성하거나 시딩과 같은 추가 명령을 포함할 수 있습니다.

다음 두 모델을 `schema.prisma` 파일에 추가합니다:

```

모델 사용자 {
  id      Int      기본값(자동증가()) id
  이메일  문자열
  이름    문자열?   고유
} 게시물 게시물[]

모델 포스트 {
  id      Int      기본값(자동증가()) id
  title    자열
  콘텐츠  부울인라인?@default(false)
  작성자   사용자   관계(필드: [authorId], 참조: [id])
} authorId ?
          Int?

```

Prisma 모델이 준비되면 SQL 마이그레이션 파일을 생성하고 데이터베이스에 대해 실행할 수 있습니다. 터미널에서 다음 명령을 실행합니다:

```
$ npx prisma 마이그레이션 개발 --이름 초기화
```

이 **prisma 마이그레이션 개발** 명령은 SQL 파일을 생성하고 데이터베이스에 대해 직접 실행합니다. 이 경우 기존 **prisma** 디렉터리에 다음과 같은 마이그레이션 파일이 생성되었습니다:

```
트리 프리즘 프리즘
├─ dev.db
├─ 마이그레이션
│   └─ 20201207100915_init
│       └─ migration.sql
└─ schema.prisma
```

▶ 확장하여 생성된 SQL 문을 확인합니다.

다음 표는 SQLite 데이터베이스에서 생성되었습니다:

```
-- CreateTable CREATE
TABLE "User" (
  "id" INTEGER NOT NULL PRIMARY KEY 자동 인크루먼트,
  "email" TEXT NOT NULL,
  "이름" 텍스트
);

-- CreateTable CREATE
TABLE "Post" (
  "id" INTEGER NOT NULL PRIMARY KEY 자동인크립션, "title"
  TEXT NOT NULL,
  "콘텐츠" 텍스트,
  "published" BOOLEAN DEFAULT false,
  "authorId" INTEGER,

  FOREIGN KEY("authorId") REFERENCES "User"("id") ON DELETE SET NULL ON
  UPDATE CASCADE
);

-- CreateIndex
"User"("이메일")에 고유 인덱스 "User.email_unique"를 만듭니다;
```

프리즈마 클라이언트 설치 및 생성

Prisma 클라이언트는 Prisma 모델 정의에서 생성되는 유형 안전 데이터베이스 클라이언트입니다. 이러한 접근 방

식 덕분에 Prisma 클라이언트는 모델에 *맞게* 특별히 *맞춤화된* **CRUD** 작업을 노출할 수 있습니다.

프로젝트에 프리즈마 클라이언트를 설치하려면 터미널에서 다음 명령을 실행하세요:


```
npm 설치 @prisma/client
```

설치하는 동안 Prisma는 자동으로 `prisma 생성` 명령을 호출합니다. 앞으로는 Prisma 모델을 변경할 때마다 명령을 실행하여 생성된 Prisma 클라이언트를 업데이트해야 합니다.

정보 `prisma 생성` 명령은 프리즈마 스키마를 읽고 생성된 프리즈마 클라이언트 라이브러리를 `node_modules/@prisma/client` 내부에 업데이트합니다.

NestJS 서비스에서 프리즈마 클라이언트 사용

이제 프리즈마 클라이언트로 데이터베이스 쿼리를 전송할 수 있습니다. 프리즈마 클라이언트로 쿼리를 작성하는 방법에 대해 자세히 알아보려면 [API 설명서](#)를 확인하세요.

NestJS 애플리케이션을 설정할 때 서비스 내에서 데이터베이스 쿼리를 위한 Prisma 클라이언트 API를 추상화할 수 있습니다. 시작하려면 `PrismaClient` 인스턴스와 및 데이터베이스 연결을 처리하는 새 `PrismaService`를 생성하면 됩니다.

`src` 디렉터리 내에 `prisma.service.ts`라는 새 파일을 생성하고 다음 코드를 추가합니다:

```
'@nestjs/common'에서 { Injectable, OnModuleInit }를 임포트하고,
'@prisma/client'에서 { PrismaClient }를 임포트합니다;

@Injectable()
export class PrismaService extends PrismaClient implements OnModuleInit {
  async onModuleInit() {
    await this.$connect();
  }
}
```

정보 참고 `onModuleInit`은 선택 사항이며, 이를 생략하면 Prisma는 데이터베이스에 처음 호출할 때 느리게 연결됩니다.

다음으로 Prisma 스키마에서 `사용자` 및 `게시물` 모델에 대한 데이터베이스 호출을 수행하는 데 사용할 수 있는 서비스를 작성할 수 있습니다.

