구성

애플리케이션은 종종 서로 다른 환경에서 실행됩니다. 환경에 따라 다른 구성 설정을 사용해야 합니다. 예를 들어, 일반적으로 로컬 환경에서는 로컬 DB 인스턴스에만 유효한 특정 데이터베이스 자격 증명을 사용합니다. 프로덕션 환경에서는 별도의 DB 자격 증명 집합을 사용합니다. 구성 변수가 변경되므로 환경에 구성 변수를 저장하는 것이 가장 좋습니다.

외부에서 정의한 환경 변수는 `프로세스.env` 글로벌을 통해 Node.js 내부에서 볼 수 있습니다. 환경 변수를 각 환경마다 별도로 설정하여 여러 환경의 문제를 해결할 수 있습니다. 하지만 이는 특히 이러한 값을 쉽게 모킹하거나 변경해야 하는 개발 및 테스트 환경에서는 매우 번거로울 수 있습니다.

Node.js 애플리케이션에서는 각 환경을 나타내기 위해 각 키가 특정 값을 나타내는 키-값 쌍을 포함하는 `.env` 파일을 사용하는 것이 일반적입니다. 따라서 서로 다른 환경에서 앱을 실행하려면 올바른 `.env` 파일을 교체하기만 하면 됩니다.

Nest에서 이 기술을 사용하기 위한 좋은 접근 방식은 적절한 `.env` 파일을 로드하는 `ConfigService를` 노출하는 `ConfigModule을` 만드는 것입니다. 이러한 모듈을 직접 작성할 수도 있지만, 편의를 위해 Nest는 `@nestjs/config` 패키지를 기본으로 제공합니다. 이 패키지는 이번 장에서 다루겠습니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

```
$ npm i --save @nestjs/config
```

> 정보 힌트 `@nestjs/config` 패키지는 내부적으로 dotenv를 사용합니다.

> 경고 `@nestjs/config에는` TypeScript 4.1 이상이 필요합니다.

시작하기

설치 프로세스가 완료되면 `컨피그모듈을` 임포트할 수 있습니다. 일반적으로 루트 `AppModule로` 임포트하고 `.forRoot()` 정적 메서드를 사용하여 동작을 제어합니다. 이 단계에서는 환경 변수 키/값 쌍을 구문 분석하

고 해결합니다. 나중에 다른 기능 모듈에서 구성 모듈의 구성 서비스 클래스에 액세스하기 위한 몇 가지 옵션을 살펴볼 것입니다.

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져오고,
'@nestjs/config'에서 { ConfigModule }을 가져옵니
다;

모듈({
  임포트합니다: [ConfigModule.forRoot()],
})
내보내기 클래스 AppModule {}
```

위의 코드는 기본 위치(프로젝트 루트 디렉토리)에서 `.env` 파일을 로드 및 구문 분석하고, `.env` 파일의 키/값 쌍을 `process.env에` 할당된 환경 변수와 병합한 다음, 그 결과를 `ConfigService를` 통해 액세스할 수 있는 비공개 구조에 저장합니다. `forRoot()` 메서드는 구문 분석/병합된 구성 변수를 읽기 위한 `get()` 메서드를 제공하는 `ConfigService` 공급자를 등록합니다. nestjs/config는 dotenv에 의존하기 때문에 환경 변수 이름의 충돌을 해결하기 위해 해당 패키지의 규칙을 사용합니다. 키가 런타임 환경에 환경 변수로 존재할 때(예: `export DATABASE_USER=test와` 같은 OS 셸 내보내기를 통해) 및 `.env` 파일에 모두 존재할 때 런타임 환경 변수가 우선합니다.

샘플 `.env` 파일은 다음과 같습니다:

```
DATABASE_USER=test
DATABASE_PASSWORD=test
```

사용자 지정 환경 파일 경로

기본적으로 패키지는 애플리케이션의 루트 디렉터리에서 `.env` 파일을 찾습니다. `.env` 파일의 다른 경로를 지정하려면 다음과 같이 `forRoot()`에 전달하는 (선택적) 옵션 객체의 `envFilePath` 속성을 설정합니다:

```
ConfigModule.forRoot({
  envFilePath: '.development.env',
});
```

다음과 같이 `.env` 파일에 대해 여러 경로를 지정할 수도 있습니다:

```
ConfigModule.forRoot({
  envFilePath: ['.env.development.local', '.env.development'],
});
```

변수가 여러 파일에서 발견되면 첫 번째 파일이 우선합니다. 환경 변수 로

딩 비활성화

`.env` 파일을 로드하지 않고 대신 다음에서 환경 변수에 간단히 액세스하려는 경우
런타임 환경(예: `DATABASE_USER=test` 내보내기와 같은 OS 셸 내보내기와 마찬가지로)에서 다음과 같이 옵션

개체의 `무시EnvFile` 속성을 `true로` 설정합니다:

```
ConfigModule.forRoot({
  무시환경파일: true,
});
```

모듈을 전 세계적으로 사용

다른 모듈에서 `컨피그모듈을` 사용하려면 모든 Nest 모듈의 표준처럼 컨피그모듈을 임포트해야 합니다. 또는 아래와 같이 옵션 객체의 `isGlobal` 속성을 `true로` 설정하여 전역 모듈로 선언할 수도 있습니다. 이 경우 루트 모듈(예: `AppModule`)에 로드된 후에는 다른 모듈에서 ConfigModule을 임포트할 필요가 없습니다.

```
ConfigModule.forRoot({
  isGlobal: true,
});
```

사용자 지정 구성 파일

더 복잡한 프로젝트의 경우 사용자 정의 설정 파일을 활용하여 중첩된 설정 개체를 반환할 수 있습니다. 이를 통해 관련 구성 설정을 기능별로 그룹화하고(예: 데이터베이스 관련 설정), 관련 설정을 개별 파일에 저장하여 독립적으로 관리할 수 있습니다.

사용자 정의 구성 파일은 구성 객체를 반환하는 팩토리 함수를 내보냅니다. 구성 객체는 임의로 중첩된 일반 JavaScript 객체일 수 있습니다. `process.env` 객체에는 완전히 해결된 환경 변수 키/값 쌍이 포함됩니다(위에서 설명한 대로 `.env` 파일과 외부 정의 변수가 해결되고 병합됨). 반환된 구성 객체를 제어하므로 필요한 로직을 추가하여 값을 적절한 유형으로 캐스팅하고 기본값을 설정하는 등의 작업을 수행할 수 있습니다. 예를 들어

```
@@파일명(구성/설정) 내보내기 기본값 ()
=> ({
  port: parseInt(process.env.PORT, 10) || 3000,
  database: {
    호스트: process.env.DATABASE_HOST,
    port: parseInt(process.env.DATABASE_PORT, 10) || 5432
  }
});
```

`ConfigModule.forRoot()`에 전달하는 옵션 객체의 `load` 속성을 사용하여 이 파일을 로드합니다.

메서드를 사용합니다:

```
'./config/configuration'에서 구성 가져 오기; @Module({
  임포트합니다: [
    ConfigModule.forRoot({
      로드합니다: [구성],
    }),
  ],
})
내보내기 클래스 AppModule {}
```

> 정보 `로드` 프로퍼티에 할당된 값은 배열이므로 여러 구성 파일을 로드할 수 있습니다(예: `[load: [databaseConfig, authConfig]]`).

사용자 지정 구성 파일을 사용하면 YAML 파일과 같은 사용자 지정 파일도 관리할 수 있습니다. 다음은 YAML 형식을 사용한 구성의 예입니다:

```
http:
  호스트: 'localhost'
  포트: 8080

db:
  postgres:
    url: 'localhost' 포트: 5432 데이터베이스: 'yaml-db'

  sqlite:
    데이터베이스: 'sqlite.db'
```

YAML 파일을 읽고 구문 분석하기 위해 `js-yaml` 패키지를 활용할 수 있습니다.

```
$ npm i js-yaml
$ npm i -D @타입스/js-yaml
```

패키지가 설치되면 `yaml#load` 함수를 사용하여 위에서 방금 만든 YAML 파일을 로드합니다.

```
@@파일이름(config/configuration)
import { readFileSync } from 'fs';
import * as yaml from 'js-yaml';
import { join } from 'path';

const YAML_CONFIG_FILENAME = 'config.yaml';

export default () => {
  반환 yaml.load(
    readFileSync(join(__dirname, YAML_CONFIG_FILENAME), 'utf8'),
  );
};
```

> 경고 Nest CLI는 빌드 프로세스 중에 "자산"(TS가 아닌 파일)을 `dist` 폴더로 자동으로 이동하지 않습니다. YAML 파일이 복사되도록 하려면 `nest-cli.json` 파일의 `compilerOptions#assets` 객체에 서 이를 지정해야 합니다. 예를 들어 `config` 폴더가 `src` 폴더와 같은 수준인 경우 compilerOptions#assets에 다음 값을 추가합니다.
>
> `"assets": [{{ '{' }}"include": "../config/*.yaml", "outDir": "./dist/config"{{ '}' }}]`. 여기에서 자세히 알아보세요.

컨피그서비스 사용

컨피그 서비스에서 구성 값에 액세스하려면 먼저 컨피그 서비스를 주입해야 합니다. 다른 프로바이더와 마찬가지로, 해당 프로바이더를 포함하는 모듈인 ConfigModule을 이를 사용할 모듈로 임포트해야 합니다(ConfigModule.forRoot() 메서드에 전달된 옵션 객체의 isGlobal 속성을 true로 설정하지 않는 한). 아래와 같이 기능 모듈로 임포트합니다.

```
@@파일명(feature.module) @Module({
  임포트: [구성 모듈],
  // ...
})
```

그런 다음 표준 생성자 주입을 사용하여 주입할 수 있습니다:

```
constructor(private configService: ConfigService) {}
```

> 정보 힌트 컨피그서비스는 @nestjs/config 패키지에서 가져옵니다.

그리고 수업에서 사용하세요:

```
// 환경 변수 가져오기
const dbUser = this.configService.get<string>('DATABASE_USER');

// 사용자 지정 구성 값 가져오기
const dbHost = this.configService.get<string>('database.host');
```

위와 같이 configService.get() 메서드를 사용하여 변수 이름을 전달하여 간단한 환경 변수를 가져옵니다. 위와 같이 유형을 전달하여 TypeScript 유형 힌트를 수행할 수 있습니다(예: get<string>(...)). 위의 두 번째 예에서와 같이 get() 메서드는 중첩된 사용자 지정 구성 객체(사용자 지정 구성 파일을 통해 생성됨)를 순회할 수도 있습니다.

인터페이스를 유형 힌트로 사용하여 중첩된 전체 사용자 지정 구성 개체를 가져올 수도 있습니다:

```
인터페이스 DatabaseConfig { 호
  스트: 문자열;
  포트: 숫자;
}

const dbConfig = this.configService.get<DatabaseConfig>('database');

// 이제 `dbConfig.port`와 `dbConfig.host`를 사용할 수 있습
니다;
```

또한 `get()` 메서드는 아래와 같이 키가 존재하지 않을 때 반환되는 기본값을 정의하는 선택적 두 번째 인수를 받습니다:

```
// "database.host"가 정의되지 않은 경우 "localhost" 사용
const dbHost = this.configService.get<string>('database.host',
'localhost');
```

`ConfigService`는 두 개의 선택적 제네릭(유형 인수)이 있습니다. 첫 번째는 존재하지 않는 구성 속성에 액세스하는 것을 방지하는 데 도움이 됩니다. 아래 그림과 같이 사용합니다:

```
인터페이스 EnvironmentVariables {
  PORT: 숫자;
  TIMEOUT: 문자열입니다;
}

// 코드 어딘가에
constructor(private configService: ConfigService<EnvironmentVariables>) {
  const port = this.configService.get('PORT', { infer: true });

  // 타입스크립트 오류: URL 속성이 환경변수에 정의되어 있지 않으므로 유효하지 않습니다.
  const url = this.configService.get('URL', { infer: true });
}
```

`infer` 속성을 `true`로 설정하면 `ConfigService#get` 메서드가 인터페이스를 기반으로 속성 유형을 자동으로 추론하므로, 예를 들어 PORT는 `EnvironmentVariables` 인터페이스에서 `숫자` 유형을 가지므로 `typeof port === "숫자"`(TypeScript에서 `strictNullChecks` 플래그를 사용하지 않는 경우)가 됩니다.

또한 `유추` 기능을 사용하면 점 표기법을 사용하는 경우에도 다음과 같이 중첩된 사용자 지정 구성 개체의 속성 유형을 유추할 수 있습니다:

```
constructor(private configService: ConfigService<{ database: { host:
string } }>) {
  const dbHost = this.configService.get('database.host', { infer: true
})!;
  // dbHost 유형 === "문자열"                                          |
  //
+--> 널이 아닌 어설션 연산자
}
```

두 번째 제네릭은 첫 번째 제네릭에 의존하여 정의되지 않은 모든 유형을 제거하는 유형 어설션으로 작동합니다.

ConfigService의 메서드는 strictNullChecks가 켜져 있을 때 반환할 수 있습니다. 예를 들어

```
// ...
생성자(private configService: ConfigService<{ PORT: number }, true>)
{
  //                                                              ^^^^
  const port = this.configService.get('PORT', { infer: true });
        //^^^ 포트 유형이 '숫자'가 되므로 더 이상 TS 유형 어설션이 필요하지 않습니다.
}
```

구성 네임스페이스

위의 사용자 지정 구성 파일에 표시된 대로 여러 사용자 지정 구성 파일을 정의하고 로드할 수 있습니다. 해당 섹션에 표시된 것처럼 중첩된 구성 객체를 사용하여 복잡한 구성 객체 계층 구조를 관리할 수 있습니다. 또는 다음과 같이 `registerAs()` 함수를 사용하여 "네임스페이스" 구성 객체를 반환할 수 있습니다:

```
@@파일명(config/database.config)
export default registerAs('database', () => ({
  host: process.env.DATABASE_HOST,
  포트: process.env.DATABASE_PORT || 5432
}));
```

사용자 지정 설정 파일과 마찬가지로, `registerAs()` 팩토리 함수 내부의 `process.env` 객체에는 완전히 해결된 환경 변수 키/값 쌍이 포함됩니다(위에서 설명한 대로 `.env` 파일과 외부 정의 변수가 해결되고 병합된 상태).

> 정보 힌트 `registerAs` 함수는 `@nestjs/config` 패키지에서 내보냅니다.

사용자 지정 구성 파일을 로드하는 것과 같은 방식으로 `forRoot()` 메서드의 옵션 객체의 `load` 속성을 사용하여 네임스페이스 구성을 로드합니다:

```
'./config/database.config'에서 databaseConfig 가져오기;

@Module({
  임포트합니다: [
    ConfigModule.forRoot({
      로드합니다: [데이터베이스 구성],
    }),
  ],
})
내보내기 클래스 AppModule {}
```

이제 `데이터베이스` 네임스페이스에서 `호스트` 값을 가져오려면 점 표기법을 사용합니다. 네임스페이스 이름에 해당하는 속성 이름의 접두사로 `'database'`를 사용합니다(`registerAs()` 함수의 첫 번째 인수로 전달됨):

```
const dbHost = this.configService.get<string>('database.host');
```

합리적인 대안은 데이터베이스 네임스페이스를 직접 삽입하는 것입니다. 이렇게 하면 강력한 타이핑의 이점을 누릴 수 있습니다:

```
생성자( @Inject(databaseConfig.KEY)
  비공개 dbConfig: 구성 유형<데이터베이스 구성 유형>,
) {}
```

> 정보 힌트 구성 유형은 @nestjs/config 패키지에서 내보냅니다.

캐시 환경 변수

process.env에 액세스하는 속도가 느려질 수 있으므로, ConfigModule.forRoot()에 전달되는 옵션 객체의 캐시 속성을 설정하여 process.env에 저장된 변수에 대한 ConfigService#get 메서드의 성능을 향상시킬 수 있습니다.

```
ConfigModule.forRoot({ 캐시:
  true,
});
```

부분 등록

지금까지 루트 모듈(예: AppModule)에서 forRoot() 메서드를 사용하여 구성 파일을 처리했습니다. 기능별 구성 파일이 여러 다른 디렉터리에 있는 더 복잡한 프로젝트 구조를 가지고 있을 수도 있습니다. 이러한 모든 파일을 루트 모듈에 로드하는 대신 @nestjs/config 패키지는 각 기능 모듈과 관련된 구성 파일만 참조하는 부분 등록이라는 기능을 제공합니다. 이 부분 등록을 수행하려면 다음과 같이 기능 모듈 내에서 forFeature() 정적 메서드를 사용합니다:

```
'./config/database.config'에서 databaseConfig 가져오기;
@Module({
  임포트합니다: [ConfigModule.forFeature(databaseConfig)],
데이터베이스 모듈 클래스 {} 내보내기
```

> 정보 경고 일부 상황에서는 생성자가 아닌 onModuleInit() 훅을 사용하여 부분 등록을 통해 로드된 프로퍼티에 액세스해야 할 수도 있습니다. 이는 모듈 초기화 중에 forFeature() 메서드가 실행되고 모듈)초기화 순서가 불확실하기 때문입니다. 다른 모듈에서 이러한 방식으로 로드한 값에 접근하는 경우, 생성자에서 모듈의

> 구성이 의존하는 모듈이 아직 초기화되지 않았을 수 있습니다. `onModuleInit()` 메서드는 종속된 모든 모듈이 초기화된 후에만 실행되므로 이 기법은 안전합니다.

스키마 유효성 검사

필수 환경 변수가 제공되지 않았거나 특정 유효성 검사 규칙을 충족하지 않는 경우 애플리케이션을 시작하는 동안 예외를 발생시키는 것이 표준 관행입니다. `nestjs/config` 패키지를 사용하면 두 가지 방법으로 이를 수행할 수 있습니다:

- Joi 내장 유효성 검사기. Joi를 사용하면 객체 스키마를 정의하고 이에 대해 JavaScript 객체의 유효성을 검사할 수 있습니다. • 환경 변수를 입력으로 받는 사용자 정의 유효성 검사`()` 함수.

Joi를 사용하려면 Joi 패키지를 설치해야 합니다:

```
$ npm install --save joi
```

이제 Joi 유효성 검사 스키마를 정의하고 유효성 검사 스키마를 `forRoot()` 메서드의 옵션 객체를 아래와 같이 설정합니다:

```
@@파일명(app.module) 'joi'에서
*를 Joi로 가져옵니다;