여전히 `src` 디렉터리 안에 `user.service.ts`라는 새 파일을 만들고 다음 코드를 추가합니다:

```
'@nestjs/common'에서 { Injectable }을 가져오고,  
'./prisma.service'에서 { PrismaService }를 가져오고,  
'@prisma/client'에서 { 사용자, 프리즈마 }를 가져옵니다;  
  
@Injectable()  
사용자 서비스 클래스 보내기 {  
  constructor(private prisma: PrismaService) {}  
  
  async user(  

```

```

    사용자 위치 고유 입력: Prisma.UserWhereUniqueInput,
  ): Promise<사용자 | null> {
    return this.prisma.user.findUnique({
      where: userWhereUniqueInput,
    });
  }

  async users(params: {
    skip?: 숫자; take?:
    숫자;
    cursor?: Prisma.UserWhereUniqueInput;
    where?: Prisma.UserWhereInput;
    orderBy?: Prisma.UserOrderByWithRelationInput;
  }): 약속<사용자[]> {
    const { skip, take, cursor, where, orderBy } = params;
    return this.prisma.user.findMany({
      건너뛰기,
      취하다, 커
      서, 어디,
      주문기준,
    });
  }

  async createUser(데이터: Prisma.UserCreateInput): Promise<User> {
    return this.prisma.user.create({
      데이터,
    });
  }

  async updateUser(params: {
    where: Prisma.UserWhereUniqueInput;
    데이터: Prisma.UserUpdateInput;
  }): 약속<사용자> {
    const { where, data } = params;
    return this.prisma.user.update({
      데이터,
      위치,
    });
  }

  async deleteUser(where: Prisma.UserWhereUniqueInput): Promise<User> {
    return this.prisma.user.delete({
      어디에,
    });
  }
}

```

프리즈마 클라이언트에서 생성된 유형을 사용하여 서비스에 노출되는 메소드가 올바르게 입력되었는지 확인합니다. 따라서 모델을 입력하고 추가 인터페이스 또는 DTO 파일을 생성하는 번거로움을 줄일 수 있습니다.

이제 **포스트** 모델에 대해서도 동일한 작업을 수행합니다.

여전히 **src** 디렉터리 안에 **post.service.ts**라는 새 파일을 만들고 다음 코드를 추가합니다:

```
'@nestjs/common'에서 { Injectable } 가져오기;
'./prisma.service'에서 { PrismaService } 가져오기;
'@prisma/client'에서 { Post, Prisma } 가져오기;

@Injectable()
내보내기 클래스 PostService {
  constructor(private prisma: PrismaService) {}

  비동기 포스트(
    postWhereUniqueInput: Prisma.PostWhereUniqueInput,
  ): 약속<포스트 | 널> {
    return this.prisma.post.findUnique({
      where: postWhereUniqueInput,
    });
  }

  async posts(params: {
    skip?: 숫자; take?:
    숫자;
    cursor?: Prisma.PostWhereUniqueInput;
    where?: Prisma.PostWhereInput;
    orderBy?: Prisma.PostOrderByWithRelationInput;
  }): Promise<Post[]> {
    const { skip, take, cursor, where, orderBy } = params;
    return this.prisma.post.findMany({
      건너뛰기,
      취하다, 커
      서, 어디,
      주문기준,
    });
  }

  async createPost(데이터: Prisma.PostCreateInput): Promise<Post> {
    return this.prisma.post.create({
      데이터,
    });
  }

  async updatePost(params: {
    where: Prisma.PostWhereUniqueInput;
```

```
    데이터: Prisma.PostUpdateInput;  
  }): 약속<포스트> {  
    const { 데이터, where } = 매개변수;  
    return this.prisma.post.update({  
      데이터,  
      위치,  
    });  
  }  
}
```

```
async deletePost(where: Prisma.PostWhereUniqueInput): Promise<Post> {  
  return this.prisma.post.delete({  
    어디에,  
  });  
}
```

현재 **사용자 서비스** 및 **포스트 서비스**는 프리즈마 클라이언트에서 사용할 수 있는 CRUD 쿼리를 래핑합니다. 실제 애플리케이션에서 서비스는 애플리케이션에 비즈니스 로직을 추가하는 장소이기도 합니다. 예를 들어 **사용자 서비스** 내에 사용자의 비밀번호 업데이트를 담당하는 **updatePassword**라는 메서드가 있을 수 있습니다.

기본 앱 컨트롤러에서 REST API 경로 구현하기

마지막으로 이전 섹션에서 만든 서비스를 사용하여 앱의 다양한 경로를 구현합니다. 이 가이드에서는 모든 경로를 이미 존재하는 **AppController** 클래스에 넣겠습니다.

app.controller.ts 파일의 내용을 다음 코드로 바꿉니다:

```

가져 오기 {
  Controller,
  Get,
  매개변수,
  포스트,
  본문, 넣
  기, 삭제
},
}를 '@nestjs/common'에서 가져옵니다;
'./user.service'에서 { UserService }를 가져오고,
'./post.service'에서 { PostService }를 가져옵니다;
'@prisma/client'에서 { 사용자 사용자모델, 포스트 포스트모델 }을 가져옵니다;

컨트롤러()
내보내기 클래스 AppController {
  생성자(
    비공개 읽기 전용 사용자 서비스: UserService, 비공
    개 읽기 전용 postService: 포스트서비스,
  ) {}

  @Get('post/:id')
  async getPostById(@Param('id') id: 문자열): Promise<PostModel> {
    return this.postService.post({ id: Number(id) });
  }

  @Get('feed')
  비동기 getPublishedPosts(): Promise<PostModel[]> {
    return this.postService.posts({
      where: { 게시됨: true },
    });
  }
}