모듈({ import: [
    ConfigModule.forRoot({
      validationSchema: Joi.object({
        NODE_ENV: Joi.string()
          .valid('개발', '프로덕션', '테스트', '프로비저닝')
          .default('development'),
        PORT: Joi.number().default(3000),
      }),
    }),
  ],
})
내보내기 클래스 AppModule {}
```

기본적으로 모든 스키마 키는 선택 사항으로 간주됩니다. 여기서는 환경(`.env` 파일 또는 프로세스 환경)에서 이러한 변수를 제공하지 않을 경우 사용되는 `NODE_ENV` 및 `PORT`에 대한 기본값을 설정합니다. 또는 `required()`

유효성 검사 메서드를 사용하여 환경(`.env` 파일 또는 프로세스 환경)에 값이 정의되어 있어야 함을 요구할 수 있습니다. 이 경우 유효성 검사 단계는 환경에 변수를 제공하지 않으면 예외를 발생시킵니다. 유효성 검사 스키마를 구성하는 방법에 대한 자세한 내용은 Joi 유효성 검사 메서드를 참조하세요.

기본적으로 알 수 없는 환경 변수(스키마에 키가 없는 환경 변수)는 허용되며 유효성 검사 예외를 트리거하지 않습니다. 기본적으로 모든 유효성 검사 오류가 보고됩니다. `forRoot()` 옵션 객체의 `validationOptions` 키를 통해 옵션 객체를 전달하여 이러한 동작을 변경할 수 있습니다. 이 옵션 객체에는 다음과 같은 표준 유효성 검사 옵션이 포함될 수 있습니다.

프로퍼티를 변경할 수 있습니다. 예를 들어 위의 두 설정을 반전시키려면 다음과 같은 옵션을 전달합니다:

```
@@파일명(app.module) 'joi'에서
*를 Joi로 가져옵니다;

모듈({ import: [
    ConfigModule.forRoot({
      validationSchema: Joi.object({
        NODE_ENV: Joi.string()
          .valid('개발', '프로덕션', '테스트', '프로비저닝')
          .default('development'),
        PORT: Joi.number().default(3000),
      }),
      유효성 검사 옵션: {
        allowUnknown: false,
        abortEarly: true,
      },
    }),
  ],
})
  내보내기 클래스 AppModule {}
```

nestjs/config 패키지는 다음과 같은 기본 설정을 사용합니다:

- allowUnknown: 환경 변수에 알 수 없는 키를 허용할지 여부를 제어합니다. 기본값은 true입니다.
- abortEarly: 참이면 첫 번째 오류에서 유효성 검사를 중지하고, 거짓이면 모든 오류를 반환합니다. 기본값은 false입니다.

유효성 검사 옵션 객체를 전달하기로 결정한 후에는 명시적으로 전달하지 않은 모든 설정은 @nestjs/config 기본값이 아닌 Joi 표준 기본값으로 기본값이 설정된다는 점에 유의하세요. 예를 들어, 사용자 지정 유효성 검사 옵션 객체에서 allowUnknowns를 지정하지 않은 상태로 두면 Joi 기본값이 false로 설정됩니다. 따라서 사용자 지정 객체에서 이 두 가지 설정을 모두 지정하는 것이 가장 안전합니다.

사용자 지정 유효성 검사 기능

또는 환경 변수가 포함된 객체(환경 파일 및 프로세스에서)를 가져와서 필요한 경우 변환/변환할 수 있도록 유효성이 검사된 환경 변수가 포함된 객체를 반환하는 동기식 유효성 검사 함수를 지정할 수 있습니다. 이 함수가 오류를 발생시키면 애플리케이션이 부트스트랩되지 않습니다.

이 예제에서는 클래스 트랜스포머와 클래스 유효성 검사기 패키지로 진행하겠습니다. 먼저 정의해야 합니다:

- 유효성 검사 제약 조건이 있는 클래스입니다,
- `plainToInstance` 및 `validateSync` 함수를 사용하는 유효성 검사 함수입니다.

```
```

```
@@파일명(환경 유효성 검사)
'class-transformer'에서 { plainToInstance }를 가져옵니다;
'class-validator'에서 { IsEnum, IsNumber, validateSync }를 가져옵니다;

열거형 환경 {
  개발 = "개발", 프로덕션 = "프로덕
  션", 테스트 = "테스트",
  프로비저닝 = "프로비저닝",
}

환경 변수 클래스 { @IsEnum(환경)
  NODE_ENV: 환경;

  IsNumber()
  PORT: 숫자;
}

export 함수 validate(config: Record<string, unknown>) { const
  validatedConfig = plainToInstance(
    환경 변수, config,
    { enableImplicitConversion: true },
  );
  const errors = validateSync(validatedConfig, { skipMissingProperties:
false });

  if (errors.length > 0) {
    새로운 오류(errors.toString())를 던집니다;
  }
  유효성 검사된 컨피그를 반환합니다;
}
```

이 설정이 완료되면 다음과 같이 유효성 검사 함수를 `ConfigModule의` 구성 옵션으로 사용합니다:

```
@@파일명(앱.모듈)
'./env.validation'에서 { validate }를 가져옵니다;

모듈({ import: [
    ConfigModule.forRoot({
      validate,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

사용자 지정 게터 함수

ConfigService는 키별로 구성 값을 검색하는 일반 get() 메서드를 정의합니다. 좀 더 자연스러운 코딩 스타일을 구현하기 위해 게터 함수를 추가할 수도 있습니다:

```
@@파일명()
@Injectable()
내보내기 클래스 ApiConfigService {
  constructor(private configService: ConfigService) {}

  get isAuthEnabled(): boolean {
    return this.configService.get('AUTH_ENABLED') === 'true';
  }
}
@@스위치 @종속성(컨피규레이션서비스) @인젝

터블()
export class ApiConfigService {
  constructor(configService) {
    this.config서비스 = config서비스;
  }

  get isAuthEnabled() {
    return this.configService.get('AUTH_ENABLED') === 'true';
  }
}
```

이제 다음과 같이 게터 함수를 사용할 수 있습니다:

```
@@파일명(app.service) @Injectable()
export class AppService {
  constructor(apiConfigService: ApiConfigService) {
    if (apiConfigService.isAuthEnabled) {
      // 인증이 활성화되었습니다.
    }
  }
}
@@스위치 @종속성(ApiConfigService)
@Injectable()
export class AppService {
  constructor(apiConfigService) {
    if (apiConfigService.isAuthEnabled) {
      // 인증이 활성화되었습니다.
    }
  }
}
```

환경 변수 로드 후크

모듈 구성이 환경 변수에 의존하고 이러한 변수는

`.env` 파일과 상호 작용하기 전에 파일이 로드되었는지 확인하기 위해 `ConfigModule.envVariablesLoaded`

훅을 사용할 수 있습니다(다음 예제 참조):

```
export async function getStorageModule() {
  await ConfigModule.envVariablesLoaded;
  반환 process.env.STORAGE === 'S3' ? S3StorageModule : 디폴트스토리지모듈;
}
```

이 구조는 `ConfigModule.envVariablesLoaded` 프로미스가 해결된 후 모든 구성 변수가 로드되도록 보장합니다.

확장 가능한 변수

`nestjs/config` 패키지는 환경 변수 확장을 지원합니다. 이 기술을 사용하면 한 변수가 다른 변수의 정의 내에서 참조되는 중첩 환경 변수를 만들 수 있습니다. 예를 들어

```
APP_URL=mywebsite.com 지원 이메일
=support@${APP_URL}
```

이 구성을 사용하면 변수 `SUPPORT_EMAIL이 'support@mywebsite.com'`로 해석됩니다. `'{' }}...{{ '}' }}` 구문을 사용하여 `SUPPORT_EMAIL` 정의 내에서 변수 APP_URL의 값 확인을 트리거합니다.

> 정보 힌트 이 기능의 경우 `@nestjs/config` 패키지는 내부적으로 dotenv-expand를 사용합니다.

아래 그림과 같이 `ConfigModule의 forRoot()` 메서드에 전달된 옵션 객체의 `expandVariables` 속성을 사용하여 환경 변수 확장을 활성화합니다:

```
@@파일명(app.module)
@Module({
  임포트합니다: [
    ConfigModule.forRoot({
      // ... 확장변수: true,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

`main.ts에서` 사용

구성은 서비스에 저장되어 있지만 `메인.ts` 파일에서도 사용할 수 있습니다. 이렇게 하면 애플리케이션 포트나 CORS 호스트와 같은 변수를 저장하는 데 사용할 수 있습니다.

액세스하려면 `app.get()` 메서드 뒤에 서비스 참조를 사용해야 합니다:

```
const configService = app.get(ConfigService);
```

그런 다음 구성 키로 `get` 메서드를 호출하여 평소처럼 사용할 수 있습니다:

```
const port = configService.get('PORT');
```

데이터베이스

Nest는 데이터베이스에 구애받지 않으므로 모든 SQL 또는 NoSQL 데이터베이스와 쉽게 통합할 수 있습니다. 선호도에 따라 다양한 옵션을 사용할 수 있습니다. 가장 일반적인 수준에서 Nest를 데이터베이스에 연결하려면 Express 또는 Fastify를 사용할 때와 마찬가지로 데이터베이스에 적합한 Node.js 드라이버를 로드하기만 하면 됩니다.

또한, 더 높은 수준의 추상화에서 작동하기 위해 MikroORM(MikroORM 레시피 참조), Sequelize(Sequelize 통합 참조), Knex.js(Knex.js 튜토리얼 참조), TypeORM, Prisma(Prisma 레시피 참조) 등 범용 Node.js 데이터베이스 통합 라이브러리 또는 ORM을 직접 사용할 수도 있습니다.

편의를 위해 Nest는 이번 챕터에서 다룰 `@nestjs/typeorm` 및 `@nestjs/sequelize` 패키지를 통해 TypeORM 및 Sequelize와 즉시 사용할 수 있는 긴밀한 통합 기능을 제공하며, 이 챕터에서 다룰 `@nestjs/mongoose` 패키지를 통해 Mongoose와도 통합할 수 있습니다. 이러한 통합은 모델/리포지토리 주입, 테스트 가능성, 비동기 구성과 같은 추가적인 NestJS 전용 기능을 제공하여 선택한 데이터베이스에 더욱 쉽게 액세스할 수 있도록 해줍니다.

TypeORM 통합

Nest는 SQL 및 NoSQL 데이터베이스와의 통합을 위해 `@nestjs/typeorm` 패키지를 제공합니다. TypeORM은 TypeScript에서 사용할 수 있는 가장 성숙한 객체 관계형 매퍼(ORM)입니다. TypeScript로 작성되었기 때문에 Nest 프레임워크와 잘 통합됩니다.

사용을 시작하려면 먼저 필요한 종속성을 설치합니다. 이 장에서는 널리 사용되는 MySQL 관계형 DBMS를 사용하는 데모를 보여드리지만, TypeORM은 PostgreSQL, Oracle, Microsoft SQL Server, SQLite, 심지어 MongoDB와 같은 NoSQL 데이터베이스와 같은 많은 관계형 데이터베이스를 지원합니다. 이 장에서 안내하는 절차는 TypeORM에서 지원하는 모든 데이터베이스에 대해 동일합니다. 선택한 데이터베이스에 대한 관련 클라이언트 API 라이브러리를 설치하기만 하면 됩니다.

```
npm install --save @nestjs/typeorm typeorm mysql2
```

설치 프로세스가 완료되면 `TypeOrmModule을` 루트 `앱모듈로` 가져올 수 있습니다.

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져옵니다;

모듈({ import: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      호스트: 'localhost',
      포트: 3306, 사용자 이
      름: 'root', 비밀번호:
      'root', 데이터베이스:
      'test', entities:
      [],
```

```
        동기화: true,
      }),
    ],
  })
  내보내기 클래스 AppModule {}
```

> 경고 동기화 설정: true는 프로덕션 환경에서 사용해서는 안 됩니다. 그렇지 않으면 프로덕션 데이터가 손실될 수 있습니다.

forRoot() 메서드는 TypeORM 패키지의 DataSource 생성자에 의해 노출되는 모든 구성 속성을 지원합니다. 또한 아래에 설명된 몇 가지 추가 구성 속성이 있습니다.

retryAttempts	데이터베이스에 연결하려는 시도 횟수(기본값: 10)
retryDelay	연결 재시도 시도 사이의 지연 시간(ms)(기본값: 3000)
autoLoadEntities 참이면	엔티티가 자동으로 로드됩니다(기본값: false).

> 정보 힌트 여기에서 데이터 원본 옵션에 대해 자세히 알아보세요.

이 작업이 완료되면 예를 들어 모듈을 임포트할 필요 없이 전체 프로젝트에 TypeORM DataSource 및 EntityManager 객체를 삽입할 수 있습니다:

```
@@파일명(앱.모듈)
'typeorm'에서 { DataSource }를 가져옵니다;

모듈({
  임포트: [TypeOrmModule.forRoot(), UsersModule],
})
내보내기 클래스 AppModule {
  constructor(private dataSource: DataSource) {}
}
@@switch
'typeorm'에서 { DataSource }를 가져옵니다;

@Dependencies(DataSource)
@Module({
  임포트: [TypeOrmModule.forRoot(), UsersModule],
})
export class AppModule {
  constructor(dataSource) {
    이 데이터 소스 = 데이터 소스;
  }
}
```

리포지토리 패턴

TypeORM은 리포지토리 디자인 패턴을 지원하므로 각 엔티티에는 자체 리포지토리가 있습니다. 이러한 리포지토리는 데이터베이스 데이터 소스에서 가져올 수 있습니다.

예제를 계속하려면 엔티티가 하나 이상 필요합니다. User 엔티티를 정의해 보겠습니다.

```
@@파일명(사용자.엔티티)
'typeorm'에서 { Entity, Column, PrimaryGeneratedColumn }을 가져옵니다;

엔티티()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  Column()
  firstName: 문자열;

  @Column() 성: 문자
  열;

  @Column({ 기본값: true })
  isActive: boolean;
}
```

> 정보 힌트 TypeORM 문서에서 엔티티에 대해 자세히 알아보세요.

사용자 엔티티 파일은 사용자 디렉터리에 있습니다. 이 디렉터리에는 UsersModule과 관련된 모든 파일이 들어 있습니다. 모델 파일을 어디에 보관할지 결정할 수 있지만 도메인 근처의 해당 모듈 디렉터리에 만드는 것이 좋습니다.

사용자 엔티티를 사용하려면 엔티티에 삽입하여 TypeORM에 해당 엔티티에 대해 알려야 합니다.
배열을 사용할 수 있습니다(정적 글로브 경로를 사용하지 않는 경우):

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,
'./users/user.entity'에서 { User }를 가져옵니다;

모듈({ import: [
    TypeOrmModule.forRoot({
      type: 'mysql',
      호스트: 'localhost',
      포트: 3306, 사용자 이
      름: 'root', 비밀번호:
      'root', 데이터베이스:
      'test', entities:
      [사용자], 동기화:
      true,
    }),
  ],
})
내보내기 클래스 AppModule {}
```

다음으로 UsersModule을 살펴보겠습니다:

```
@@파일명(users.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule } 가져오기,
'./users.service'에서 { UsersService } 가져오기,
'./users.controller'에서 { UsersController } 가져오기,
'./user.entity'에서 { User } 가져오기;

모듈({
  임포트합니다: [TypeOrmModule.forFeature([User])],
  providers: [UsersService],
  컨트롤러: [UsersController],
})
사용자 모듈 클래스 {} 내보내기
```

이 모듈은 forFeature() 메서드를 사용하여 현재 스코프에 등록된 리포지토리를 정의합니다. 이렇게 정의했으면 @InjectRepository() 데코레이터를 사용하여 UsersRepository를 UsersService에 삽입할 수 있습니다:

```
@@파일명(users.service)
'@nestjs/common'에서 { Injectable } 임포트;
'@nestjs/typeorm'에서 { InjectRepository } 임포트;
'typeorm'에서 { Repository } 임포트;
'./user.entity'에서 { User }를 가져옵니다;

@Injectable()
내보내기 클래스 UsersService { 생성자(
    @InjectRepository(사용자)
    비공개 사용자리포지토리: 리포지토리<사용자>,
  ) {}

  findAll(): Promise<User[]> {
    this.usersRepository.find()를 반환합니다;
  }

  findOne(id: 숫자): Promise<사용자 | null> { return
    this.usersRepository.findOneBy({ id });
  }

  async remove(id: 숫자): Promise<void> { await
    this.usersRepository.delete(id);
  }
}
@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 가져오고,
'@nestjs/typeorm'에서 { getRepositoryToken }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
```

```
  주입 가능()
  @Dependencies(getRepositoryToken(User))
  export class UsersService {
    constructor(usersRepository) {
      this.usersRepository = usersRepository;
    }

    findAll() {
      this.usersRepository.find()를 반환합니다;
    }

    findOne(id) {
      이.usersRepository.findOneBy({ id })를 반환합니다;
    }

    async remove(id) {
      await this.usersRepository.delete(id);
    }
  }
```

> 경고 UsersModule을 루트 앱모듈로 임포트하는 것을 잊지 마세요.

TypeOrmModule.forFeature를 가져오는 모듈 외부에서 리포지토리를 사용하려면 해당 모듈에서 생성된 프로바이더를 다시 내보내야 합니다. 다음과 같이 전체 모듈을 내보내면 됩니다:

```
  @@파일명(users.module)
  '@nestjs/common'에서 { Module }을 가져옵니다;
  '@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,
  './user.entity'에서 { User }를 가져옵니다;

  모듈({
    임포트합니다:
    [TypeOrmModule.forFeature([User])], 내보내기:
    [TypeOrmModule]
  })
  사용자 모듈 클래스 {} 내보내기
```

이제 UserHttpModule에서 UsersModule을 가져오면 후자의 모듈의 공급자에 @InjectRepository(User)를 사용할 수 있습니다.

```
@@파일명(users-http.module)
'@nestjs/common'에서 { Module } 가져오기;
'./users.module'에서 { UsersModule } 가져오기;
'./users.service'에서 { UsersService } 가져오기;
'./users.controller'에서 { UsersController }를 가져옵니다;

모듈({
  임포트: [UsersModule], 공급자:
  [UsersService],
```

```
    컨트롤러: [UsersController]
  })
  내보내기 클래스 UserHttpModule {}
```

관계

관계는 둘 이상의 테이블 간에 설정된 연결입니다. 관계는 각 테이블의 공통 필드를 기반으로 하며, 종종 기본 키와 외래 키가 포함됩니다.

관계에는 세 가지 유형이 있습니다:

일대일	주 테이블의 모든 행에는 외래 테이블에 연결된 행이 하나만 있습니다. 이 유형의 관계를 정의하려면 `@OneToOne()` 데코레이터를 사용합니다.
일대다 / 다대일	주 테이블의 모든 행에는 외래 테이블에 하나 이상의 관련 행이 있습니다. 주 테이블에서 이 유형의 관계를 정의하는 데 `@OneToMany()` 및 `@ManyToOne()` 데코레이터를 사용할 수 있습니다.
다대다	주 테이블의 모든 행은 외래 테이블에 많은 관련 행이 있고, 외래 테이블의 모든 레코드는 주 테이블에 많은 관련 행이 있습니다. 이러한 유형의 관계를 정의하려면 `@ManyToMany()` 데코레이터를 사용합니다.

엔티티에서 관계를 정의하려면 해당 데코레이터를 사용합니다. 예를 들어 각 `사용자`는 여러 장의 사진을 가질 수 있으므로 `@OneToMany()` 데코레이터를 사용합니다.

```
@@파일명(사용자.엔티티)
'typeorm'에서 { Entity, Column, PrimaryGeneratedColumn, OneToMany }를 가져옵
니다;
'../사진/사진.엔티티'에서 { 사진 }을 가져옵니다;

엔티티()
export class User {
  @PrimaryGeneratedColumn()
  id: number;

  Column()
  firstName: 문자열;

  @Column() 성: 문자
  열;

  @Column({ 기본값: true })
  isActive: boolean;

  원투다수(유형 => 사진, 사진 => 사진.사용자) 사진:
  Photo[];
}
```

> 정보 힌트 TypeORM의 관계에 대해 자세히 알아보려면 TypeORM 문서를 참조하세요.

엔티티 자동 로드

데이터 소스 옵션의 `엔티티` 배열에 엔티티를 수동으로 추가하는 작업은 번거로울 수 있습니다. 또한 루트 모듈에서 엔티티를 참조하면 애플리케이션 도메인 경계가 무너지고 애플리케이션의 다른 부분으로 구현 세부 정보가 유출될 수 있습니다. 이 문제를 해결하기 위해 대체 솔루션이 제공됩니다. 엔티티를 자동으로 로드하려면 아래 그림과 같이 구성 객체(`forRoot()` 메서드에 전달됨)의 `autoLoadEntities` 속성을 `true로` 설정합니다:

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져옵니다;

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

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

> 경고 `forFeature()` 메서드를 통해 등록되지 않고 (관계를 통해서만) 엔티티에서 참조되는 엔티티는 `자동로드 엔티티` 설정을 통해 포함되지 않는다는 점에 유의하세요.

엔티티 정의 분리

데코레이터를 사용하여 모델에서 바로 엔티티와 해당 열을 정의할 수 있습니다. 그러나 일부 사람들은 '엔티티 스키마'를 사용하여 별도의 파일 내에 엔티티와 해당 열을 정의하는 것을 선호합니다.

```
'typeorm'에서 { EntitySchema } 가져오기;

'./user.entity'에서 { User } 가져오기;

export const UserSchema = new EntitySchema<User>({
  name: 'User,
  대상: 사용자, 열: {
    id: {
      유형입니다: 숫자, 기본:
      true, 생성됨: true,
    },
    firstName: {
```

```
        유형: 문자열,
      },
      성: { type: 문자
        열,
      },
      isActive: {
        type: 부울, 기
        본값: true,
      },
    },
    관계: { photos: {
        유형: '일대다',
        대상: '사진', // 사진 스키마의 이름
      },
    },
  });
```

> 경고 오류 경고 대상 옵션을 제공하는 경우 이름 옵션 값은 대상 클래스의 이름과 동일해야 합니다. 대상을 제공하지 않으면 아무 이름이나 사용할 수 있습니다.

예를 들어 엔티티가 필요한 곳이면 어디에서나 Nest를 사용하여 엔티티 스키마 인스턴스를 사용할 수 있습니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,
'./user.schema'에서 { UserSchema }를 가져옵니다;
'./users.controller'에서 { UsersController }를 가져오고,
'./users.service'에서 { UsersService }를 가져옵니다;

모듈({
  임포트합니다: [TypeOrmModule.forFeature([UserSchema])],
  제공자: [UsersService],
  컨트롤러: [UsersController],
})
사용자 모듈 클래스 {} 내보내기
```

유형ORM 트랜잭션

데이터베이스 트랜잭션은 데이터베이스 관리 시스템 내에서 데이터베이스에 대해 수행되는 작업 단위를 의미하며, 다른 트랜잭션과 독립적으로 일관성 있고 신뢰할 수 있는 방식으로 처리됩니다. 트랜잭션은 일반적으로 데이터베

이스의 모든 변경 사항을 나타냅니다(자세히 알아보기).

TypeORM 트랜잭션을 처리하는 데는 여러 가지 전략이 있습니다. 저희는
트랜잭션을 완전히 제어할 수 있기 때문입니다.

먼저, 일반적인 방법으로 데이터 소스 객체를 클래스에 주입해야 합니다:

```
@Injectable()
사용자 서비스 클래스 내보내기 {
```

```
  constructor(private dataSource: DataSource) {}
}
```

> 정보 힌트 `DataSource` 클래스는 `typeorm` 패키지에서 가져옵니다.

이제 이 객체를 사용하여 트랜잭션을 생성할 수 있습니다.

```
async createMany(users: User[]) {
  const queryRunner = this.dataSource.createQueryRunner();

  await queryRunner.connect();
  await queryRunner.startTransaction();
  try { {
    await queryRunner.manager.save(users[0]);
    await queryRunner.manager.save(users[1]);

    await queryRunner.commitTransaction();
  } catch (err) {
    // 오류가 발생했으므로 변경 사항을 롤백하겠습니다 await
    queryRunner.rollbackTransaction();
  } finally {
    // 수동으로 인스턴스화된 쿼리 러너를 릴리스해야 합니다 await
    queryRunner.release();
  }
}
```

> 정보 힌트 `데이터소스는 쿼리러너를` 생성하는 데에만 사용된다는 점에 유의하세요. 그러나 이 클래스를 테스트하려면 여러 메서드를 노출하는 전체 `DataSource` 객체를 모킹해야 합니다. 따라서 헬퍼 팩토리 클래스(예: `QueryRunnerFactory`)를 사용하고 트랜잭션을 유지하는 데 필요한 제한된 메서드 집합으로 인터페이스를 정의하는 것이 좋습니다. 이 기법을 사용하면 이러한 메서드를 매우 간단하게 모킹할 수 있습니다.

또는 `데이터 소스의 트랜잭션` 메서드와 함께 콜백 스타일 접근 방식을 사용할 수 있습니다.

객체 (자세히 읽기).

```
async createMany(users: User[]) {
  await this.dataSource.transaction(async manager => {
    await manager.save(users[0]);
    await manager.save(users[1]);
  });
}
```

구독자

TypeORM 구독자를 사용하면 특정 엔티티 이벤트를 수신할 수 있습니다.

```
가져 오기 {
  DataSource,
  엔티티 구독자 인터페이스, 이벤트
  구독자, 삽입 이벤트,
}를 'typeorm'에서 가져옵니다;
'./user.entity'에서 { User }를 가져옵니다;

이벤트 구독자()
export class UserSubscriber 구현 EntitySubscriberInterface<User> {
  constructor(dataSource: DataSource) {
    dataSource.subscribers.push(this);
  }

  listenTo() {
    return User;
  }

  beforeInsert(event: InsertEvent<사용자>) { console.log(`
    사용자가 삽입되기 전: `, event.entity);
  }
}
```

> 오류 경고 이벤트 구독자를 요청 범위로 지정할 수 없습니다.

이제 `공급자` 배열에 `UserSubscriber` 클래스를 추가합니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/typeorm'에서 { TypeOrmModule }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;
'./users.controller'에서 { UsersController } 가져오기,
'./users.service'에서 { UsersService } 가져오기,
'./user.subscriber'에서 { UserSubscriber } 가져오기;

모듈({
  임포트합니다: [TypeOrmModule.forFeature([User])], 제공자
  : [UsersService, UserSubscriber], 컨트롤러:
  [UsersController],
})
사용자 모듈 클래스 {} 내보내기
```

> 정보 헌트 법인 구독자에 대한 자세한 내용은 여기를 참조하세요.

마이그레이션

마이그레이션은 데이터베이스의 기존 데이터를 보존하면서 데이터베이스 스키마를 점진적으로 업데이트하여 애플리케이션의 데이터 모델과 동기화 상태를 유지할 수 있는 방법을 제공합니다. 마이그레이션을 생성, 실행 및 되돌리기 위해 TypeORM은 전용 CLI를 제공합니다.

마이그레이션 클래스는 Nest 애플리케이션 소스 코드와는 별개입니다. 마이그레이션 클래스의 라이프사이클은 TypeORM CLI에 의해 유지 관리됩니다. 따라서 마이그레이션을 통해 종속성 주입 및 기타 Nest 관련 기능을 활용할 수 없습니다. 마이그레이션에 대해 자세히 알아보려면 TypeORM 설명서의 가이드를 따르세요.

여러 데이터베이스

일부 프로젝트에는 여러 데이터베이스 연결이 필요합니다. 이 모듈을 사용하면 이 작업도 수행할 수 있습니다. 여러 연결로 작업하려면 먼저 연결을 만듭니다. 이 경우 데이터 소스 이름 지정이 필수가 됩니다.

자체 데이터베이스에 저장된 `앨범` 엔티티가 있다고 가정해 보겠습니다.

```
const defaultOptions = {
  type: 'postgres',
  port: 5432,
  사용자 이름: '사용자', 비밀번호
  : '비밀번호', 데이터베이스:
  'db', synchronize: true,
};

모듈({ import: [
    TypeOrmModule.forRoot({
      ...defaultOptions,
      host: 'user_db_host',
      entities: [사용자],
    }),
    TypeOrmModule.forRoot({
      ...기본옵션,
      이름: 'albumsConnection', 호
      스트: 'album_db_host',
      entities: [앨범],
    }),
  ],
})
세분비기 클래스 AppModule {}
```

경고 데이터 소스의 `이름을` 설정하지 않으면 해당 이름이 `기본값으로` 설정됩니다. 이름이 없거나 이름이 같은 연결이 여러 개 있으면 재정의되므로 주의하세요.

경고 `TypeOrmModule.forRootAsync를` 사용하는 경우, `사용 팩토리` 외부에 데이터 소스 이름도 설정해야 합니다. 예를 들어

```
TypeOrmModule.forRootAsync({
  name: 'albumsConnection',
  useFactory:...,
```

```
    주입: ...,
  }),
```

자세한 내용은 이번 호를 참조하세요.

이 시점에서 사용자 및 앨범 엔티티가 자체 데이터 소스에 등록되어 있습니다. 이 설정을 사용하면 TypeOrmModule.forFeature() 메서드와 @InjectRepository() 데코레이터에 어떤 데이터 소스를 사용해야 하는지 알려주어야 합니다. 데이터 소스 이름을 전달하지 않으면 기본 데이터 소스가 사용됩니다.

```
모듈({ import: [
    TypeOrmModule.forFeature([User]),
    TypeOrmModule.forFeature([Album], 'albumsConnection'),
  ],
})
내보내기 클래스 AppModule {}
```

지정된 데이터 소스에 대한 DataSource 또는 EntityManager를 삽입할 수도 있습니다:

```
@Injectable()
내보내기 클래스 앨범 서비스 { 생성자(
    @InjectDataSource('albumsConnection') 비공개
    데이터 소스: DataSource,
    @InjectEntityManager('albumsConnection')
    private entityManager: EntityManager,
  ) {}
}
```

데이터소스를 공급자에게 주입할 수도 있습니다:

```
모듈({ providers:
  [
    {
      제공 앨범 서비스,
      useFactory: (albumsConnection: DataSource) => {
        반환 새 앨범 서비스(albumsConnection);
      },
      주입합니다: [getDataSourceToken('albumsConnection')],
    },
  ],
})
내보내기 클래스 앨범 모듈 {}
```

테스트

애플리케이션을 단위 테스트할 때 일반적으로 데이터베이스 연결을 피하고 테스트 스위트를 독립적으로 유지하며 실행 프로세스를 가능한 한 빠르게 유지하려고 합니다. 하지만 클래스가 데이터 소스(연결) 인스턴스에서 가져온 리포지토리에 의존할 수 있습니다. 이를 어떻게 처리할까요? 해결책은 모의 리포지토리를 만드는 것입니다. 이를 위해 사용자 지정 공급자를 설정합니다. 등록된 각 리포지토리는 자동으로 `<EntityName> 리포지토리` 토큰으로 표시되며, 여기서 `EntityName은` 엔티티 클래스의 이름입니다.

`nestjs/typeorm` 패키지는 주어진 엔티티를 기반으로 준비된 토큰을 반환하는 `getRepositoryToken()` 함수를 노출합니다.

```
모듈({ providers:
  [
    사용자 서비스,
    {
      제공: getRepositoryToken(User), 사용값: 모
      의 리포지토리,
    },
  ],
})
사용자 모듈 클래스 {} 내보내기
```

이제 대체 mockRepository가 `UsersRepository로` 사용됩니다. 어떤 클래스에서 `@InjectRepository()` 데코레이터를 사용하여 `UsersRepository를` 요청할 때마다 Nest는 등록된 `mockRepository` 객체를 사용합니다.

비동기 구성

리포지토리 모듈 옵션을 정적이 아닌 비동기적으로 전달하고 싶을 수도 있습니다. 이 경우 비동기 구성을 처리하는 여러 가지 방법을 제공하는 `forRootAsync()` 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
TypeOrmModule.forRootAsync({
  useFactory: () => ({
    유형: 'mysql', 호스
    트: 'localhost',
    포트: 3306, 사용자명
    : 'root', 비밀번호:
    'root', 데이터베이스
    : 'test',
    entities: [],
    synchronize: true,
  }),
});
```

저희 팩토리는 다른 비동기 공급자처럼 동작합니다(예: 비동기일 수 있고, 주입을 통해 종속성을 주입할 수 있습니다).

```
TypeOrmModule.forRootAsync({
  import: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    type: 'mysql',
    host:  configService.get('HOST'),
    port: +configService.get('PORT'),
    사용자                                  이름:
    configService.get('USERNAME'),   비밀번호:
    configService.get('PASSWORD'),   데이터베이
    스:        configService.get('DATABASE'),
    entities: [],
    동기화: true,
  }),
    주입합니다: [구성 서비스],
});
```

또는 useClass 구문을 사용할 수도 있습니다:

```
TypeOrmModule.forRootAsync({
  useClass: TypeOrmConfigService,
});
```

위의 구조는 TypeOrmModule 내부에 TypeOrmConfigService를 인스턴스화하고 이를 사용하여 createTypeOrmOptions()를 호출하여 옵션 객체를 제공합니다. 이는 아래 그림과 같이 TypeOrmConfigService가 TypeOrmOptionsFactory 인터페이스를 구현해야 한다는 것을 의미합니다:

```
@Injectable()
export 클래스 TypeOrmConfigService 구현 TypeOrmOptionsFactory {
  createTypeOrmOptions(): TypeOrmModuleOptions {
    반환 {
      유형: 'mysql', 호스트
      : 'localhost', 포트
      : 3306, 사용자명:
      'root', 비밀번호:
      'root', 데이터베이스:
      'test', entities:
      [], synchronize:
      true,
    };
  }
}
```

`TypeOrmModule` 내부에 `TypeOrmConfigService`를 생성하지 않고 다른 모듈에서 가져온 프로바이더를 사용하려면 `useExisting` 구문을 사용할 수 있습니다.

```
TypeOrmModule.forRootAsync({
  import: [ConfigModule],
  useExisting: ConfigService,
});
```

이 구조는 사용클래스와 동일하게 작동하지만 한 가지 중요한 차이점이 있습니다. TypeOrmModule은 가져온 모듈을 조회하여 새 구성 서비스를 인스턴스화하는 대신 기존 구성 서비스를 재사용합니다.

> 정보 힌트 이름 프로퍼티가 사용팩토리, 사용클래스 또는 사용값 프로퍼티와 동일한 수준에서 정의되었는지 확인하세요. 이렇게 하면 Nest가 적절한 주입 토큰 아래에 데이터 소스를 올바르게 등록할 수 있습니다.

맞춤형 데이터 소스 팩토리

사용Factory, 사용Class 또는 사용Existing을 사용하는 비동기 구성과 함께, 선택적으로 데이터소스팩토리 함수를 지정하여 TypeOrmModule이 데이터 소스를 생성하도록 허용하는 대신 자체 TypeORM 데이터 소스를 제공할 수 있도록 할 수 있습니다.

dataSourceFactory는 useFactory, useClass 또는 useExisting을 사용하여 비동기 구성 중에 구성된 TypeORM 데이터 소스 옵션을 수신하고 TypeORM 데이터 소스를 확인하는 Promise를 반환합니다.

```
TypeOrmModule.forRootAsync({
  import: [ConfigModule],
  inject: [ConfigService],
  // 사용Factory, 사용Class 또는 사용Existing 사용
  //를 사용하여 데이터 소스 옵션을 구성합니다. 사용 팩토리:
  (config서비스: 구성 서비스) => ({
    유형: 'mysql',
    host: configService.get('HOST'),
    port: +configService.get('PORT'),
    사용자 이름:
    configService.get('USERNAME'), 비밀번호:
    configService.get('PASSWORD'), 데이터베이
    스: configService.get('DATABASE'),
    entities: [],
    동기화: true,
  }),
  // 데이터소스는 구성된 데이터소스옵션을 수신합니다.
  //를 호출하고 약속<데이터소스>를 반환합니다.
  dataSourceFactory: async (옵션) => {
    const dataSource = await new DataSource(options).initialize();
    return dataSource;
  },
});
```

정보 힌트 DataSource 클래스는 typeorm 패키지에서 가져옵니다.

예

작동하는 예제는 여기에서 확인할 수 있습니다.

시퀄라이즈 통합

TypeORM을 사용하는 대신 `@nestjs/sequelize` 패키지와 함께 Sequelize ORM을 사용하는 방법도 있습니다. 또한 엔티티를 선언적으로 정의하기 위해 추가 데코레이터 세트를 제공하는 sequelize-typescript 패키지를 활용합니다.

사용을 시작하려면 먼저 필요한 종속성을 설치합니다. 이 장에서는 널리 사용되는 MySQL 관계형 DBMS를 사용하는 데모를 보여드리지만, Sequelize는 PostgreSQL, MySQL, Microsoft SQL Server, SQLite 및 MariaDB와 같은 많은 관계형 데이터베이스를 지원합니다. 이 장에서 안내하는 절차는 Sequelize에서 지원하는 모든 데이터베이스에 대해 동일합니다. 선택한 데이터베이스에 대한 관련 클라이언트 API 라이브러리를 설치하기만 하면 됩니다.

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

설치 프로세스가 완료되면 `SequelizeModule을` 루트 `앱 모듈로` 가져올 수 있습니다.