```



```

@Get('filtered-posts/:searchString')
async getFilteredPosts(
  Param('searchString') searchString: 문자열,
): Promise<PostModel[]> {
  return this.postService.posts({
    where: {
      또는: [
        {
          title: { 포함: searchString },
        },
        {
          콘텐츠: { 포함: 검색 문자열 },
        },
      ],
    },
  });
}

@Post('post')
async createDraft(
  Body() postData: { 제목: 문자열; 내용?: 문자열; 작성자 이메일: 문자열 },
): 약속<포스트모델> {
  const { title, content, authorEmail } = postData;
  return this.postService.createPost({
    제목, 콘텐츠, 작
    성자: {
      연결합니다: { 이메일: 작성자 이메일 },
    },
  });
}

@Post('user')
async signupUser(
  @Body() 사용자데이터: { 이름?: 문자열; 이메일: 문자열 },
): 약속<사용자 모델> {
  this.userService.createUser(userData)를 반환합니다;
}

@Put('publish/:id')
async publishPost(@Param('id') id: 문자열): Promise<PostModel> {
  return this.postService.updatePost({
    where: { id: Number(id) },
    data: { published: true },
  });
}

삭제('post/:id')

```

```
    async deletePost(@Param('id') id: 문자열): Promise<PostModel> {  
        return this.postService.deletePost({ id: Number(id) });  
    }  
}
```

이 컨트롤러는 다음 경로를 구현합니다:

GET

- `/post/:id`: 아이디로 단일 게시물 가져오기
- `/피드`: 게시된 모든 글 가져오기
- `/필터-게시물/:검색 문자열: 제목 또는 콘텐츠로 글 필터링`

POST

- `/post`: 새 글 작성
 - 본문:
 - 제목: 문자열 (필수): 글의 제목
 - 콘텐츠입니다: 문자열 (선택 사항): 글의 콘텐츠
 - 작성자 이메일: 문자열 (필수): 글을 작성한 사용자의 이메일입니다.
- `/user`: 새 사용자 만들기

본문:

- 이메일: 문자열 (필수): 사용자의 이메일 주소
- 이름: 문자열 (선택 사항): 사용자의 이름

PUT

- `/publish/:id`: 해당 아이디로 글 게시

삭제

- `/post/:id`: 해당 아이디로 글 삭제

요약

이 레시피에서는 Prisma를 NestJS와 함께 사용하여 REST API를 구현하는 방법을 배웠습니다. API의 경로를 구현하는 컨트롤러는 `PrismaService`를 호출하고, 이 컨트롤러는 다시 Prisma 클라이언트를 사용하여 들어오는 요청의 데이터 요구를 충족하기 위해 데이터베이스에 쿼리를 보냅니다.

Prisma에서 NestJS를 사용하는 방법에 대해 자세히 알아보려면 다음 리소스를 확인하세요:

- ◆ [NestJS 및 프리즈마](#)
- ◆ [바로 실행 가능한 REST 및 GraphQL용 예제 프로젝트](#)
- ◆ [프로덕션 지원 스타터 키트](#)
- ◆ [비디오: Prisma로 NestJS를 사용하여 데이터베이스에 액세스하기\(5분\)](#) 작성자: [마크 스탬머요한\(Marc Stammerjohann\)](#)

정적 제공

SPA(단일 페이지 애플리케이션)와 같은 정적 콘텐츠를 제공하기 위해 `ServeStaticModule`을 사용할 수 있습니다.

를 사용하여 `@nestjs/serve-static` 패키지를 생성합

니다. 설치

먼저 필요한 패키지를 설치해야 합니다:

```
npm install --save @nestjs/serve-static
```

부트스트랩

설치 프로세스가 완료되면 `ServeStaticModule`을 루트 `AppModule`로 가져올 수 있습니다.

를 생성하고 구성 객체를 `forRoot()` 메서드에 전달하여 구성합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./app.controller'에서 { AppController }를 임포트하고,
'./app.service'에서 { AppService }를 임포트합니다;
'@nestjs/serve-static'에서 { ServeStaticModule }을 가져오고,
'path'에서 { join }을 가져옵니다;

모듈({ import: [
  ServeStaticModule.forRoot({
    rootPath: join(__dirname, '..', 'client'),
  }),
],
  컨트롤러: [AppController], 공급자: [
    앱서비스],
})
내보내기 클래스 AppModule {}
```

이 상태로 정적 웹사이트를 빌드하고 루트 경로로 지정된 위치에 콘텐츠를 배치합니다.

속성을 설정

합니다. 구성

ServeStaticModule은 다양한 옵션으로 구성하여 동작을 사용자 정의할 수 있습니다. 설정할 수 있습니다.

경로를 사용하여 정적 앱을 렌더링하고, 제외 경로를 지정하고, 캐시 제어 응답 헤더 설정을 활성화 또는 비활성화할 수 있습니다. 전체 옵션 목록은 [여기에서](#) 확인하세요.

경고 정적 앱의 기본 렌더경로는 `*`(모든 경로)이며, 모듈은 응답으로 "index.html" 파일을 전송합니다. 이를 통해 SPA에 대한 클라이언트 측 라우팅을 생성할 수 있습니다. 컨트롤러에 지정된 경로는 서버로 폴백됩니다. 이 동작을 다른 옵션과 결합하여 `serveRoot`, `renderPath`를 설정하여 변경할 수 있습니다.

예

작동하는 예제는 [여기에서](#) 확인할 수 있습니다.

등지 사령관

[독립형 애플리케이션](#) 문서를 확장하면 일반적인 Nest 애플리케이션과 유사한 구조로 명령줄 애플리케이션을 작성할 수 있는 [nest-commander](#) 패키지도 있습니다.

정보 [nest-commander](#)는 타사 패키지이며 NestJS 핵심 팀 전체가 관리하지 않습니다. 라이브러리와 관련된 문제가 발견되면 [해당 리포지토리에](#) 보고해 주세요.

설치

다른 패키지와 마찬가지로 사용하려면 먼저 설치해야 합니다.