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/sequelize'에서 { SequelizeModule }을 가져옵니다;

모듈({ import: [
    SequelizeModule.forRoot({
      dialect: 'mysql',
      호스트: 'localhost',
      포트: 3306, 사용자 이
      름: 'root', 비밀번호:
      'root', 데이터베이스:
      'test', models:
      [],
    }),
  ],
})
내보내기 클래스 AppModule {}
```

`forRoot()` 메서드는 Sequelize 생성자에 의해 노출되는 모든 구성 속성을 지원합니다(자세히 보기). 또한 아래에 설명된 몇 가지 추가 구성 프로퍼티가 있습니다.

`retryAttempts`	데이터베이스 연결 시도 횟수(기본값: `10`) `retryDelay` 연결 재시도 시도 사이의 지연 시간(ms)(기본값: `3000`) `autoLoadModels` 참이면 모델이 자동으로 로드됩니다(기본값: `거짓`).
`keepConnectionAlive`	참이면 애플리케이션 종료 시 연결이 닫히지 않습니다(기본값):

	거짓)
동기화	true인 경우 자동으로 로드된 모델이 동기화됩니다(기본값: true).

이 작업이 완료되면 예를 들어 모듈을 임포트할 필요 없이 전체 프로젝트에 Sequelize 오브젝트를 삽입할 수 있습니다:

```
@@파일명(앱.서비스)
'@nestjs/common'에서 { Injectable }을 임포트하고,
'sequelize-typescript'에서 { Sequelize }를 임포트합
니다;

@Injectable()
내보내기 클래스 AppService {
  constructor(private sequelize: Sequelize) {}
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트하고,
'sequelize-typescript'에서 { Sequelize }를 임포트합
니다;

종속성(시퀄라이즈) @인젝터블()
export class AppService {
  constructor(sequelize) {
    this.sequelize = 시퀄라이즈;
  }
}
```

모델

Sequelize는 활성 레코드 패턴을 구현합니다. 이 패턴을 사용하면 모델 클래스를 직접 사용하여 데이터베이스와 상호 작용합니다. 예제를 계속하려면 적어도 하나의 모델이 필요합니다. 사용자 모델을 정의해 보겠습니다.

```
@@파일명(사용자.모델)
'sequelize-typescript'에서 { Column, Model, Table }을 가져옵니다;

테이블
내보내기 클래스 User extends Model {
  @Column
  이름: 문자열입니다;

  칼럼
  성: 문자열입니다;

  @Column({ defaultValue: true })
  isActive: boolean;
}
```

> 정보 힌트 여기에서 사용 가능한 데코레이터에 대해 자세히 알아보세요.

사용자 모델 파일은 사용자 디렉터리에 있습니다. 이 디렉터리에는 UsersModule과 관련된 모든 파일이 들어 있습니다. 모델 파일을 어디에 보관할지 결정할 수 있지만 도메인 근처의 해당 모듈 디렉터리에 만드는 것이 좋습니다.

User 모델을 사용하려면 모듈의 forRoot() 메서드 옵션에 있는 모델 배열에 삽입하여 Sequelize에 해당 모델을 알려야 합니다:

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/sequelize'에서 { SequelizeModule }을 임포트하고,
'./users/user.model'에서 { User }를 임포트합니다;

모듈({ import: [
    SequelizeModule.forRoot({
      dialect: 'mysql',
      호스트: 'localhost',
      포트: 3306, 사용자 이
      름: 'root', 비밀번호:
      'root', 데이터베이스:
      'test', models: [사
      용자],
    }),
  ],
})
내보내기 클래스 AppModule {}
```

다음으로 UsersModule을 살펴보겠습니다:

```
@@파일명(users.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/sequelize'에서 { SequelizeModule }을 임포트하고,
'./user.model'에서 { User }를 임포트합니다;
'./users.controller'에서 { UsersController }를 가져오고,
'./users.service'에서 { UsersService }를 가져옵니다;

모듈({
  임포트합니다: [SequelizeModule.forFeature([User])],
  providers: [UsersService],
  컨트롤러: [UsersController],
})
사용자 모듈 클래스 {} 내보내기
```

이 모듈은 `forFeature()` 메서드를 사용하여 현재 스코프에 등록된 모델을 정의합니다. 이를 통해 `@InjectModel()` 데코레이터를 사용하여 `사용자 모델을 사용자서비스에` 삽입할 수 있습니다:

```
@@파일명(users.service)
'@nestjs/common'에서 { Injectable }을 임포트하고,
'@nestjs/sequelize'에서 { InjectModel }을 임포트하고,
'./user.model'에서 { User }를 임포트합니다;

@Injectable()
내보내기 클래스 UsersService { 생성자(
    @InjectModel(사용자)
    비공개 사용자 모델: 사용자 유형,
  ) {}

  비동기 findAll(): Promise<User[]> {
    return this.userModel.findAll();
  }

  findOne(id: 문자열): Promise<User> {
    return this.userModel.findOne({
      where: {
        id,
      },
    });
  }

  async remove(id: 문자열): Promise<void> {
    const user = await this.findOne(id);
    await user.destroy();
  }
}
@@switch
'@nestjs/common'에서 { Injectable, Dependencies }를 임포트하고,
'@nestjs/sequelize'에서 { getModelToken }을 임포트합니다;
'./user.model'에서 { User }를 가져옵니다;

주입 가능()
@Dependencies(getModelToken(User)) 내보
내기 클래스 UsersService {
  constructor(usersRepository) {
    this.usersRepository = usersRepository;
  }

  async findAll() {
    이.사용자모델.모두 찾기()를 반환합니다;
  }
```

```
  findOne(id) {
    return this.userModel.findOne({
      where: {
        id,
      },
    });
  }
```

```
  async remove(id) {
    const user = await this.findOne(id);
    await user.destroy();
  }
}
```

> 경고 UsersModule을 루트 앱모듈로 임포트하는 것을 잊지 마세요.

SequelizeModule.forFeature를 임포트하는 모듈 외부에서 리포지토리를 사용하려면 모듈에서 생성된 프로바이더를 다시 내보내야 합니다. 다음과 같이 전체 모듈을 내보내면 됩니다:

```
@@파일명(users.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/sequelize'에서 { SequelizeModule }을 가져오고,
'./user.entity'에서 { User }를 가져옵니다;

모듈({
  임포트합니다: [SequelizeModule.forFeature([User])], 내보
  내기: [SequelizeModule]
})
사용자 모듈 클래스 {} 내보내기
```

이제 UserHttpModule에서 UsersModule을 가져오면, 후자의 모듈의 프로바이더에서 @InjectModel(User)를 사용할 수 있습니다.

```
@@파일명(users-http.module)
'@nestjs/common'에서 { Module } 가져오기;
'./users.module'에서 { UsersModule } 가져오기;
'./users.service'에서 { UsersService } 가져오기;
'./users.controller'에서 { UsersController }를 가져옵니다;

모듈({
  임포트합니다: [UsersModule], 공급
  자: [UsersService], 컨트롤러:
  [UsersController]
})
내보내기 클래스 UserHttpModule {}
```

관계

관계는 둘 이상의 테이블 간에 설정된 연결입니다. 관계는 각 테이블의 공통 필드를 기반으로 하며, 종종 기본 키와 외래 키가 포함됩니다.

관계에는 세 가지 유형이 있습니다:

일대일	기본 테이블의 모든 행에는 외래 테이블에 연결된 행이 하나만 있습니다.

	테이블
일대다 / 다대일	주 테이블의 모든 행에는 외래 테이블에 하나 이상의 관련 행이 있습니다.
다대다	주 테이블의 모든 행은 외래 테이블에 많은 관련 행이 있고, 외래 테이블의 모든 레코드는 주 테이블에 많은 관련 행이 있습니다.

모델에서 관계를 정의하려면 해당 데코레이터를 사용합니다. 예를 들어 각 사용자 는 여러 장의 사진을 가질 수 있으므로 @HasMany() 데코레이터를 사용합니다.

```
@@파일명(사용자.모델)
'sequelize-typescript'에서 { Column, Model, Table, HasMany } 가져오기; '../
사진/사진.모델'에서 { Photo } 가져오기;

테이블
내보내기 클래스 User extends Model {
  @Column
  이름: 문자열입니다;

  칼럼
  성: 문자열입니다;

  @Column({ defaultValue: true })
  isActive: boolean;

  HasMany(() => 사진) 사진:
  Photo[];
}
```

> 정보 힌트 시퀄라이즈에서 연결에 대해 자세히 알아보려면 이 장을 읽어보세요.

자동 로드 모델

연결 옵션의 모델 배열에 모델을 수동으로 추가하는 작업은 번거로울 수 있습니다. 또한 루트 모듈에서 모델을 참조하면 애플리케이션 도메인 경계가 깨지고 애플리케이션의 다른 부분으로 구현 세부 정보가 유출될 수 있습니다. 이 문제를 해결하려면 아래 그림과 같이 autoLoadModels를 모두 설정하여 모델을 자동으로 로드하고 구성 객체의 동기화 속성(forRoot() 메서드에 전달)을 true로 설정하세요:

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/sequelize'에서 { SequelizeModule }을 가져옵니다;

모듈({ import: [
    SequelizeModule.forRoot({
      ...
```

```
        autoLoadModels: true,
        synchronize: true,
      }),
    ],
})
내보내기 클래스 AppModule {}
```

이 옵션을 지정하면 `forFeature()` 메서드를 통해 등록된 모든 모델이 구성 개체의 `모델` 배열에 자동으로 추가됩니다.

> 경고 `forFeature()` 메서드를 통해 등록되지 않고 모델에서 (연관을 통해서만) 참조되는 모델은 포함되지 않습니다.

거래 시퀄라이즈

데이터베이스 트랜잭션은 데이터베이스 관리 시스템 내에서 데이터베이스에 대해 수행되는 작업 단위를 의미하며, 다른 트랜잭션과 독립적으로 일관성 있고 신뢰할 수 있는 방식으로 처리됩니다. 트랜잭션은 일반적으로 데이터베이스의 모든 변경 사항을 나타냅니다(자세히 알아보기).

시퀄라이즈 트랜잭션을 처리하는 전략에는 여러 가지가 있습니다. 다음은 관리되는 트랜잭션(자동 콜백)의 샘플 구현입니다.

먼저 일반적인 방법으로 `Sequelize` 객체를 클래스에 주입해야 합니다:

```
@Injectable()
export class UsersService {
  constructor(private sequelize: Sequelize) {}
}
```

> 정보 힌트 `시퀄라이즈` 클래스는 `시퀄라이즈-타입스크립트` 패키지에서 가져옵니다.

이제 이 객체를 사용하여 트랜잭션을 생성할 수 있습니다.

```
async createMany() {
  try {
    await this.sequelize.transaction(async t => {
      const transactionHost = { 트랜잭션: t };

      await this.userModel.create(
          { firstName: '아브라함', lastName: '링컨' }, 트랜잭션호스트
          ,
      );
      await this.userModel.create(
          { firstName: 'John', lastName: '부스' }, 트랜
          잭션호스트,
      );
    });
  } catch (err) {
    // 트랜잭션이 롤백되었습니다.
```

```
      // 오류는 트랜잭션 콜백에 반환된 프로미스 체인이 거부한 모든 것입니다.
    }
  }
```

> 정보 힌트 `Sequelize` 인스턴스는 트랜잭션을 시작할 때만 사용된다는 점에 유의하세요. 그러나 이 클래스를 테스트하려면 여러 메서드를 노출하는 전체 `Sequelize` 객체를 모킹해야 합니다. 따라서 헬퍼 팩토리 클래스(예: `TransactionRunner`)를 사용하고 트랜잭션을 유지하는 데 필요한 제한된 메서드 집합으로 인터페이스를 정의하는 것이 좋습니다. 이 기법을 사용하면 이러한 메서드를 매우 간단하게 모킹할 수 있습니다.

마이그레이션

마이그레이션은 데이터베이스의 기존 데이터를 보존하면서 데이터베이스 스키마를 점진적으로 업데이트하여 애플리케이션의 데이터 모델과 동기화 상태를 유지할 수 있는 방법을 제공합니다. 마이그레이션을 생성, 실행 및 되돌리기 위해 Sequelize는 전용 CLI를 제공합니다.

마이그레이션 클래스는 Nest 애플리케이션 소스 코드와는 별개입니다. 마이그레이션 클래스의 수명 주기는 Sequelize CLI에 의해 유지 관리됩니다. 따라서 마이그레이션을 통해 종속성 주입 및 기타 Nest 특정 기능을 활용할 수 없습니다. 마이그레이션에 대해 자세히 알아보려면 Sequelize 설명서의 가이드를 참조하세요.

여러 데이터베이스

일부 프로젝트에는 여러 데이터베이스 연결이 필요합니다. 이 모듈을 사용하면 이 작업도 수행할 수 있습니다. 여러 연결로 작업하려면 먼저 연결을 만듭니다. 이 경우 연결 이름 지정은 필수입니다.

자체 데이터베이스에 저장된 `앨범` 엔티티가 있다고 가정해 보겠습니다.

```
const defaultOptions = {
  dialect: 'postgres',
  port: 5432,
  사용자 이름: '사용자', 비밀번호
  : '비밀번호', 데이터베이스:
  'db', synchronize: true,
};

모듈({ import: [
    SequelizeModule.forRoot({
      ...defaultOptions,
      host: 'user_db_host',
      models: [사용자],
    }),
    SequelizeModule.forRoot({
      ...기본옵션,
      name: 'albumsConnection',
      host: 'album_db_host',
      models: [앨범],
```

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

> 경고 연결의 이름을 설정하지 않으면 해당 이름이 기본값으로 설정됩니다. 이름이 없거나 같은 이름을 가진 연결이 여러 개 있으면 재정의되므로 주의하세요.

이 시점에서 사용자 및 앨범 모델이 자체 연결로 등록되어 있습니다. 이 설정을 사용하면 SequelizeModule.forFeature() 메서드와 @InjectModel() 데코레이터에 어떤 연결을 사용해야 하는지 알려줘야 합니다. 연결 이름을 전달하지 않으면 기본 연결이 사용됩니다.

```
모듈({ import: [
    SequelizeModule.forFeature([User]),
    SequelizeModule.forFeature([Album], 'albumsConnection'),
  ],
})
내보내기 클래스 AppModule {}
```

지정된 연결에 대해 Sequelize 인스턴스를 삽입할 수도 있습니다:

```
@Injectable()
내보내기 클래스 앨범 서비스 { 생성자(
    주입 연결('albumsConnection') 비공개 readonly 시퀄라이즈:
    시퀄라이즈,
  ) {}
}
```

공급자에게 시퀄라이즈 인스턴스를 주입할 수도 있습니다:

```
모듈({
  providers: [
    {
      제공 앨범 서비스,
      useFactory: (albumsSequelize: Sequelize) => { 반환
        새 앨범 서비스(albumsSequelize);
      },
      주입합니다: [getDataSourceToken('albumsConnection')],
    },
  ],
})
내보내기 클래스 앨범 모듈 {}
```

테스트

애플리케이션을 단위 테스트할 때 일반적으로 데이터베이스 연결을 피하고 테스트 스위트를 독립적으로 유지하며 실행 프로세스를 가능한 한 빠르게 유지하려고 합니다. 하지만 클래스가 연결 인스턴스에서 가져온 모델에 의존할 수 있습니다. 이를 어떻게 처리할까요? 해결책은 모의 모델을 만드는 것입니다. 이를 위해 사용자 지정 공급자를 설정합니다. 등록된 각 모델은 자동으로 `<모델명>모델` 토큰으로 표시되며, 여기서 `모델명은` 모델 클래스의 이름입니다.

`nestjs/sequelize` 패키지는 주어진 모델을 기반으로 준비된 토큰을 반환하는 `getModelToken()` 함수를 노출합니다.

```
모듈({ providers:
  [
    사용자 서비스,
    {
      제공: getModelToken(User), 사용값:
      mockModel,
    },
  ],
})
사용자 모듈 클래스 {} 내보내기
```

이제 대체 mockModel이 `UserModel로` 사용됩니다. 어떤 클래스가 `UserModel을` 요청할 때마다 `인젝트 모델()` 데코레이터를 사용하면 Nest는 등록된 `mockModel` 객체를 사용합니다. 비

동기 구성

정적이 아닌 비동기적으로 `SequelizeModule` 옵션을 전달하고 싶을 수도 있습니다. 이 경우 비동기 구성을 처리하는 여러 가지 방법을 제공하는 `forRootAsync()` 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
SequelizeModule.forRootAsync({
  useFactory: () => ({
    방언을 사용합니다:
    'mysql', host:
    'localhost', port:
    3306, 사용자명:
    'root', 비밀번호:
    'root', 데이터베이스
    : 'test', models:
    [],
  }),
});
```

저희 팩토리는 다른 비동기 공급자와 마찬가지로 작동합니다(예: 비동기일 수 있고 주입을 통해 종속성을 주입할 수 있습니다).

```
SequelizeModule.forRootAsync({
  import: [ConfigModule],
  useFactory: (configService: ConfigService) => ({
    dialect: 'mysql',
    host:  configService.get('HOST'),
    port: +configService.get('PORT'),
    username: configService.get('USERNAME'),
    password: configService.get('PASSWORD'),
    database: configService.get('DATABASE'),
    models: [],
  }),
  주입합니다: [구성 서비스],
});
```

또는 `useClass` 구문을 사용할 수도 있습니다:

```
SequelizeModule.forRootAsync({
  useClass: SequelizeConfigService,
});
```

위의 구조는 `SequelizeModule` 내부에 `SequelizeConfigService`를 `인스턴스화하고`이를 사용하여 `createSequelizeOptions()`를 호출하여 옵션 객체를 제공합니다. 이는 아래 그림과 같이 SequelizeConfigService가 `SequelizeOptionsFactory` 인터페이스를 구현해야 한다는 것을 의미합니다:

```
@Injectable()
SequelizeConfigService 클래스는 SequelizeOptionsFactory {
  createSequelizeOptions() 를 구현합니다: SequelizeModuleOptions {
    반환 {
      방언을 사용합니다:
      'mysql', host:
      'localhost', port:
      3306, 사용자명:
      'root', 비밀번호:
      'root', 데이터베이스:
      'test', models:
      [],
    };
  }
}
```

`SequelizeModule` 내부에 `SequelizeConfigService`를 생성하지 않고 다른 모듈에서 가져온 프로바이더를 사용하려면 `useExisting` 구문을 사용하면 됩니다.

```
SequelizeModule.forRootAsync({
  import: [ConfigModule],
```

```
    사용Existing: ConfigService,
  });
```

이 구조는 `useClass와` 동일하게 작동하지만 한 가지 중요한 차이점이 있습니다. SequelizeModule은 가져온 모듈을 조회하여 새 `구성 서비스를` 인스턴스화하는 대신 기존 `구성 서비스를` 재사용합니다.

예

작동 예제는 여기에서 확인할 수 있습니다.

몽고

Nest는 두 가지 방법으로 MongoDB 데이터베이스와 통합할 수 있습니다. 여기에 설명된 내장 TypeORM 모듈을 사용하거나, MongoDB용 커넥터가 있는 내장 TypeORM 모듈을 사용하거나, 가장 널리 사용되는 MongoDB 객체 모델링 도구인 Mongoose를 사용할 수 있습니다. 이 장에서는 전용 `@nestjs/mongoose` 패키지를 사용하여 후자를 설명하겠습니다.