```
$ npm i nest-commander
```

명령 파일

[nest-commander](#)를 사용하면 클래스의 경우 [@Command\(\)](#) 데코레이터를, 해당 클래스의 메서드의 경우 [@Option\(\)](#) 데코레이터를 통해 [데코레이터](#)를 사용하여 새 명령줄 애플리케이션을 쉽게 작성할 수 있습니다. 모든 명령 파일은 [CommandRunner](#) 추상 클래스를 구현해야 하며 [@Command\(\)](#) 데코레이터로 데코레이션해야 합니다.

모든 명령은 Nest에서 [@Injectable\(\)](#)로 간주되므로 일반적인 의존성 주입은 여전히 예상대로 작동합니다. 주목해야 할 것은 각 명령이 구현해야 하는 [CommandRunner](#) 추상 클래스뿐입니다. [CommandRunner](#) 추상 클래스는 모든 명령에 [Promise<void>](#)를 반환하고 매개변수 [string\[\], Record<string, any>](#)를 받는 [run](#) 메서드를 갖도록 합니다. [실행](#) 명령은 모든 로직을 시작할 수 있는 곳이며, 옵션 플래그와 일치하지 않는 매개변수를 받아 배열로 전달하므로 여러 매개변수로 작업하려는 경우에 대비해 사용할 수 있습니다. 옵션의 경우 [Record<string, any>](#), 이러한 프로퍼티의 이름은 [@Option\(\)](#) 데코레이터에 지정된 [이름](#) 프로퍼티와 일치하고, 그 값은 옵션 핸들러의 반환과 일치합니다. 더 나은 유형 안전성을 원한다면 옵션에 대한 인터페이스도 만들 수 있습니다.

명령 실행

NestJS 애플리케이션에서 [NestFactory](#)를 사용하여 서버를 생성하고 [listen](#)을 사용하여 서버를 실행하는 것과

유사하게, `nest-commander` 패키지는 서버를 실행하기 위해 사용하기 쉬운 API를 노출합니다.

`CommandFactory`를 가져와서 정적 메서드 `run`을 사용하고 애플리케이션의 루트 모듈에 전달하세요. 이것은 아마도 아래와 같이 보일 것입니다.

```
'nest-commander'에서 { CommandFactory }를 임포트하고,  
'./app.module'에서 { AppModule }을 임포트합니다;  
  
비동기 함수 부트스트랩() {  
  await CommandFactory.run(AppModule);  
}
```

```
부트스트랩();
```

기본적으로 `CommandFactory`를 사용할 때 Nest의 로거는 비활성화되어 있습니다. 하지만 `실행` 함수의 두 번째 인수로 로거를 제공할 수 있습니다. 사용자 정의 NestJS 로거를 제공하거나 유지하려는 로그 수준의 배열을 제공할 수 있습니다. Nest의 오류 로그만 인쇄하려는 경우 여기에 `['error']`를 제공하는 것이 유용할 수 있습니다.

```
'nest-commander'에서 { CommandFactory }를 임포트하고,  
'./app.module'에서 { AppModule }을 임포트합니다;  
import { LogService } './log.service';  
  
비동기 함수 부트스트랩() {  
  await CommandFactory.run(AppModule, new LogService());  
  
  // 또는 Nest의 경고와 오류만 출력하려면  
  CommandFactory.run(AppModule, ['경고', '오류'])을 기다립니다;  
}
```

```
부트스트랩();
```

그게 다입니다. 내부적으로 `CommandFactory`는 `NestFactory`를 호출하고 필요할 때 `app.close()`를 호출하므로 메모리 누수에 대해 걱정할 필요가 없습니다. 오류 처리를 추가해야 하는 경우, `실행` 명령을 래핑하는 `try/catch`를 사용하거나 `.catch()` 메서드를 `부트스트랩()` 호출에 연결할 수 있습니다.

테스트

아주 쉽게 테스트할 수 없다면 아주 멋진 명령줄 스크립트를 작성하는 것이 무슨 소용이 있을까요? 다행히도 `nest-commander`에는 NestJS 에코시스템에 완벽하게 맞는 몇 가지 유틸리티를 사용할 수 있으며, 모든 네슬링이 집처럼 편안하게 사용할 수 있습니다. 테스트 모드에서 명령을 빌드할 때 `CommandFactory`를 사용하는 대신 `CommandTestFactory`를 사용하고 메타데이터를 전달할 수 있는데, 이는 `@nestjs/testing`의 `Test.createTestingModule`이 작동하는 방식과 매우 유사합니다. 실제로 내부적으로 이 패키지를 사용합니다. 또한 `compile()`을 호출하기 전에 `overrideProvider` 메서드를 체인에 연결할 수 있으므로 테스트에서 바로 DI 조각을 교체할 수 있습니다.

모든 것을 종합하기

다음 클래스는 하위 명령을 `기본으로` 받거나 직접 호출할 수 있는 CLI 명령과 동일하며, `-n`, `-s` 및 `-b`(긴 플래그와

함께)가 모두 지원되고 각 옵션에 대한 사용자 정의 구문 분석기가 있습니다. 커맨더와 마찬가지로 `--help` 플래그도 지원됩니다.

```
'nest-commander'에서 { Command, CommandRunner, Option } 가져오기;
```

```
'./log.service'에서 { LogService } 가져오기;
```

```
인터페이스 BasicCommandOptions { 문자열
```

```
  ? : 문자열;
```

```
  부울? : 부울; 숫자? : 숫자
```

```
  ;
```

```

}

Command({ name: 'basic', description: '매개변수 파싱' }) export
class BasicCommand extends CommandRunner {
  constructor(private readonly logService: LogService) {
    super()
  }

  비동기 실행(
    passedParam: 문자열[], options?
    기본 명령 옵션,
  ): 약속<무효> {
    if (options?.boolean !== undefined && options?.boolean !== null) {
      this.runWithBoolean(passedParam, options.boolean);
    } else if (options?.number) {
      this.runWithNumber(passedParam, options.number);
    } else if (options?.string) {
      this.runWithString(passedParam, options.string);
    } else {
      this.runWithNone(passedParam);
    }
  }

  @Option({
    플래그: '-n, --number [숫자]', 설명: '기본 숫
    자 구문 분석기',
  })
  parseNumber(val: 문자열): number {
    return Number(val);
  }

  @Option({
    플래그: '-s, --string [문자열]', 설명: '문자열
    반환',
  })
  parseString(val: 문자열): 문자열 {
    return val;
  }

  @Option({
    플래그: '-b, --boolean [부울]', 설명: '부울 구
    문 분석기',
  })
  parseBoolean(val: 문자열): boolean {
    return JSON.parse(val);
  }
}

```

```
runWithString(매개변수: 문자열[], 옵션: 문자열): void { this.logService.log({  
  매개변수, 문자열: 옵션 });  
}
```

```
runWithNumber(매개변수: 문자열[], 옵션: 숫자): void { this.logService.log({  
  매개변수, 숫자: 옵션 });  
}
```

```
runWithBoolean(param: string[], option: boolean): void {
    this.logService.log({ param, boolean: option });
}

runWithNone(매개변수: 문자열[]): void {
    this.logService.log({ 매개변수 });
}
}
```

명령 클래스가 모듈에 추가되었는지 확인합니다.

```
모듈({
    공급자: [LogService, BasicCommand],
})
내보내기 클래스 AppModule {}
```

이제 main.ts에서 CLI를 실행하려면 다음을 수행할 수 있습니다.

```
비동기 함수 부트스트랩() {
    await CommandFactory.run(AppModule);
}

부트스트랩();
```

이렇게 하면 명령줄 애플리케이션이 완성됩니다. 자세한 정

보

자세한 정보, 예제 및 API 설명서를 보려면 [nest-commander 문서 사이트](#)를 방문하세요.

비동기 로컬 스토리지

AsyncLocalStorage는 함수 매개변수로 명시적으로 전달하지 않고도 애플리케이션을 통해 로컬 상태를 전파할 수 있는 대체 방법을 제공하는 [Node.js API](#)([async_hooks](#) API 기반)입니다. 다른 언어의 스레드 로컬 스토리지와 유사합니다.

비동기 로컬 저장소의 주요 아이디어는 일부 함수 호출을 `AsyncLocalStorage#run` 호출로 래핑할 수 있다는 것입니다. 래핑된 호출 내에서 호출되는 모든 코드는 각 호출 체인에 고유한 동일한 [저장소](#)에 액세스하게 됩니다.

NestJS의 맥락에서 이는 요청의 라이프사이클 내에서 요청의 나머지 코드를 래핑할 수 있는 위치를 찾을 수 있다면 해당 요청에만 표시되는 상태에 액세스하고 수정할 수 있다는 의미이며, 이는 요청 범위 제공자 및 일부 제한 사항에 대한 대안으로 사용될 수 있습니다.

또는 ALS를 사용하여 컨텍스트를 서비스 전체에 명시적으로 전달하지 않고 시스템의 일부(예: *트랜잭션* 객체)에 대해서만 컨텍스트를 전파하여 격리 및 캡슐화를 강화할 수 있습니다.

사용자 지정 구현

NestJS 자체는 `AsyncLocalStorage`에 대한 내장 추상화를 제공하지 않으므로 전체 개념을 더 잘 이해하기 위해 가장 간단한 HTTP 사례에 대해 직접 구현하는 방법을 살펴 보겠습니다:

정보 정보 기성 [전용 패키지](#)에 대해서는 아래를 계속 읽어보세요.

- 먼저 공유 소스 파일에 `AsyncLocalStorage`의 새 인스턴스를 생성합니다. NestJS를 사용하고 있으므로 사용자 정의 공급자를 사용하여 모듈로 전환해 보겠습니다.

```
@@파일명(als.module) @Module({
  공급자: [
    {
      제공: AsyncLocalStorage, useValue:
        new AsyncLocalStorage(),
    },
  ],
  내보내기: [AsyncLocalStorage],
})
내보내기 클래스 AlsModule {}
```

정보 힌트 `AsyncLocalStorage`는 [async_hook](#)에서 가져옵니다.

. 여기서는 HTTP에만 관심이 있으므로 미들웨어를 사용하여 다음 함수를 `AsyncLocalStorage#run`으로 래핑해 보겠습니다. 미들웨어는 요청이 가장 먼저 닿는 곳이기 때문에 모든 인핸서와 나머지 시스템에서 `저장소`를 사용할 수 있게 됩니다.