필요한 종속성을 설치하는 것으로 시작하세요:

```
npm i @nestjs/mongoose 몽구스 몽구스
```

설치 프로세스가 완료되면 `몽구스모듈을` 루트 `앱모듈로` 가져올 수 있습니다.

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/mongoose'에서 { 몽구스모듈 }을 가져옵니다;

모듈({
  임포트합니다: [몽구스모듈.forRoot('mongodb://localhost/nest')],
})
내보내기 클래스 AppModule {}
```

`forRoot()` 메서드는 여기에 설명된 대로 Mongoose 패키지의 `mongoose.connect()`와 동일한 구성 객체를 받아들입니다.

모델 주입

몽구스에서는 모든 것이 스키마에서 파생됩니다. 각 스키마는 MongoDB 컬렉션에 매핑되며 해당 컬렉션 내의 문서 모양을 정의합니다. 스키마는 모델을 정의하는 데 사용됩니다. 모델은 기본 MongoDB 데이터베이스에서 문서를 만들고 읽는 일을 담당합니다.

스키마는 NestJS 데코레이터를 사용하거나 몽구스 자체에서 수동으로 생성할 수 있습니다. 데코레이터를 사용하여 스키마를 생성하면 상용구가 크게 줄어들고 전반적인 코드 가독성이 향상됩니다.

`CatSchema를` 정의해 보겠습니다:

```
@@파일명(스키마/캣.스키마)
'@nestjs/mongoose'에서 { Prop, Schema, SchemaFactory }를 가져오고,
'mongoose'에서 { HydratedDocument }를 가져옵니다;

내보내기 유형 CatDocument = HydratedDocument<Cat>;

@Schema()
내보내기 클래스 Cat {
  @Prop()
  이름: 문자열;
```

```
  @Prop()
  나이: 숫자;

  Prop() 품종: 문
  자열;
}

export const CatSchema = SchemaFactory.createForClass(Cat);
```

> 정보 힌트 nestjs/mongoose의 DefinitionsFactory 클래스를 사용하여 원시 스키마 정의를 생성할 수도 있습니다. 이렇게 하면 제공한 메타데이터를 기반으로 생성된 스키마 정의를 수동으로 수정할 수 있습니다. 이는 데코레이터로 모든 것을 표현하기 어려울 수 있는 특정 에지 케이스에 유용합니다.

스키마() 데코레이터는 클래스를 스키마 정의로 표시합니다. 이 데코레이터는 Cat 클래스를 같은 이름의 몽고 DB 컬렉션에 매핑하지만 끝에 "s"가 추가되므로 최종 몽고 컬렉션 이름은 cats가 됩니다. 이 데코레이터는 스키마 옵션 객체인 단일 선택적 인수를 받습니다. 이 객체는 일반적으로 mongoose.Schema 클래스 생성자의 두 번째 인수로 전달할 수 있는 객체라고 생각하면 됩니다(예: new mongoose.Schema(_, options)). 사용 가능한 스키마 옵션에 대해 자세히 알아보려면 이 장을 참조하세요.

Prop() 데코레이터는 문서에서 속성을 정의합니다. 예를 들어, 위의 스키마 정의에서는 이름, 나이, 품종이라는 세 가지 속성을 정의했습니다. 이러한 속성의 스키마 유형은 TypeScript 메타데이터(및 리플렉션) 기능 덕분에 자동으로 추론됩니다. 그러나 유형을 암시적으로 반영할 수 없는 더 복잡한 시나리오(예: 배열 또는 중첩된 객체 구조)에서는 다음과 같이 유형을 명시적으로 표시해야 합니다:

```
@Prop([문자열]) 태
그: 문자열[];
```

또는 @Prop() 데코레이터는 옵션 객체 인수를 받습니다(사용 가능한 옵션에 대해 자세히 알아보기). 이를 통해 프로퍼티가 필요한지 여부를 표시하거나 기본값을 지정하거나 변경 불가능으로 표시할 수 있습니다. 예를 들어

```
@Prop({ 필수: true })
name: 문자열;
```

나중에 채우기 위해 다른 모델과의 관계를 지정하려는 경우 @Prop() 데코레이터를 사용할 수도 있습니다. 예를

들어 `Cat에 소유자라는` 다른 컬렉션에 저장된 소유자가 있는 경우 프로퍼티에 유형과 참조가 있어야 합니다. 예를 들어

```
'몽구스'에서 *를 몽구스로 가져옵니다;
'../owners/schemas/owner.schema'에서 { Owner }를 가져옵니다;
```

```
// 클래스 정의 내부
Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'Owner' }) 소
유자: 소유자;
```

소유자가 여러 명인 경우, 숙소 구성은 다음과 같이 표시되어야 합니다:

```
Prop({ 유형: [{ 유형: mongoose.Schema.Types.ObjectId, ref: 'Owner' }] }) 소유자:
소유자[];
```

마지막으로, 원시 스키마 정의를 데코레이터에 전달할 수도 있습니다. 이는 예를 들어 프로퍼티가 클래스로 정의되지 않은 중첩된 객체를 나타낼 때 유용합니다. 이를 위해 다음과 같이 `@nestjs/mongoose` 패키지의 `raw()` 함수를 사용합니다:

```
@Prop(raw({
  firstName: { type: 문자열 }, 성 {
  유형: String }
}))
세부 정보: 레코드<스트링, 임의>;
```

또는 데코레이터를 사용하지 않으려는 경우 스키마를 수동으로 정의할 수 있습니다. 예를 들어

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

`cat.schema` 파일은 `cats` 디렉터리의 폴더에 있으며, 이 폴더에서 `CatsModule도` 정의합니다. 스키마 파일은 원하는 위치에 저장할 수 있지만, 관련 도메인 객체 근처의 적절한 모듈 디렉터리에 저장하는 것이 좋습니다.

`CatsModule을` 살펴봅시다:

```
@@파일명(cats.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/mongoose'에서 { 몽구스모듈 } 임포트;
'./cats.controller'에서 { 캣츠컨트롤러 } 임포트;
'./cats.service'에서 { 캣츠서비스 } 임포트;
'./schmas/cat.schema'에서 { Cat, CatSchema }를 가져옵니다;

모듈({
  임포트합니다: [몽구스모듈.forFeature([{ 이름: Cat.name, 스키마: CatSchema
}])],
  컨트롤러: [CatsController], 제공자:
  [CatsService],
```

```
})
내보내기 클래스 CatsModule {}
```

몽구스모듈은 현재 범위에 등록할 모델을 정의하는 등 모듈을 구성할 수 있는 `forFeature()` 메서드를 제공합니다. 다른 모듈에서도 모델을 사용하려면 `CatsModule의 내보내기` 섹션에 몽구스모듈을 추가하고 다른 모듈에서 CatsModule을 가져오면 됩니다.

스키마를 등록한 후에는 다음을 사용하여 `Cat` 모델을 `CatsService에` 주입할 수 있습니다.

`@InjectModel()` 데코레이터:

```
@@파일명(cats.service)
'몽구스'에서 { 모델 }을 가져옵니다;
'@nestjs/common'에서 { Injectable }을 가져오고,
'@nestjs/mongoose'에서 { InjectModel }을 가져오고,
'./schemas/cat.schema'에서 { Cat }을 가져옵니다;
'./dto/create-cat.dto'에서 { CreateCatDto }를 가져옵니다;

@Injectable()
내보내기 클래스 CatsService {
  생성자(@InjectModel(Cat.name) private catModel: Model<Cat>) {}

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

  비동기 findAll(): Promise<Cat[]> {
    return this.catModel.find().exec();
  }
}
@@switch
'몽구스'에서 { 모델 }을 가져옵니다;
'@nestjs/common'에서 { Injectable, Dependencies }를 가져오고,
'@nestjs/mongoose'에서 { getModelToken }을 가져옵니다;
'./chemas/cat.schema'에서 { Cat }을 가져옵니다;

주입 가능()
@Dependencies(getModelToken(Cat.name))
내보내기 클래스 CatsService {
  constructor(catModel) {
```

```
    this.catModel = catModel;
  }

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

  async findAll() {
    이.catModel.find().exec()를 반환합니다;
```

```
  }
}
```

연결

때로는 네이티브 몽구스 연결 객체에 액세스해야 할 수도 있습니다. 예를 들어 연결 객체에서 네이티브 API 호출을 하고 싶을 수 있습니다. 다음과 같이 `@InjectConnection()` 데코레이터를 사용하여 몽구스 커넥션을 삽입할 수 있습니다:

```
'@nestjs/common'에서 { Injectable }을 임포트합니다;
'@nestjs/mongoose'에서 { InjectConnection }을 임포트하고,
'mongoose'에서 { Connection }을 임포트합니다;

@Injectable()
내보내기 클래스 CatsService {
  생성자(@InjectConnection() 비공개 연결: Connection) {}
}
```

여러 데이터베이스

일부 프로젝트에는 여러 데이터베이스 연결이 필요합니다. 이 모듈을 사용하면 이 작업도 수행할 수 있습니다. 여러 연결로 작업하려면 먼저 연결을 만듭니다. 이 경우 연결 이름 지정은 필수입니다.

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/mongoose'에서 { 몽구스모듈 }을 가져옵니다;

모듈({ import: [
    몽구스모듈.forRoot('mongodb://localhost/test', {
      connectionName: 'cats',
    }),
    MongooseModule.forRoot('mongodb://localhost/users', {
      connectionName: 'users',
    }),
  ],
})
내보내기 클래스 AppModule {}
```

> 경고 주의 이름이 없거나 같은 이름의 연결이 여러 개 있으면 재정의됩니다.

이 설정을 사용하면 `MongooseModule.forFeature()` 함수에 어떤 연결을 사용해야 하는지 알려줘야 합니다
.

```
```

```
모듈({ import: [
    몽구스모듈.forFeature([{ 이름: Cat.name, 스키마: CatSchema }], 'cats'),
  ],
})
내보내기 클래스 CatsModule {}
```

지정된 연결에 대한 `연결을` 삽입할 수도 있습니다:

```
'@nestjs/common'에서 { Injectable }을 임포트합니다;
'@nestjs/mongoose'에서 { InjectConnection }을 임포트하고,
'mongoose'에서 { Connection }을 임포트합니다;

@Injectable()
내보내기 클래스 CatsService {
  생성자(@InjectConnection('cats') 비공개 연결: Connection) {}
}
```

지정된 `연결을` 사용자 지정 공급자(예: 팩토리 공급자)에 주입하려면 `getConnectionToken()` 함수에 연결 이름을 인수로 전달합니다.

```
{
  제공하세요: CatsService,
  useFactory: (catsConnection: Connection) => {
    return new CatsService(catsConnection);
  },
  주입합니다: [getConnectionToken('cats')],
}
```

명명된 데이터베이스에서 모델을 주입하려는 경우 연결 이름을 `@InjectModel()` 데코레이터의 두 번째 매개변수로 사용할 수 있습니다.

```
@@파일명(cats.service)
@Injectable()
내보내기 클래스 CatsService {
  constructor(@InjectModel(Cat.name, 'cats') private catModel: Model<Cat>)
{}
}
@@스위치 @Injectable()
@Dependencies(getModelToken(Cat.name, 'cats'))
export class CatsService {
  constructor(catModel) {
    this.catModel = catModel;
```

```
    }
  }
```

후크(미들웨어)

미들웨어(프리훅 및 포스트훅이라고도 함)는 비동기 함수를 실행하는 동안 제어권을 전달하는 함수입니다. 미들웨어는 스키마 수준에서 지정되며 플러그인(소스) 작성에 유용합니다. 몽구스에서는 모델을 컴파일한 후 `pre()` 또는 `post()`를 호출하면 작동하지 않습니다. 모델 등록 전에 훅을 등록하려면 팩토리 프로바이더(예: `useFactory`)와 함께 `MongooseModule의 forFeatureAsync()` 메서드를 사용합니다. 이 기법을 사용하면 스키마 객체에 액세스한 다음 `pre()` 또는 `post()` 메서드를 사용하여 해당 스키마에 훅을 등록할 수 있습니다. 아래 예시를 참조하세요:

```
모듈({ import: [
    몽구스모듈.forFeatureAsync([[
      {
        name: Cat.name,
        useFactory: () => {
          const schema = CatsSchema;
          schema.pre('save', function () {
            console.log('안녕하세요 저장 전입니다');
          });
          반환 스키마;
        },
      },
    ]),
  ],
})
내보내기 클래스 AppModule {}
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 `비동기화될` 수 있으며 다음을 통해 종속성을 주입할 수 있습니다. `주입합니다`.

```
모듈({ import: [
    몽구스모듈.forFeatureAsync([[
      {
        이름: Cat.name,
        임포트: [구성 모듈],
        useFactory: (configService: ConfigService) => {
          const schema = CatsSchema; schema.pre('save',
          function() {
            콘솔 로그(
              `${configService.get('APP_NAME')}: 안녕하세요, 사전 저장에서`,
            ),
          });
          반환 스키마;
        },
```

```
        주입합니다: [구성 서비스],
      },
    ]),
  ],
})
내보내기 클래스 AppModule {}
```

플러그인

주어진 스키마에 대한 플러그인을 등록하려면 `forFeatureAsync()` 메서드를 사용합니다.

```
모듈({ import: [
    몽구스모듈.forFeatureAsync([[
      {
        name: Cat.name,
        useFactory: () => {
          const schema = CatsSchema;
          schema.plugin(require('mongoose-autopopulate'));
          return schema;
        },
      },
    ]),
  ],
})
내보내기 클래스 AppModule {}
```

모든 스키마에 대한 플러그인을 한 번에 등록하려면 `Connection` 객체의 `.plugin()` 메서드를 호출합니다. 모델을 생성하기 전에 연결에 액세스해야 하며, 이를 위해 `connectionFactory`를 사용합니다:

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/mongoose'에서 { 몽구스모듈 }을 가져옵니다;

모듈({ import: [
    MongooseModule.forRoot('mongodb://localhost/test', {
      connectionFactory: (connection) => {
        connection.plugin(require('mongoose-autopopulate')); 반환
        연결;
      }
    }),
  ],
})
내보내기 클래스 AppModule {}
```

차별 주의자

판별자는 스키마 상속 메커니즘입니다. 이를 통해 동일한 기본 MongoDB 컬렉션 위에 스키마가 겹치는 여러 모델을 가질 수 있습니다.

단일 컬렉션에서 다양한 유형의 이벤트를 추적하고 싶다고 가정해 보겠습니다. 모든 이벤트에는 타임스탬프가 있습니다.

```
@@파일명(이벤트.스키마)
스키마({ 판별자 키: '종류' }) 내보내기 클래스 이
벤트 {
  @Prop({
    유형입니다: 문자열,
    필수: true,
    열거형: [ClickedLinkEvent.name, SignUpEvent.name],
  })
  종류: 문자열;

  Prop({ 유형: 날짜, 필수: 참 }) 시간: 날짜;
}
```

> 정보 힌트 몽구스가 다른 판별자 모델 간의 차이를 구분하는 방법은 "판별자 키"를 사용하는 것입니다. _____입니다. 몽구스는 스키마에 문자열 경로인 _____라는 문자열 경로를 스키마에 추가하여 이 문서가 어떤 판별자의 인스턴스인지 추적하는 데 사용합니다. 판별자 키 옵션을 사용하여 판별 경로를 정의할 수도 있습니다.

SignedUpEvent 및 ClickedLinkEvent 인스턴스는 일반 이벤트와 동일한 컬렉션에 저장됩니다.

이제 다음과 같이 ClickedLinkEvent 클래스를 정의해 보겠습니다:

```
@@파일명(클릭-링크-이벤트.스키마) @Schema()
내보내기 클래스 ClickedLinkEvent { 종
  류: 문자열;
  시간: 날짜;

  @Prop({ 유형: 문자열, 필수: true }) url: 문자열
  ;
}

export const ClickedLinkEventSchema =
SchemaFactory.createForClass(ClickedLinkEvent);
```

그리고 `SignUpEvent` 클래스:

```
@@파일명(가입-이벤트.스키마) @Schema()
내보내기 클래스 SignUpEvent {
  종류: 문자열;
  시간: 날짜;

  @Prop({ 유형: 문자열, 필수: true }) 사용자: 문자
  열;
}

export const SignUpEventSchema =
SchemaFactory.createForClass(SignUpEvent);
```

이렇게 하면 판별자 옵션을 사용하여 주어진 스키마에 대한 판별자를 등록할 수 있습니다. 이 옵션은 `몽구스모듈.forFeature`와 `몽구스모듈.forFeatureAsync` 모두에서 작동합니다:

```
@@파일명(event.module)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/mongoose'에서 { 몽구스모듈 }을 가져옵니다;

모듈({ import: [
    몽구스모듈.forFeature([[
      {
        이름: Event.name, 스키마
        : EventSchema, 판별자:
        [
          { 이름: ClickedLinkEvent.name, 스키마: ClickedLinkEventSchema },
          { 이름: SignUpEvent.name, 스키마: SignUpEventSchema },
        ],
      },
    ]),
  ]
})
이벤트 모듈 클래스 {} 내보내기
```

테스트

애플리케이션을 단위 테스트할 때 일반적으로 데이터베이스 연결을 피하여 테스트 스위트를 더 간단하게 설정하고 실행 속도를 높이고자 합니다. 하지만 클래스는 연결 인스턴스에서 가져온 모델에 의존할 수 있습니다. 이러한 클

래스를 어떻게 해결할 수 있을까요? 해결책은 모의 모델을 만드는 것입니다.

이 작업을 더 쉽게 하기 위해 `@nestjs/mongoose` 패키지는 토큰 이름에 따라 준비된 인젝션 토큰을 반환하는 `getModelToken()` 함수를 노출합니다. 이 토큰을 사용하면 `사용 클래스`, 사용 `값`, `사용 팩토리` 등 표준 사용자 정의 공급자 기법을 사용하여 모의 구현을 쉽게 제공할 수 있습니다. 예를 들어

```
모듈({ providers:
  [
    CatsService,
    {
      제공: getModelToken(Cat.name), 사용값:
      catModel,
    },
  ],
})
내보내기 클래스 CatsModule {}
```

이 예제에서는 소비자가 객체 인스턴스를 주입할 때마다 하드코딩된 `catModel`(객체 인스턴스)이 제공됩니다.

`Model<Cat>에 @InjectModel()` 데코레이터를 사용

합니다. 비동기 구성

모듈 옵션을 정적이 아닌 비동기적으로 전달해야 하는 경우, `forRootAsync()`
메서드를 사용합니다. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성을 처리하는 몇 가지 기술을 제공합니

다. 한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
MongooseModule.forRootAsync({
  useFactory: () => ({
    uri: 'mongodb://localhost/nest',
  }),
});
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 `비동기화될` 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.

`주입합니다`.

```
MongooseModule.forRootAsync({
  import: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    uri: configService.get<string>('MONGODB_URI'),
  }),
  주입합니다: [구성 서비스],
});
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 `몽구스` 모듈을 구성할 수도 있습니다:

```
MongooseModule.forRootAsync({
  useClass: 몽구스컨피그서비스,
});
```

위의 구성은 `MongooseModule` 내부에 MongooseConfigService를 인스턴스화하여 이를 사용하여 필요한 옵션 객체를 생성합니다. 이 예제에서 `MongooseConfigService는` 아래와 같이 `MongooseOptionsFactory` 인터페이스를 구현해야 한다는 점에 유의하세요. `몽구스모듈은` 제공된 클래스의 인스턴스화된 객체에서 `createMongooseOptions()` 메서드를 호출합니다.

```
@Injectable()
export 클래스 몽구스컨피그서비스는 몽구스옵션팩토리를 구현합니다 { createMongoose옵션():
  MongooseModuleOptions {
    반환 {
      uri: 'mongodb://localhost/nest',
    };
  }
}
```

내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려는 경우 몽구스모듈에 `사용Existing` 구문을 사용합니다.

```
MongooseModule.forRootAsync({
  import: [ConfigModule],
  useExisting: ConfigService,
});
```

예

작동하는 예제는 여기에서 확인할 수 있습니다.

유효성 검사

웹 애플리케이션으로 전송되는 모든 데이터의 정확성을 검증하는 것이 가장 좋습니다. 들어오는 요청의 유효성을 자동으로 검사하기 위해 Nest는 바로 사용할 수 있는 여러 파이프를 제공합니다:

- `ValidationPipe`
- `ParseIntPipe`
- `ParseBoolPipe`
- `ParseArrayPipe`
- `ParseUUIDPipe`

유효성 검사 파이프는 강력한 클래스 유효성 검사기 패키지와 선언적 유효성 검사 데코레이터를 사용합니다. `ValidationPipe`는 들어오는 모든 클라이언트 페이로드에 대해 유효성 검사 규칙을 적용하는 편리한 접근 방식을 제공하며, 특정 규칙은 각 모듈의 로컬 클래스/DTO 선언에 간단한 어노테이션으로 선언됩니다.

개요

파이프 챕터에서는 간단한 파이프를 빌드하고 컨트롤러, 메서드 또는 글로벌 앱에 바인딩하는 과정을 통해 프로세스가 어떻게 작동하는지 보여드렸습니다. 이 장의 주제를 가장 잘 이해하려면 해당 장을 반드시 복습하세요. 여기에서는 ValidationPipe의 다양한 실제 사용 사례에 초점을 맞추고 고급 사용자 정의 기능 중 일부를 사용하는 방법을 보여드리겠습니다.

기본 제공 ValidationPipe 사용

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

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

> 정보 힌트 ValidationPipe는 `@nestjs/common` 패키지에서 내보냅니다.

이 파이프는 `클래스 유효성` 검사기 및 `클래스 트랜스포머` 라이브러리를 사용하므로 사용할 수 있는 옵션이 많습니다. 파이프에 전달된 구성 개체를 통해 이러한 설정을 구성합니다. 다음은 기본 제공 옵션입니다:

```
내보내기 인터페이스 ValidationPipeOptions extends ValidatorOptions {
  transform?: boolean;
  disableErrorMessages?: boolean;
  예외 팩토리?: (오류: 유효성 검사 오류[]) => any;
}
```

이 외에도 모든 클래스 유효성 검사기 옵션(유효성 검사기 옵션 인터페이스에서 상속됨)을 사용할 수 있습니다:

옵션	유형	설명

`enableDebugMessage`	부울	true로 설정하면, 유효성 검사기가 추가 경고 메시지를 인쇄합니다. 를 콘솔로 전송하세요.
`skipUndefinedProperties`	부울	true로 설정하면 유효성 검사기는 모든 유효성 검사를 건너뜁니다. 유효성 검사 객체에 정의되지 않은 프로퍼티가 있습니다.
`skipNullProperties`	`boolean`	true로 설정하면 유효성 검사기는 유효성 검사 대상에서 null인 모든 프로퍼티의 유효성 검사를 건너뜁니다.
`skipMissingProperties`	`boolean`	true로 설정하면 유효성 검사기는 유효성 검사 객체에서 null이거나 정의되지 않은 모든 프로퍼티의 유효성 검사를 건너뜁니다.
`화이트리스트`	부울	true로 설정하면 유효성 검사기는 유효성 검사 데코레이터를 사용하지 않는 모든 프로퍼티에서 유효성 검사된(반환된) 객체를 제거합니다.
`forbidNonWhitelisted`	부울	true로 설정하면 화이트리스트에 없는 프로퍼티를 제거하는 대신 유효성 검사기가 예외를 던집니다.
`forbidUnknownValues`	부울	true로 설정하면 알 수 없는 객체의 유효성 검사 시도가 즉시 실패합니다.
`disableErrorMessage`	부울	true로 설정하면 유효성 검사 오류가 클라이언트에 반환되지 않습니다. 이 설정을 사용하면 오류 발생 시 어떤 예외 유형을 사용할지 지정할 수 있습니다. 기본적으로 `BadRequestException`을 던집니다.
`errorHttpStatusCode`	숫자	유효성 검사 오류의 배열을 받아 예외 객체를 반환합니다.
`예외 팩토리`	함수	
`groups`	`문자열[]`	객체의 유효성 검사 중에 사용할 그룹입니다.
`항상`	`boolean`	데코레이터의 `항상` 옵션을 기본값으로 설정합니다. 기본값은 다음과 같습니다.

데코레이터 옵션에서 재정의할 수 있습니다.

`strictGroups` 부울 그룹이 지정되지 않았거나 비어 있으면 그룹이 하나 이상 있는 데코레이터를 무시합니다. `dismissDefaultMessages` 부울 참으로 설정하면 유효성 검사에서 기본 메시지를 사용하지 않습니다. 명시적으로 설정되지 않은 경우 오류 메시지는 항상 정의되지 않습니다. `validationError.target` `boolean` 타겟을 유효성 검사 오류에 노출할지 여부를 나타냅니다. `validationError.value` `boolean` 유효성 검사 값을 유효성 검사 오류에 노출할지 여부를 나타냅니다. `stopAtFirstError` `boolean` true로 설정하면 첫 번째 오류가 발생한 후 지정된 프로퍼티의 유효성 검사가 중지됩니다. 기본값은 false입니다.

> 정보 공지 클래스 유효성 검사기 패키지에 대한 자세한 내용은 해당 저장소에서 확인하세요.

자동 유효성 검사

애플리케이션 수준에서 `ValidationPipe`를 바인딩하여 모든 엔드포인트가 잘못된 데이터를 수신하지 않도록 보호하는 것부터 시작하겠습니다.

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

파이프를 테스트하기 위해 기본 엔드포인트를 만들어 보겠습니다.

```
@Post()
create(@Body() createUserDto: CreateUserDto) {
  return '이 작업은 새 사용자를 추가합니다';
}
```

> 정보 힌트 TypeScript는 제네릭이나 인터페이스에 대한 메타데이터를 저장하지 않으므로 DTO에서 제네릭이나 인터페이스를 사용할 경우 ValidationPipe가 들어오는 데이터의 유효성을 제대로 검사하지 못할 수 있습니다. 따라서 DTO에 구체적인 클래스를 사용하는 것이 좋습니다.

> 정보 힌트 DTO를 가져올 때는 런타임에 퓨워치르로 유형 전용 가져오기를 사용할 수 없습니다. 즉, {{ '{' }}로 가져와야 합니다. CreateUserDto {{ '}' }} 유형 대신 {{ '{' }}을 임포트해야 합니다. CreateUserDto {{ '}' }}.

이제 CreateUserDto에 몇 가지 유효성 검사 규칙을 추가할 수 있습니다. 여기에 자세히 설명된 클래스 유효성 검사기 패키지에서 제공하는 데코레이터를 사용하여 이 작업을 수행합니다. 이렇게 하면 CreateUserDto를 사용하는 모든 경로에 이러한 유효성 검사 규칙이 자동으로 적용됩니다.

```
'class-validator'에서 { IsEmail, IsNotEmpty } import;

export class CreateUserDto {
  @IsEmail() 이메
  일: 문자열;

  IsNotEmpty() 비밀번호:
  문자열;
}
```

이러한 규칙을 적용하면 요청 본문에 잘못된 이메일 속성이 포함된 요청이 엔드포인트에 도달하면 애플리케이션이 자동으로 400 잘못된 요청 코드와 함께 다음 응답 본문으로 응답합니다:

```
{
  "statusCode": 400, "오류":
  "잘못된 요청",
  "메시지": ["이메일은 이메일이어야 합니다"]
}
```

요청 본문의 유효성을 검사하는 것 외에도, 다른 요청 객체 프로퍼티에도 ValidationPipe를 사용할 수 있습니다. 엔드포인트 경로에 :id를 허용하고 싶다고 가정해 보겠습니다. 이 요청 매개변수에 숫자만 허용되도록 하기 위해 다음 구문을 사용할 수 있습니다:

```
@Get(':id')
findOne(@Param() params: FindOneParams) {
  return '이 액션은 사용자를 반환합니다';
}
```

FindOneParams는 DTO와 마찬가지로 클래스 유효성 검사기를 사용하여 유효성 검사 규칙을 정의하는 클래스일 뿐입니다. 다음과 같이 보일 것입니다:

```
'class-validator'에서 { IsNumberString } 가져오기; 내보내기

클래스 FindOneParams {
  IsNumberString()
  id: 숫자;
}
```

세부 오류 비활성화

오류 메시지는 요청에 무엇이 잘못되었는지 설명하는 데 도움이 될 수 있습니다. 그러나 일부 프로덕션 환경에서는 자세한 오류를 비활성화하는 것을 선호합니다. 이렇게 하려면 옵션 객체를 ValidationPipe에 전달하면 됩니다 :

```
app.useGlobalPipes(
  new ValidationPipe({
    disableErrorMessages: true,
  }),
);
```

따라서 응답 본문에 자세한 오류 메시지가 표시되지 않습니다. 속성 스트리핑

또한 메서드 핸들러가 수신해서는 안 되는 프로퍼티를 필터링할 수 있는 ValidationPipe를 사용할 수 있습니다. In
이 경우 허용 가능한 속성을 화이트리스트에 추가할 수 있으며, 화이트리스트에 포함되지 않은 속성은 결과 개

체에서 자동으로 제거됩니다. 예를 들어, 처리기에서 `이메일` 및 `비밀번호` 속성을 기대하지만 요청에 `나이` 속성도 포함된 경우 이 속성은 결과 DTO에서 자동으로 제거될 수 있습니다. 이러한 동작을 사용하려면 `화이트리스트를 true로` 설정하세요.

```
app.useGlobalPipes(
  new ValidationPipe({
```

```
    화이트리스트: true,
  }),
);
```

true로 설정하면 화이트리스트에 없는 프로퍼티(유효성 검사 클래스에 데코레이터가 없는 프로퍼티)가 자동으로 제거됩니다.

또는 화이트리스트에 없는 속성이 있는 경우 요청 처리를 중지하고 사용자에게 오류 응답을 반환할 수 있습니다. 이 기능을 사용하려면 `화이트리스트를 true로` 설정하는 것과 함께 `forbidNonWhitelisted` 옵션 속성을 `true로` 설정하세요.

페이로드 개체 변환

네트워크를 통해 들어오는 페이로드는 일반 JavaScript 객체입니다. `유효성 검사 파이프는` 페이로드를 DTO 클래스에 따라 입력된 객체로 자동 변환할 수 있습니다. 자동 변환을 활성화하려면 `transform을 true로` 설정합니다. 이 작업은 메서드 수준에서 수행할 수 있습니다:

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

이 동작을 전역적으로 사용하려면 전역 파이프에서 옵션을 설정합니다:

```
app.useGlobalPipes(
  new ValidationPipe({
    transform: true,
  }),
);
```

자동 변환 옵션을 활성화하면 `ValidationPipe는` 기본 유형 변환도 수행합니다. 다음 예제에서 `findOne()` 메서드는 추출된 `id` 경로 매개변수를 나타내는 하나의 인수를 받습니다:

```
@Get(':id')
findOne(@Param('id') id: number) {
  console.log(typeof id === 'number'); // true
  반환 '이 작업은 사용자를 반환합니다';
}
```

기본적으로 모든 경로 매개변수와 쿼리 매개변수는 네트워크를 통해 문자열로 전달됩니다. 위의 예에서는 메서드 서명에서 ID 유형을 숫자로 지정했습니다. 따라서

ValidationPipe는 문자열 식별자를 숫자로 자동 변환하려고 시도합니다. 명시적 변환

위 섹션에서는 `ValidationPipe`가 쿼리와 경로를 암시적으로 변환하는 방법을 보여드렸습니다.
매개변수를 사용할 수 있습니다. 그러나 이 기능을 사용하려면 자동 변환을 사용하도록 설정해야 합니다.

또는 (자동 변환을 비활성화한 상태에서) `ParseIntPipe` 또는 `ParseBoolPipe`를 사용하여 명시적으로 값을 캐스팅할 수 있습니다(앞서 언급했듯이 모든 경로 매개변수와 쿼리 매개변수는 기본적으로 네트워크를 통해 `문자열`로 제공되므로 ParseStringPipe는 필요하지 않음).

```
@Get(':id')
findOne(
  Param('id', ParseIntPipe) id: 숫자,
  @Query('sort', ParseBoolPipe) sort: 부울,
) {
  console.log(typeof id === '숫자'); // 참
  console.log(typeof sort === '부울'); // 참 반환 '이 작업
  은 사용자를 반환합니다';
}
```

> 정보 힌트 ParseIntPipe와 ParseBoolPipe는 `@nestjs/common`에서 내보냅니다.
> 패키지입니다.

매핑된 유형

CRUD(만들기/읽기/업데이트/삭제)와 같은 기능을 구축할 때 기본 엔티티 유형에서 변형을 구성하는 것이 유용할 때가 많습니다. Nest는 유형 변환을 수행하는 여러 유틸리티 함수를 제공하여 이 작업을 더욱 편리하게 만듭니다.

> 경고 애플리케이션에서 `@nestjs/swagger` 패키지를 사용하는 경우 이 장에서 매핑된 유형에 대한 자세한 내용을 참조하세요. 마찬가지로 `@nestjs/graphql` 패키지를 사용하는 경우 이 장을 참조하세요. 두 패키지 모두 타입에 크게 의존하므로 사용하려면 다른 임포트가 필요합니다. 따라서 앱 유형에 따라 적절한 `@nestjs/mapped-types` 대신 `@nestjs/swagger` 또는 `@nestjs/graphql`을 사용하는 경우 문서화되지 않은 다양한 부작용이 발생할 수 있습니다.

입력 유효성 검사 유형(DTO라고도 함)을 구축할 때 동일한 유형에 대해 생성 및 업데이트 변형을 구축하는 것이 유용한 경우가 많습니다. 예를 들어, 만들기 변형은 모든 필드를 필수로 설정하고 업데이트 변형은 모든 필드를 선

택 사항으로 설정할 수 있습니다.

Nest는 이 작업을 더 쉽게 수행하고 상용구를 최소화하기 위해 `PartialType()` 유틸리티 함수를 제공합니다.

`PartialType()` 함수는 입력 유형의 모든 속성이 선택 사항으로 설정된 유형(클래스)을 반환합니다. 예를 들어 다음과 같은 create 유형이 있다고 가정해 보겠습니다:

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

기본적으로 이러한 필드는 모두 필수입니다. 필드는 동일하지만 각 필드가 선택 사항인 유형을 만들려면 클래스 참조(`CreateCatDto`)를 인수로 전달하는 `PartialType()`을 사용합니다:

```
export class UpdateCatDto extends PartialType(CreateCatDto) {} {}
```

> 정보 힌트 `PartialType()` 함수는 `@nestjs/mapped-types` 패키지에서 가져온 것입니다.

`PickType()` 함수는 입력 유형에서 속성 집합을 선택하여 새 유형(클래스)을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

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

이 클래스에서 `PickType()` 유틸리티 함수를 사용하여 프로퍼티 집합을 선택할 수 있습니다:

```
export class UpdateCatAgeDto extends PickType(CreateCatDto, ['age'] as
const) {} {}
```

> 정보 힌트 `PickType()` 함수는 `@nestjs/mapped-types` 패키지에서 가져온 것입니다.

`OmitType()` 함수는 입력 유형에서 모든 속성을 선택한 다음 특정 키 집합을 제거하여 유형을 구성합니다. 예를 들어 다음과 같은 유형으로 시작한다고 가정해 보겠습니다:

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

아래와 같이 이름을 제외한 모든 프로퍼티를 가진 파생 유형을 생성할 수 있습니다. 이 구조체에서 OmitType의 두 번째 인수는 프로퍼티 이름의 배열입니다.

```

```

```
export class UpdateCatDto extends OmitType(CreateCatDto, ['name'] as
const) {} {}
```

> 정보 힌트 `OmitType()` 함수는 `@nestjs/mapped-types` 패키지에서 가져온 것입니다.

`IntersectionType()` 함수는 두 유형을 하나의 새로운 유형(클래스)으로 결합합니다. 예를 들어 다음과 같은 두 가지 유형으로 시작한다고 가정해 보겠습니다:

```
export 클래스 CreateCatDto {
  이름: 문자열;
  품종: 문자열;
}

내보내기 클래스 AdditionalCatInfo {
  색상: 문자열;
}
```

두 유형의 모든 속성을 결합한 새로운 유형을 생성할 수 있습니다.

```
내보내기 클래스 UpdateCatDto extends IntersectionType(
  CreateCatDto,
  AdditionalCatInfo,
) {}
```

> 정보 힌트 `IntersectionType()` 함수는 `@nestjs/mapped-types에서` 가져온 것입니다.
> 패키지입니다.

유형 매핑 유틸리티 함수는 컴포지션이 가능합니다. 예를 들어 다음은 `이름을` 제외한 `CreateCatDto` 유형의 모든 속성을 가진 유형(클래스)을 생성하며, 해당 속성은 선택 사항으로 설정됩니다:

```
내보내기 클래스 UpdateCatDto extends PartialType(
  OmitType(CreateCatDto, ['name] as const),
) {}
```

배열 구문 분석 및 유효성 검사

TypeScript는 제네릭이나 인터페이스에 대한 메타데이터를 저장하지 않으므로 DTO에서 제네릭이나 인터페이스

를 사용할 때 ValidationPipe가 들어오는 데이터의 유효성을 제대로 검사하지 못할 수 있습니다. 예를 들어, 다음 코드에서는 `createUserDtos`의 유효성이 올바르게 검사되지 않습니다:

```
@Post()
createBulk(@Body() createUserDtos: CreateUserDto[]) {
```

```
    '이 작업은 새 사용자를 추가합니다'를 반환합니다;
}
```

배열의 유효성을 검사하려면 배열을 감싸는 프로퍼티가 포함된 전용 클래스를 만들거나 `ParseArrayPipe`.