```

@@파일명(app.module) @Module({
  imports: [AlsModule] 공급자:
  [CatService], 컨트롤러:
  [CatController],
})
내보내기 클래스 AppModule은 NestModule을 구현합니다 { 생성자(
  // 모듈 생성자에 AsyncLocalStorage를 비공개 읽기 전용으로 삽입합니다
  : AsyncLocalStorage
) {}

configure(consumer: MiddlewareConsumer) {
  // 미들웨어, 소비자 바인딩
  .apply((req, res, next) => {
    // 몇 가지 기본값으로 스토어 채우기
    // 요청에 따라, const
    store = {
      userId: req.headers['X-user-id'],
    };
    // 그리고 "next" 함수를 콜백으로 전달합니다.
    //를 스토어와 함께 "als.run" 메서드에 추가합니다. this.als.run(store, ()
    => next());
  })
  // 모든 경로에 등록합니다 (Fastify의 경우 '(.*)' 사용).
  .forRoutes('*');
}
}
@@switch
@Module({
  imports: [AlsModule] 공급자:
  [CatService], 컨트롤러:
  [CatController],
})
@Dependencies(AsyncLocalStorage)
export 클래스 AppModule {
  constructor(as) {
    // 모듈 생성자, this.als = als에 AsyncLocalStorage를 삽입합니다.
  }

  configure(consumer) {
    // 미들웨어, 소비자 바인딩
    .apply((req, res, next) => {
      // 몇 가지 기본값으로 스토어 채우기
      // 요청에 따라, const

```

```
store = {  
  userId: req.headers['X-user-id'],  
};  
// 그리고 "next" 함수를 콜백으로 전달합니다.  
//를 스토어와 함께 "als.run" 메서드에 추가합니다.
```

```

        this.als.run(store, () => next());
    })
    // 모든 경로에 등록합니다 (Fastify의 경우 '(.*)' 사용).
    .forRoutes('*');
}
}

```

. 이제 요청의 라이프사이클 내 어디서나 로컬 스토어 인스턴스에 액세스할 수 있습니다.

```

@@파일명(cat.service)
@Injectable()
내보내기 클래스 CatService { 생성
    자(
        // 제공된 ALS 인스턴스를 삽입할 수 있습니다:
        AsyncLocalStorage, 비공개 읽기 전용 catRepository:
        CatRepository,
    ) {}

    getCatForUser() {
        // "getStore" 메소드는 항상
        // 주어진 요청과 연관된 저장소 인스턴스입니다. const userId =
        this.als.getStore()["userId"] as number; return
        this.catRepository.getForUser(userId)를 반환합니다;
    }
}

@@스위치
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
    constructor(als, catRepository) {
        // 제공된 ALS 인스턴스를 삽입할 수 있습니다.
        this.catRepository = 고양이 저장소
    }

    getCatForUser() {
        // "getStore" 메소드는 항상
        // 주어진 요청과 연관된 저장소 인스턴스입니다. const userId =
        this.als.getStore()["userId"] as number; return
        this.catRepository.getForUser(userId)를 반환합니다;
    }
}

```

. 그게 다입니다. 이제 전체 요청을 주입할 필요 없이 요청 관련 상태를 공유할 수 있는 방법이 생겼습니다.

요청 객체입니다.

경고 경고 이 기술은 많은 사용 사례에 유용하지만 본질적으로 코드 흐름을 난독화하므로(암시적 컨텍스트 생성) 책임감 있게 사용해야 하며, 특히 컨텍스트에 맞는 '신 객체'를 만들지 않도록 주의하세요.

NestJS CLS

`nestjs-cls` 패키지는 일반 `AsyncLocalStorage`를 사용할 때보다 몇 가지 DX 개선 사항을 제공합니다(CLS는 *연속 로컬 스토리지*라는 용어의 약어입니다). 이 패키지는 구현을 강력한 타이핑 지원뿐만 아니라 다양한 전송(HTTP뿐만 아니라)에 대해 *저장소*를 초기화하는 다양한 방법을 제공하는 `ClsModule`로 추상화합니다.

그런 다음 삽입 가능한 `ClsService`를 사용하여 스토어에 액세스하거나 *프록시 공급자*를 사용하여 비즈니스 로직에서 완전히 추상화할 수 있습니다.

정보 `nestjs-cls`는 타사 패키지이며 NestJS 코어 팀에서 관리하지 않습니다. 라이브러리와 관련된 문제가 발견되면 [해당 리포지토리에](#) 보고해 주세요.

설치

`nestjs` 라이브러리에 대한 피어 종속성을 제외하고는 기본 제공 Node.js API만 사용합니다. 다른 패키지로 설치하세요.

NPM 및 NESTJS-CLS

사용법

[위에서](#) 설명한 것과 유사한 기능을 다음과 같이 `nestjs-cls`를 사용하여 구현할 수 있습니다:

```
. 루트 모듈에서 ClsModule을 가져옵니다.
```

```
@@파일명(app.module) @Module({
  수입: [
    // ClsModule 등록, ClsModule.forRoot({
    미들웨어: {
      // 자동으로 마운트
      // 모든 경로에 대한 ClsMiddleware 마운트:
      true,
      // 설정 방법을 사용하여
      // 기본 저장소 값을 제공합니다. 설정:
      (cls, req) => {
        cls.set('userId', req.headers['x-user-id']);
      },
    },
  ],
  제공자: [CatService], 컨트롤러:
  [CatController],
})

내보내기 클래스 AppModule {}
```

. 그런 다음 `ClsService`를 사용하여 저장소 값에 액세스할 수 있습니다.