```
@Post()
createBulk(
  Body(new ParseArrayPipe({ items: CreateUserDto }))
  createUserDtos: CreateUserDto[],
) {
  '이 작업은 새 사용자를 추가합니다'를 반환합니다;
}
```

또한 쿼리 매개변수를 구문 분석할 때 `ParseArrayPipe가` 유용하게 사용될 수 있습니다. 예를 들어 쿼리 매개변수로 전달된 식별자를 기반으로 사용자를 반환하는 `findByIds()` 메서드입니다.

```
@Get()
findByIds(
  쿼리('ids', new ParseArrayPipe({ 항목: 숫자, 구분자: ',' })) ids: 숫자[],
) {
  반환 '이 액션은 아이디별로 사용자를 반환합니다';
}
```

이 구조는 다음과 같이 HTTP `GET` 요청에서 들어오는 쿼리 매개변수의 유효성을 검사합니다:

```
GET /?ids=1,2,3
```

웹소켓 및 마이크로서비스

이 장에서는 HTTP 스타일 애플리케이션(예: Express 또는 Fastify)을 사용한 예제를 보여드리지만, `ValidationPipe는` 사용되는 전송 방법에 관계없이 웹소켓 및 마이크로서비스에 대해 동일하게 작동합니다.

자세히 알아보기

여기에서 `클래스 유효성` 검사기 패키지에서 제공하는 사용자 지정 유효성 검사기, 오류 메시지 및 사용 가능한 데코레이터에 대해 자세히 알아보세요.

캐싱

캐싱은 앱의 성능을 향상시키는 데 도움이 되는 훌륭하고 간단한 기술입니다. 캐싱은 고성능 데이터 액세스를 제공하는 임시 데이터 저장소 역할을 합니다.

설치

먼저 필요한 패키지를 설치합니다:

```
npm 설치 @nestjs/cache-manager 캐시-관리자
```

> 경고 캐시 관리자 버전 4는 TTL(Time-To-Live)에 초를 사용합니다. 현재 버전의 캐시 관리자(v5)는 대신 밀리초를 사용하도록 전환되었습니다. NestJS는 값을 변환하지 않고 단순히 사용자가 제공한 TTL을 라이브러리에 전달합니다. 다시 말해
>
> - 캐시 관리자 v4를 사용하는 경우, ttl을 초 단위로 입력합니다.
> - 캐시 관리자 v5를 사용하는 경우, ttl을 밀리초 단위로 입력합니다.
>
> NestJS는 캐시 관리자 버전 4를 대상으로 출시되었으므로 문서에서는 초를 기준으로 합니다.

인메모리 캐시

Nest는 다양한 캐시 스토리지 제공업체를 위한 통합 API를 제공합니다. 기본 제공되는 것은 인메모리 데이터 저장소입니다. 그러나 Redis와 같은 보다 포괄적인 솔루션으로 쉽게 전환할 수 있습니다.

캐싱을 활성화하려면 캐시 모듈을 임포트하고 등록() 메서드를 호출합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/cache-manager'에서 { CacheModule }을 임포트하고,
'./app.controller'에서 { AppController }를 임포트합니다;

모듈({
  임포트합니다: [CacheModule.register()], 컨트롤러:
  [AppController],
})
내보내기 클래스 AppModule {}
```

캐시 스토어와 상호 작용하기

캐시 관리자 인스턴스와 상호 작용하려면 다음과 같이 `캐시 관리자` 토큰을 사용하여 클래스에 주입하세요:

```
생성자(@Inject(CACHE_MANAGER) private cacheManager: Cache) {}
```

> 정보 힌트 `캐시` 클래스는 `캐시 매니저에서` 가져오고, `캐시_매니저` 토큰은 `@nestjs/cache-manager` 패키지에서 가져옵니다.

`캐시` 인스턴스(`캐시 관리자` 패키지의 캐시 인스턴스)의 `get` 메서드는 캐시에서 항목을 검색하는 데 사용됩니다. 항목이 캐시에 존재하지 않으면 null이 반환됩니다.

```
const value = await this.cacheManager.get('key');
```

캐시에 항목을 추가하려면 `set` 메서드를 사용합니다:

```
await this.cacheManager.set('key', 'value');
```

캐시의 기본 만료 시간은 5초입니다.

다음과 같이 이 특정 키에 대한 TTL(초 단위의 만료 시간)을 수동으로 지정할 수 있습니다:

```
await this.cacheManager.set('key', 'value', 1000);
```

캐시 만료를 비활성화하려면 `ttl` 구성 속성을 `0으로` 설정합니다:

```
await this.cacheManager.set('key', 'value', 0);
```

캐시에서 항목을 제거하려면 `del` 메서드를 사용합니다:

```
await this.cacheManager.del('key');
```

전체 캐시를 지우려면 `재설정` 방법을 사용하세요:

```
await this.cacheManager.reset();
```

응답 자동 캐싱

> 경고 GraphQL 애플리케이션에서 인터셉터는 각 필드 리졸버에 대해 별도로 실행됩니다. 따라서 인터셉터를 사용하여 응답을 캐시하는 CacheModule이 제대로 작동하지 않습니다.

응답 자동 캐싱을 사용하려면 데이터를 캐시하려는 위치에 `CacheInterceptor`를 연결하기만 하면 됩니다.

```
컨트롤러() @사용인터셉터(캐시인터셉터)
```

```
내보내기 클래스 AppController {
  @Get()
  findAll(): string[] {
    return [];
  }
}
```

> 경고 경고 `GET` 엔드포인트만 캐시됩니다. 또한 네이티브 응답 객체(`@Res()`)를 삽입하는 HTTP 서버 경로는 캐시 인터셉터를 사용할 수 없습니다. 자세한 내용은 응답 매핑을 참조하세요.

필요한 상용구의 양을 줄이려면 모든 엔드포인트에 `캐시인터셉터를` 전역적으로 바인딩하면 됩니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/cache-manager'에서 { CacheModule, CacheInterceptor }를 임포트하고,
'./app.controller'에서 { AppController }를 임포트합니다;
'@nestjs/core'에서 { APP_INTERCEPTOR }를 임포트합니다;

모듈({
  임포트합니다: [CacheModule.register()],
  컨트롤러: [AppController], providers: [
    {
      제공: APP_INTERCEPTOR,
      useClass: 캐시인터셉터,
    },
  ],
})
내보내기 클래스 AppModule {}
```

캐싱 사용자 지정

모든 캐시된 데이터에는 고유한 만료 시간(TTL)이 있습니다. 기본값을 사용자 정의하려면 옵션 객체를 `register()` 메서드에 전달합니다.

```
CacheModule.register({
  ttl: 5, // 초
  최대: 10, // 캐시 내 최대 항목 수
});
```

모듈을 전 세계적으로 사용

다른 모듈에서 `캐시 모듈을` 사용하려면 모든 Nest 모듈의 표준처럼 `캐시` 모듈을 임포트해야 합니다. 또는 아래와 같이 옵션 객체의 `isGlobal` 속성을 `true로` 설정하여 전역 모듈로 선언할 수도 있습니다. 이 경우 루트 모듈(예: `AppModule`)에서 CacheModule을 로드한 후에는 다른 모듈에서 임포트할 필요가 없습니다.

```
CacheModule.register({
  isGlobal: true,
});
```

글로벌 캐시 재정의

글로벌 캐시가 활성화되어 있는 동안 캐시 항목은 경로 경로에 따라 자동 생성되는 캐시키 아래에 저장됩니다. 메소드별로 특정 캐시 설정(@CacheKey() 및 @CacheTTL())을 재정의할 수 있으므로 개별 컨트롤러 메소드에 대한 맞춤형 캐시 전략을 사용할 수 있습니다. 이는 서로 다른 캐시 저장소를 사용할 때 가장 적절할 수 있습니다 .

```
컨트롤러()
export class AppController {
  @CacheKey('custom_key')
  @CacheTTL(20)
  findAll(): string[] {
    return [];
  }
}
```

> 정보 힌트 @CacheKey() 및 @CacheTTL() 데코레이터는 nestjs/cache-manager 패키지.

캐시키() 데코레이터는 해당 @CacheTTL() 데코레이터와 함께 또는 없이 사용할 수 있으며, 그 반대의 경우도 마찬가지입니다. 캐시키() 데코레이터만 재정의하거나 캐시TTL() 데코레이터만 재정의하도록 선택할 수 있습니다. 데코레이터로 재정의하지 않은 설정은 전역에 등록된 기본값을 사용합니다(캐싱 사용자 정의 참조).

웹소켓 및 마이크로서비스

사용 중인 전송 방법에 관계없이 마이크로서비스의 패턴뿐만 아니라 웹소켓 구독자에게도 캐시인터셉터를 적용할 수 있습니다.

```
@@파일명()
@CacheKey('events')
사용 인터셉터(캐시인터셉터) @SubscribeMessage('이벤트')
handleEvent(client: Client, data: string[]): Observable<string[]> {
  return [];
}
@@스위치
@CacheKey('events')
UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client, data) {
  반환 [];
}
```

그러나 캐시된 데이터를 나중에 저장하고 검색하는 데 사용되는 키를 지정하려면 추가 `@CacheKey()` 데코레이터가 필요합니다. 또한 모든 것을 캐시해서는 안 된다는 점에 유의하세요. 단순히 데이터를 쿼리하는 것이 아니라 일부 비즈니스 작업을 수행하는 액션은 절대 캐시해서는 안 됩니다.

또한 `@CacheTTL()` 데코레이터를 사용하여 캐시 만료 시간(TTL)을 지정할 수 있으며, 이 경우 글로벌 기본 TTL 값이 재정의됩니다.

```
@@파일명()
@CacheTTL(10)
사용 인터셉터(캐시인터셉터)

@SubscribeMessage('이벤트')
handleEvent(client: Client, data: string[]): Observable<string[]> {
  return [];
}
@@switch
@CacheTTL(10)
UseInterceptors(CacheInterceptor)
@SubscribeMessage('events')
handleEvent(client, data) {
  반환 [];
}
```

> 정보 힌트 `@CacheTTL()` 데코레이터는 대응하는 `@CacheKey()`와 함께 또는 없이 사용할 수 있습니다.

데코레이터.

추적 조정

기본적으로 Nest는 요청 URL(HTTP 앱의 경우) 또는 캐시 키(웹소켓 및 마이크로서비스 앱의 경우 `@CacheKey()` 데코레이터를 통해 설정됨)를 사용하여 캐시 레코드를 엔드포인트와 연결합니다.
그럼에도 불구하고 때때로 다른 요소(예: `프로필` 엔드포인트를 올바르게 식별하기 위한 `권한 부여)를` 사용하는 등 다양한 요소에 따라 추적을 설정하고 싶을 수 있습니다.

이를 위해 `CacheInterceptor의` 서브클래스를 생성하고 `trackBy()` 메서드를 재정의합니다.
메서드를 사용합니다.

```
@Injectable()
클래스 HttpCacheInterceptor extends CacheInterceptor {
  trackBy(context: ExecutionContext): string | undefined {
    '키'를 반환합니다;
  }
}
```

다른 매장

이 서비스는 내부적으로 캐시 매니저를 활용합니다. 캐시 관리자 패키지는 Redis 스토어와 같은 다양한 유용한 스토어를 지원합니다. 지원되는 저장소의 전체 목록은 여기에서 확인할 수 있습니다. 설정하려면

에서 해당 옵션과 함께 패키지를 등록() 함수에 전달하기만 하면 됩니다.

메서드를 사용합니다.

```
'redis'에서 { RedisClientOptions } 유형을 가져옵니다;
'cache-manager-redis-store'에서 redisStore로 * 임포트;
'@nestjs/common'에서 { Module }을 임포트합니다;
'@nestjs/cache-manager'에서 { CacheModule }을 임포트하고,
'./app.controller'에서 { AppController }를 임포트합니다;

모듈({ import: [
    CacheModule.register<RedisClientOptions>({
      store: redisStore,

      // 스토어별 구성: host: 'localhost',
      포트: 6379,
    }),
  ],
  컨트롤러: [앱 컨트롤러],
})
내보내기 클래스 AppModule {}
```

> 경고 경고 캐시-매니저-레디스 스토어는 레디스 v4를 지원하지 않습니다. ClientOpts 인터페이스가 존재하고 올바르게 작동하려면 최신 redis 3.x.x 주 릴리스를 설치해야 합니다. 이 업그레이드의 진행 상황을 추적하려면 이 문제를 참조하세요.

비동기 구성

컴파일 시 정적으로 전달하는 대신 모듈 옵션을 비동기적으로 전달하고 싶을 수 있습니다. 이 경우 비동기 구성을 처리하는 여러 가지 방법을 제공하는 registerAsync() 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
CacheModule.registerAsync({
  useFactory: () => ({
    ttl: 5,
  }),
});
```

저희 팩토리는 다른 모든 비동기 모듈 팩토리처럼 작동합니다(비동기일 수 있고 인젝트를 통해 종속성을 주입할 수

있습니다).

```
CacheModule.registerAsync({
  import: [ConfigModule],
  사용 팩토리: 비동기 (config서비스: 구성 서비스) => ({
```

```
    ttl: configService.get('CACHE_TTL'),
  }),
  주입합니다: [구성 서비스],
});
```

또는 `useClass` 메서드를 사용할 수 있습니다:

```
CacheModule.registerAsync({
  useClass: CacheConfigService,
});
```

위의 구조는 `CacheModule` 내부에 `CacheConfigService를` 인스턴스화하고 이를 사용해 옵션 객체를 가져옵니다. CacheConfigService는 구성 옵션을 제공하기 위해 `CacheOptionsFactory` 인터페이스를 구현해야 합니다:

```
@Injectable()
CacheConfigService 클래스는 CacheOptionsFactory { createCacheOptions() 를 구현합
  니다: CacheModuleOptions {
    반환 { ttl:
      5,
    };
  }
}
```

다른 모듈에서 가져온 기존 구성 공급자를 사용하려면 `사용기존` 구문:

```
CacheModule.registerAsync({
  import: [ConfigModule],
  useExisting: ConfigService,
});
```

한 가지 중요한 차이점을 제외하고는 `사용클래스와` 동일하게 작동하지만, 캐시모듈은 가져온 모듈을 조회하여 자체적으로 인스턴스화하지 않고 이미 생성된 `컨피그서비스를` 재사용합니다.

> 정보 힌트 `CacheModule#register` 및 CacheModule#registerAsync와 CacheOptionsFactory에는 스토어별 구성 옵션의 범위를 좁힐 수 있는 선택적 제네릭(유형 인수)이 있으므로 유형 안전합니다.

예

작동하는 예제는 여기에서 확인할 수 있습니다.

직렬화

직렬화는 네트워크 응답에서 객체가 반환되기 전에 발생하는 프로세스입니다. 클라이언트에 반환할 데이터를 변환하고 살균하기 위한 규칙을 제공하기에 적합한 곳입니다. 예를 들어 비밀번호와 같은 민감한 데이터는 항상 응답에서 제외해야 합니다. 또는 엔티티 속성의 하위 집합만 전송하는 등 특정 속성에 추가 변환이 필요할 수도 있습니다. 이러한 변환을 수동으로 수행하면 지루하고 오류가 발생하기 쉬우며 모든 경우를 처리했는지 확실하지 않을 수 있습니다.

개요

Nest는 이러한 작업을 간단한 방법으로 수행할 수 있도록 지원하는 내장 기능을 제공합니다. `ClassSerializerInterceptor` 인터셉터는 강력한 클래스 트랜스포머 패키지를 사용하여 선언적이고 확장 가능한 객체 변환 방법을 제공합니다. 이 인터셉터가 수행하는 기본 작업은 메서드 핸들러가 반환한 값을 가져와 클래스 트랜스포머의 `instanceToPlain()` 함수를 적용하는 것입니다. 이렇게 함으로써 아래 설명된 대로 엔티티/DTO 클래스에 `클래스 트랜스포머` 데코레이터로 표현된 규칙을 적용할 수 있습니다.

> 정보 힌트 직렬화는 StreamableFile 응답에는 적용되지 않습니다.

속성 제외

사용자 엔티티에서 `비밀번호` 속성을 자동으로 제외하려고 한다고 가정해 보겠습니다. 다음과 같이 엔티티에 주석을 추가합니다:

```
import { Exclude } from 'class-transformer';

export class UserEntity {
  id: 숫자; 이름: 문자
  열; 성: 문자열;

  @Exclude() 비밀번호:
  문자열;

  constructor(partial: Partial<UserEntity>) {
    Object.assign(this, partial);
  }
}
```

이제 이 클래스의 인스턴스를 반환하는 메서드 핸들러가 있는 컨트롤러를 생각해 보겠습니다.

```
사용 인터셉터(클래스시리얼라이저인터셉터) @Get()
findOne(): UserEntity {
  return new UserEntity({
    id: 1,
```

```
    이름: '카밀', 성 '미슬리비츠
    ', 비밀번호: '비밀번호',
  });
}
```

> 경고 클래스의 인스턴스를 반환해야 한다는 점에 유의하세요. 예를 들어 `{{ '{' }} user: new UserEntity() {{ '}' }}`와 같이 반환하면 객체가 제대로 직렬화되지 않습니다.

> 정보 힌트 ClassSerializerInterceptor는 `@nestjs/common`에서 가져옵니다.

이 엔드포인트가 요청되면 클라이언트는 다음과 같은 응답을 받습니다:

```
{
  "id": 1,
  "이름": "카밀", "성": "미슬리
  비츠"
}
```

인터셉터는 애플리케이션 전체에 적용될 수 있습니다(여기서 설명한 대로). 인터셉터와 엔티티 클래스 선언을 조합하면 `UserEntity를` 반환하는 모든 메서드에서 `비밀번호` 속성을 제거할 수 있습니다. 이를 통해 이 비즈니스 규칙을 중앙 집중식으로 시행할 수 있습니다.

속성 노출

`노출()` 데코레이터를 사용하여 아래와 같이 프로퍼티의 별칭 이름을 제공하거나 프로퍼티 값을 계산하는 함수(게터 함수와 유사)를 실행할 수 있습니다.

```
@Expose()
get fullName(): 문자열 {
  반환 `${이름} ${이름}`을 반환합니다;
}
```

변환

`Transform()` 데코레이터를 사용하여 추가 데이터 변환을 수행할 수 있습니다. 예를 들어 다음 구문은 전체 객체를 반환하는 대신 `역할 엔티티의` 이름 속성을 반환합니다.

```
@Transform(({ 값 }) => value.name) 역할:
RoleEntity;
```

패스 옵션

변환 함수의 기본 동작을 수정하고 싶을 수 있습니다. 기본 설정을 재정의하려면 옵션 객체에 @SerializeOptions() 데코레이터를 사용하여 해당 설정을 전달합니다.

```
@SerializeOptions({ 제외 접두사:
  ['_'],
})
@Get()
findOne(): UserEntity { 반환
  새 UserEntity();
}
```

정보 힌트 @SerializeOptions() 데코레이터는 @nestjs/common에서 가져온 것입니다.