```

@@파일명(cat.service)
@Injectable()
내보내기 클래스 CatService { 생성
    자(
        // 제공된 ClsService 인스턴스, 비공개 읽기 전용 cls를 삽입할
        수 있습니다: ClsService,
        비공개 읽기 전용 고양이 저장소: 고양이 저장소,
    ) {}

    getCatForUser() {
        // "get" 메서드를 사용하여 저장된 값을 검색합니다. const userId =
        this.cls.get('userId');
        this.catRepository.getForUser(userId)를 반환합니다;
    }
}

@@스위치
@Injectable()
@Dependencies(AsyncLocalStorage, CatRepository)
export class CatService {
    constructor(cls, catRepository) {
        // 제공된 ClsService 인스턴스, this.cls = cls를 삽입할 수
        있습니다.
        this.catRepository = 고양이 저장소
    }

    getCatForUser() {
        // "get" 메서드를 사용하여 저장된 값을 검색합니다. const userId =
        this.cls.get('userId');
        this.catRepository.getForUser(userId)를 반환합니다;
    }
}

```

. `ClsService`에서 관리하는 저장소 값을 강력하게 입력하려면(또한 문자열 키의 자동 제안을 받으려면), 선택적 유형 매개변수 `ClsService<MyClsStore>`를 삽입할 때 사용할 수 있습니다.

```

내보내기 인터페이스 MyClsStore는 ClsStore를 확장합니다
{
    userId: 숫자;
}

```

정보 힌트 패키지가 자동으로 요청 ID를 생성하도록 하고 나중에 `cls.getId()`를 사용하여 액세스하거나 `cls.get(CLS_REQ)`를 사용하여 전체 요청 객체를 가져올 수도 있습니다.

테스트

ClsService는 또 다른 인젝터블 프로바이더이므로 단위 테스트에서 완전히 모의 테스트할 수 있습니다.

그러나 특정 통합 테스트에서는 여전히 실제 `ClsService` 구현을 사용하고 싶을 수 있습니다. 이 경우 컨텍스트 인식 코드 조각을 `ClsService#run` 또는 `ClsService#runWith` 호출로 래핑해야 합니다.

```
describe('CatService', () => {
  let service: CatService
  let cls: ClsService
  const mockCatRepository = createMock<CatRepository>()

  beforeEach(async () => {
    const module = await Test.createTestingModule({
      // 대부분의 테스트 모듈을 평소와 같이 설정합니다: [
      CatService,
      {
        // 제공: CatRepository 사용값:
        mockCatRepository
      }
    ],
    // 수입: [
    // 다음만 제공하는 정적 버전의 ClsModule을 가져옵니다.
    //를 호출하지만 어떤 방식으로든 저장소를 설정하지 않습니다. ClsModule
  ]).compile()

  service = module.get(CatService)

  // 나중에 사용할 수 있도록 ClsService도 검색합니다. cls
  = module.get(ClsService)
})

describe('getCatForUser', () => {
  it('사용자 아이디를 기반으로 고양이를 검색합니다', async () => {
    const expectedUserId = 42
    mockCatRepository.getForUser.mockImplementationOnce(
      (id) => ({ userId: id })
    )

    // 테스트 호출을 `runWith` 메서드로 감싸기
    // 수작업으로 만든 저장소 값을 전달할 수 있습니다. const cat =
    await cls.runWith(
      { userId: expectedUserId },
      () => service.getCatForUser()
    )

    기대(cat.userId).toEqual(expectedUserId)
  })
})
```

```
} )  
})
```

자세한 정보

전체 API 문서와 더 많은 코드 예제를 보려면 [NestJS CLS GitHub 페이지](#)를 방문하세요.

오토목

Automock은 단위 테스트를 위한 독립형 라이브러리입니다. 내부적으로 TypeScript Reflection API([반영 메타데이터](#))를 사용하여 모의 객체를 생성하는 Automock은 클래스 외부 종속성을 자동으로 모의하여 테스트 개발을 간소화합니다.

정보 Automock은 타사 패키지이며 NestJS 코어 팀에서 관리하지 않습니다. 라이브러리와 관련된 문제가 발견되면 [해당 리포지토리에](#) 보고해 주세요.

소개

의존성 주입(DI) 컨테이너는 Nest 모듈 시스템의 필수 구성 요소입니다. 이 컨테이너는 테스트와 애플리케이션 실행 중에 모두 활용됩니다. 단위 테스트는 DI 컨테이너 내의 공급자/서비스를 완전히 재정의해야 한다는 점에서 통합 테스트와 같은 다른 유형의 테스트와 다릅니다. 소위 "단위"라고 하는 외부 클래스 종속성(공급자)은 완전히 격리되어야 합니다. 즉, DI 컨테이너 내의 모든 종속성을 모의 객체로 대체해야 합니다. 결과적으로 대상 모듈을 로드하고 그 안에 있는 프로바이더를 교체하는 것은 그 자체로 반복되는 프로세스입니다. Automock은 모든 클래스 외부 프로바이더를 자동으로 모킹하여 테스트 중인 유닛을 완전히 격리함으로써 이 문제를 해결합니다.

설치

```
$ npm i -D @automock/jest
```

오토목은 추가 설정이 필요하지 않습니다.

정보 정보 Jest는 현재 Automock에서 지원하는 유일한 테스트 프레임워크입니다. Sinon은 곧 출시될 예정입니다.

예

생성자 매개변수 세 개를 받는 다음 cats 서비스를 생각해 보세요:

```
@@파일명(cats.service)

'@nestjs/core'에서 { Injectable }을 임포트합니다;

@Injectable()
내보내기 클래스 CatsService { 생

  성자(
    개인 로거: 로거,
    개인 httpService: HttpService, 비공개
    catsDal: CatsDal,
  ) {}

  async getAllCats() {
    const cats = await
    this.httpService.get('http://localhost:3000/api/cats');
```

```

        this.logger.log('성공적으로 모든 고양이를 가져왔습니다');

        this.catsDal.saveCats(cats);
    }
}

```

이 서비스에는 다음 단위 테스트에 예제로 사용하는 메서드인 `getAllCats`라는 공용 메서드가 하나 포함되어 있습니다:

```

@@파일명(cats.service.spec)

'@automock/jest'에서 { TestBed }를 임포트하고,
'./cats.service'에서 { CatsService }를 임포트합니
다;

describe('CatsService 단위 사양', () => { let
    underTest: CatsService;
    logger: jest.Mocked<Logger>;
    let httpService: jest.Mocked<HttpService>;
    let catsDal: jest.Mocked<CatsDal>;

    beforeAll(() => {
        const { unit, unitRef } = TestBed.create(CatsService)
            .mock(HttpService)
            .using({ get: jest.fn() })
            .mock(Logger)
            .using({ log: jest.fn() })
            .mock(CatsDal)
            .using({ saveCats: jest.fn() })
            .compile();

        underTest = unit;

        logger = unitRef.get(Logger);
        httpService = unitRef.get(HttpService);
        catsDal = unitRef.get(CatsDal);
    });

    describe('모든 고양이를 구할 때', () => { test('then
        meet some expectations', async () => {
            httpService.get.mockResolvedValueOnce([
                { id: 1, name: 'Catty' }
            ]);
            await catsService.getAllCats();

            expect(logger.log).toBeCalled();
            expect(catsDal).toBeCalledWith([
                { id: 1, name: 'Catty' }
            ]);
        });
    });
});

```

정보 `jest.Mocked` 유틸리티 유형은 Jest 모의 함수의 유형 정의로 래핑된 소스 유형을 반환합니다. ([참조](#))

단위 및 unitRef 정보

다음 코드를 살펴보겠습니다:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();
```

`.compile()`을 호출하면 두 개의 프로퍼티, `unit`과 `unitRef`가 있는 객체가 반환됩니다.

단위는 테스트 중인 단위로, 테스트 중인 클래스의 실제 인스턴스입니다.

`unitRef`는 테스트된 클래스의 모의 종속성이 작은 컨테이너에 저장되는 "단위 참조"입니다. 컨테이너의 `.get()` 메서드는 모든 메서드가 자동으로 스텝된(`jest.fn()` 사용) 모킹된 의존성을 반환합니다:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();

let httpServiceMock: jest.Mocked<HttpService> = unitRef.get(HttpService);
```

정보 정보 `.get()` 메서드는 문자열 또는 실제 클래스(`Type`)를 인수로 받을 수 있습니다. 이는 기본적으로 프로바이더가 테스트 중인 클래스에 주입되는 방식에 따라 달라집니다.

다양한 제공업체와 협력

프로바이더는 Nest에서 가장 중요한 요소 중 하나입니다. 서비스, 저장소, 팩토리, 헬퍼 등 많은 기본 Nest 클래스를 프로바이더로 생각할 수 있습니다. 프로바이더의 주요 기능은 `Injectable` 의존성의 형태를 취하는 것입니다.

매개변수 하나를 취하는 다음 `CatsService`를 고려해 보겠습니다. 이 매개변수는 다음 `Logger`의 인스턴스입니다. 인터페이스:

```
export interface Logger {
  log(message: 문자열): void;
}

export class CatsService {
  constructor(private logger: Logger) {}
}
```


TypeScript의 리플렉션 API는 아직 인터페이스 리플렉션을 지원하지 않습니다. Nest는 문자열 기반 인젝션 토큰으로 이 문제를 해결합니다([사용자 정의 공급자](#) 참조):

```
export const MyLoggerProvider = {
  provide: 'MY_LOGGER_TOKEN',
  useValue: { ... },
}
```

```
내보내기 클래스 CatsService {  
    생성자(@Inject('MY_LOGGER_TOKEN') 비공개 읽기 전용 로거: Logger)  
    {}  
}
```

Automock은 이러한 관행을 따르며 `unitRef.get()` 메서드에서 실제 클래스를 제공하는 대신 문자열 기반 토큰을 제공할 수 있습니다:

```
const { unit, unitRef } = TestBed.create(CatsService).compile();  
  
let loggerMock: jest.Mocked<Logger> = unitRef.get('MY_LOGGER_TOKEN');
```

자세한 정보

자세한 내용은 [Automock GitHub 리포지토리](#)를 참조하세요.