SerializeOptions()를 통해 전달된 옵션은 기본 인스턴스ToPlain() 함수의 두 번째 인수로 전달됩니다. 이 예제에서는 _ 접두사로 시작하는 모든 프로퍼티를 자동으로 제외합니다.

예

작동 예제는 여기에서 확인할 수 있습

니다. 웹소켓 및 마이크로서비스

이 장에서는 HTTP 스타일 애플리케이션(예: Express 또는 Fastify)을 사용하는 예제를 보여 주지만, 이 장에서는 ClassSerializerInterceptor는 사용되는 전송 방법에 관계없이 웹소켓과 마이크로서비스에서 동일하게 작동합니다.

자세히 알아보기

클래스 트랜스포머 패키지에서 제공하는 데코레이터 및 옵션에 대한 자세한 내용은 여기를 참조하세요.

버전 관리

> 정보 힌트 이 장은 HTTP 기반 애플리케이션에만 해당됩니다.

버전 관리를 사용하면 동일한 애플리케이션 내에서 서로 다른 버전의 컨트롤러 또는 개별 경로를 실행할 수 있습니다. 애플리케이션은 매우 자주 변경되며, 이전 버전의 애플리케이션을 계속 지원하면서 변경해야 하는 경우가 드물지 않습니다.

지원되는 버전 관리 유형은 4가지입니다:

`URI 버전 관리`	버전은 요청의 URI 내에서 전달됩니다(기본값) 사용자 지정 요청 헤
`헤더 버전 관리`	더에 버전이 지정됩니다.
`미디어 유형 버전 관리`	요청의 `Accept` 헤더에는 버전이 지정됩니다.
`사용자 지정 버전 관리`	요청의 모든 측면을 사용하여 버전을 지정할 수 있습니다. 해당 버전을 추출하기 위한 사용자 정의 함수가 제공됩니다.

URI 버전 관리 유형

URI 버전 관리에서는 다음과 같이 요청의 URI 내에 전달된 버전을 사용합니다.

`https://example.com/v1/route` 및 https://example.com/v2/route.

> 경고 URI 버전 지정 시 버전은 글로벌 경로 접두사(있는 경우) 뒤와 컨트롤러 또는 경로 경로 앞에 자동으로 URI에 추가됩니다.

애플리케이션에 대해 URI 버전 관리를 사용 설정하려면 다음과 같이 하세요:

```
@@파일명(메인)
const app = await NestFactory.create(AppModule);
// 또는 "app.enableVersioning()"
app.enableVersioning({
  유형입니다: 버전 관리 유형.URI,
});
await app.listen(3000);
```

> 경고 URI의 버전은 기본적으로 자동으로 접두사 앞에 `v가` 붙지만 `접두사` 키를 원하는 접두사로 설정하거나 비활성화하려는 경우 `false로` 설정하여 접두사 값을 구성할 수 있습니다.

> 정보 힌트 `유형` 속성에 사용할 수 있는 `VersioningType` 열거형은 `@nestjs/common` 패키지에서 가져옵니다.

헤더 버전 관리 유형

헤더 버전 관리에서는 사용자 지정 사용자 지정 요청 헤더를 사용하여 헤더 값이 요청에 사용할 버전이 되는 버전을 지정합니다.

헤더 버전 관리를 위한 HTTP 요청 예시:

애플리케이션에 헤더 버전 관리를 사용 설정하려면 다음과 같이 하세요:

```
@@파일명(메인)
const app = await NestFactory.create(AppModule);
app.enableVersioning({
  유형: VersioningType.HEADER, 헤더
  : '사용자 지정 헤더',
});
await app.listen(3000);
```

`헤더` 속성은 요청의 버전을 포함할 헤더의 이름이어야 합니다.

> 정보 힌트 `유형` 속성에 사용할 수 있는 `VersioningType` 열거형은 `@nestjs/common` 패키지에서 가져옵니다.

미디어 유형 버전 관리 유형

미디어 유형 버전 지정은 요청의 `Accept` 헤더를 사용하여 버전을 지정합니다.

`Accept` 헤더 내에서 버전은 세미콜론(`;`)으로 미디어 유형과 구분됩니다. 그런 다음 요청에 사용할 버전을 나타내는 키-값 쌍을 포함해야 합니다(예: `Accept: application/json;v=2`). 키와 구분 기호를 포함하도록 구성할 버전을 결정할 때 키는 접두사로 더 많이 취급됩니다.

애플리케이션에 미디어 유형 버전 관리를 사용 설정하려면 다음을 수행합니다:

```
@@파일명(메인)
const app = await NestFactory.create(AppModule);
app.enableVersioning({
  유형: 버전관리유형.MEDIA_TYPE, 키:
  'v=',
});
await app.listen(3000);
```

`키` 속성은 버전을 포함하는 키-값 쌍의 키와 구분 기호여야 합니다. `Accept: application/json;v=2` 예제에

서 `키` 속성은 `v=로` 설정됩니다.

> 정보 힌트 `유형` 속성에 사용할 수 있는 `VersioningType` 열거형은 `@nestjs/common` 패키지에서 가져옵니다.

사용자 지정 버전 관리 유형

사용자 정의 버전 관리에서는 요청의 모든 측면을 사용하여 버전(또는 버전)을 지정합니다. 들어오는 요청은 문자열 또는 문자열 배열을 반환하는 추출기 함수를 사용하여 분석됩니다.

요청자가 여러 버전을 제공한 경우 추출 함수는 가장 큰/가장 높은 버전에서 가장 작은/가장 낮은 버전 순으로 정렬된 문자열 배열을 반환할 수 있습니다. 버전은 가장 높은 버전에서 가장 낮은 버전 순으로 경로에 매칭됩니다.

추출기에서 빈 문자열 또는 배열이 반환되면 일치하는 경로가 없으며 404가 반환됩니다. 예를 들어, 들어오는 요청이 버전 1, 2, 3을 지원한다고 지정하는 경우 추출기는 반드시 `[3, 2, 1]`. 이렇게 하면 가능한 가장 높은 경로 버전이 먼저 선택됩니다.

버전 `[3, 2, 1]`이 추출되었지만 경로가 버전 2와 1에 대해서만 존재하는 경우 버전 2와 일치하는 경로가 선택됩니다(버전 3은 자동으로 무시됨).

> 경고 주의 추출기에서 반환된 배열을 기준으로 가장 일치하는 버전을 선택하는 것은 설계상의 제한으로 인해 Express 어댑터에서 안정적으로 작동하지 않습니다. 단일 버전(문자열 또는 요소 1개로 구성된 배열)은 Express에서 정상적으로 작동합니다. Fastify는 가장 일치하는 버전 선택과 단일 버전 선택을 모두 올바르게 지원합니다.

애플리케이션에 사용자 지정 버전 관리를 사용하려면 다음과 같이 추출기 함수를 만들어 애플리케이션에 전달합니다:

```
@@파일명(메인)
// 사용자 정의 헤더에서 버전 목록을 가져와 정렬된 배열로 변환하는 추출기 예제입니다.
// 이 예에서는 Fastify를 사용하지만 Express 요청도 비슷한 방식으로 처리할 수 있습니다.
const extractor = (request: FastifyRequest): string | string[] =>
  [request.headers['custom-versioning-field'] ?? '']
     .flatMap(v => v.split(','))
     .filter(v => !!v)
     .sort()
     .reverse()

const app = await NestFactory.create(AppModule);
app.enableVersioning({
  유형: 버전 관리 유형, 추출기,
});
await app.listen(3000);
```

사용법

버전 관리를 사용하면 컨트롤러와 개별 경로를 버전 관리할 수 있으며 특정 리소스가 버전 관리를 거부할 수 있는 방법도 제공합니다. 버전 관리의 사용법은 애플리케이션에서 사용하는 버전 관리 유형에 관계없이 동일합니다 .

> 경고 애플리케이션에 버전 관리가 활성화되어 있지만 컨트롤러 또는 경로가 버전을 지정하지 않은 경우 해당 컨트롤러/경로에 대한 모든 요청은 `404` 응답 상태로 반환됩니다.

> 마찬가지로 해당 컨트롤러나 경로가 없는 버전이 포함된 요청이 수신되면 `404` 응답 상태도 반환됩니다.

컨트롤러 버전

컨트롤러에 버전을 적용하여 컨트롤러 내의 모든 경로에 대한 버전을 설정할 수 있습니다. 컨트롤러

에 버전을 추가하려면 다음과 같이 하세요:

```
@@파일명(cats.controller)
@Controller({
  버전: '1',
})
내보내기 클래스 CatsControllerV1 {
  @Get('cats')
  findAll(): 문자열 {
    반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다;
  }
}
@@스위치 @Controller({
  버전: '1',
})
내보내기 클래스 CatsControllerV1 {
  @Get('cats')
  findAll() {
    반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다;
  }
}
```

경로 버전

버전은 개별 경로에 적용할 수 있습니다. 이 버전은 컨트롤러 버전과 같이 경로에 영향을 미치는 다른 모든 버전보다 우선합니다.

개별 경로에 버전을 추가하려면 다음과 같이 하세요:

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

컨트롤러()
내보내기 클래스 CatsController {
  @Version('1')
  @Get('cats')
  findAllV1(): string {
    반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다;
  }

  버전('2')
  @Get('cats')
```

```
    findAllV2(): 문자열 {
      반환 '이 작업은 버전 2의 모든 고양이를 반환합니다;
    }
  }
  @@switch
  '@nestjs/common'에서 { Controller, Get, Version }을 가져옵니다;

  컨트롤러()
  내보내기 클래스 CatsController {
    @Version('1')
    @Get('cats')
    findAllV1() {
      반환 '이 작업은 버전 1에 대한 모든 고양이를 반환합니다;
    }

    버전('2')
    @Get('cats')
    findAllV2() {
      반환 '이 작업은 버전 2의 모든 고양이를 반환합니다;
    }
  }
```

여러 버전

컨트롤러 또는 경로에 여러 버전을 적용할 수 있습니다. 여러 버전을 사용하려면 버전을 배열로 설정합니다.

여러 버전을 추가하려면 다음과 같이 하세요:

```
@@파일명(cats.controller)
@Controller({
  버전입니다: ['1', '2'],
})
내보내기 클래스 CatsController {
  @Get('cats')
  findAll(): 문자열 {
    반환 '이 작업은 버전 1 또는 2에 대한 모든 고양이를 반환합니다;
  }
}
@@스위치 @Controller({
  버전입니다: ['1', '2'],
})
내보내기 클래스 CatsController {
  @Get('cats')
  findAll() {
    반환 '이 작업은 버전 1 또는 2에 대한 모든 고양이를 반환합니다;
  }
}
```

버전 "중립"

일부 컨트롤러나 라우트는 버전에 상관하지 않고 버전에 관계없이 동일한 기능을 사용할 수 있습니다. 이를 위해 버전을 `버전_중립` 기호로 설정할 수 있습니다.

들어오는 요청은 요청에 버전이 전혀 포함되어 있지 않은 경우뿐만 아니라 요청에 전송된 버전과 관계없이 `VERSION_NEUTRAL` 컨트롤러 또는 경로에 매핑됩니다.

> 경고 URI 버전 관리의 경우, `버전_중립` 리소스는 URI에 버전이 존재하지 않습니다.

버전 중립 컨트롤러 또는 경로를 추가하려면 다음과 같이 하세요:

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

컨트롤러({
  버전: 버전_중립,
})
내보내기 클래스 CatsController {
  @Get('cats')
  findAll(): 문자열 {
    반환 '이 작업은 버전에 관계없이 모든 고양이를 반환합니다';
  }
}
@@switch
'@nestjs/common'에서 { Controller, Get, VERSION_NEUTRAL }을 가져옵니다;

컨트롤러({
  버전: 버전_중립,
})
내보내기 클래스 CatsController {
  @Get('cats')
  findAll() {
    반환 '이 작업은 버전에 관계없이 모든 고양이를 반환합니다';
  }
}
```

글로벌 기본 버전

각 컨트롤러/개별 경로에 대한 버전을 제공하지 않거나 특정 버전을 지정하지 않은 모든 컨트롤러/경로에 대해

특정 버전을 기본 버전으로 설정하려는 경우 다음과 같이 `defaultVersion`을 설정할 수 있습니다:

```
@@filename(main)
app.enableVersioning({
  // ...
  defaultVersion: '1'
  // 또는
```

```
    defaultVersion: ['1', '2']
    // 또는
    기본 버전: 버전_중립
  });
```

미들웨어 버전 관리

미들웨어는 버전 관리 메타데이터를 사용하여 특정 경로의 버전에 맞게 미들웨어를 구성할 수도 있습니다. 이렇게 하려면 `MiddlewareConsumer.forRoutes()` 메서드의 매개변수 중 하나로 버전 번호를 제공하면 됩니다:

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

모듈({
  수입: [CatsModule],
})
export class AppModule implements NestModule {
  configure(consumer: MiddlewareConsumer) {
    소비자
      .apply(LoggerMiddleware)
      .forRoutes({ path: 'cats', method: RequestMethod.GET, version: '2'
});
  }
}
```

위의 코드를 사용하면 LoggerMiddleware는 `/cats` 엔드포인트의 버전 '2'에만 적용됩니다.

> 정보 공지 미들웨어는 이 섹션에 설명된 모든 버전 관리 유형에서 작동합니다: `URI`, `헤더`, `미디어 유형` 또는 `사용자 정의`.

작업 예약

작업 예약을 사용하면 임의의 코드(메서드/함수)가 정해진 날짜/시간에, 반복되는 간격으로 또는 지정된 간격 후에 한 번 실행되도록 예약할 수 있습니다. Linux 환경에서는 종종 OS 수준에서 cron과 같은 패키지로 이 작업을 처리합니다. Node.js 앱의 경우 크론과 유사한 기능을 에뮬레이트하는 여러 패키지가 있습니다. Nest는 널리 사용되는 Node.js cron 패키지와 통합되는 `@nestjs/schedule` 패키지를 제공합니다. 이 패키지는 이번 장에서 다루겠습니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

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

작업 스케줄링을 활성화하려면 `스케줄` 모듈을 루트 `앱 모듈로` 가져와서 `forRoot()`를 실행합니다.
정적 메서드를 사용합니다:

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/schedule'에서 { ScheduleModule }을 가져옵니다;

모듈({ import: [
    ScheduleModule.forRoot()
  ],
})
내보내기 클래스 AppModule {}
```

`.forRoot()` 호출은 스케줄러를 초기화하고 앱 내에 존재하는 모든 선언적 크론 작업, 타임아웃 및 간격을 등록합니다. 등록은 `onApplicationBootstrap` 수명 주기 후크가 발생할 때 발생하며, 모든 모듈이 예약된 작업을 로드하고 선언했는지 확인합니다.

선언적 크론 작업

크론 작업은 임의의 함수(메서드 호출)가 자동으로 실행되도록 예약합니다. 크론 작업이 실행될 수 있습니다:

- 지정된 날짜/시간에 한 번.
- 반복 작업은 지정된 간격(예: 한 시간에 한 번, 일주일에 한 번, 5분에 한 번) 내에서 지정된 순간에 실행할 수 있습니다.

다음과 같이 실행할 코드가 포함된 메서드 정의 앞에 `@Cron()` 데코레이터를 사용하여 크론 작업을 선언합니다:

```
'@nestjs/common'에서 { Injectable, Logger }를 가져오고,
'@nestjs/schedule'에서 { Cron }을 가져옵니다;
```

```
@Injectable()
내보내기 클래스 TasksService {
  비공개 읽기 전용 로거 = 새로운 로거(TasksService.name);

  @Cron('45 * * * * *')
  handleCron() {
    this.logger.debug('현재 초가 45일 때 호출됩니다');
  }
}
```

이 예제에서는 현재 초가 `45가 될` 때마다 `handleCron()` 메서드가 호출됩니다. 즉, 메서드는 1분에 한 번, 45초가 될 때마다 실행됩니다.

`Cron()` 데코레이터는 모든 표준 크론 패턴을 지원합니다:

- 별표(예: `*`)
- 범위(예: `1-3,5`)• 걸음
- 수(예: `*/2`)

위의 예에서는 데코레이터에 `45 * * * * * *`를 전달했습니다. 다음 키는 크론 패턴 문자열의 각 위치가 어떻게 해석되는지 보여줍니다:

```
* * * * * *
| | | | | |
| | | | 요일
| | | 개월
| | | 월의 요일
| | 시간
| 분
초(선택 사항)
```

몇 가지 샘플 크론 패턴은 다음과 같습니다:

`* * * * * *`	매초
`45 * * * * *`	매분, 45초마다
`0 10 * * * *`	매시간, 10분이 시작될 때

`0 */30 9-17 * * *` 오전 9시에서 오후 5시 사이 30분마다

`0 30 11 * * 1-5` 월요일부터 금요일 오전 11시 30분

`nestjs/schedule` 패키지는 일반적으로 사용되는 크론 패턴이 포함된 편리한 열거형을 제공합니다. 이 열거형은 다음과 같이 사용할 수 있습니다:

```
'@nestjs/common'에서 { Injectable, Logger }를 임포트하고,
'@nestjs/schedule'에서 { Cron, CronExpression }을 임포트합니다
;

@Injectable()
내보내기 클래스 TasksService {
  비공개 읽기 전용 로거 = 새로운 로거(TasksService.name);

  @Cron(CronExpression.EVERY_30_SECONDS)
  handleCron() {
    this.logger.debug('30초마다 호출');
  }
}
```

이 예제에서는 30초마다 `handleCron()` 메서드가 호출됩니다.

또는 `@Cron()` 데코레이터에 JavaScript `Date` 객체를 제공할 수도 있습니다. 이렇게 하면 지정된 날짜에 작업이 정확히 한 번 실행됩니다.

> 정보 힌트 자바스크립트 날짜 산술을 사용하여 현재 날짜를 기준으로 작업을 예약합니다. 예를 들어 `@Cron(new Date(Date.now() + 10 * 1000))`을 사용하여 앱이 시작된 후 10초 후에 실행되도록 작업을 예약할 수 있습니다.

또한 `@Cron()` 데코레이터에 두 번째 매개변수로 추가 옵션을 제공할 수도 있습니다.

`이름`	크론 작업이 선언된 후 크론 작업에 액세스하고 제어하는 데 유용합니다.
`시간대`	실행할 시간대를 지정합니다. 이렇게 하면 실제 시간이 사용자의 시간대입니다. 시간대가 유효하지 않은 경우 오류가 발생합니다. 사용 가능한 모든 표준 시간대는 모멘트 표준 시간대 웹사이트에서 확인할 수 있습니다.
`utcOffset`	이렇게 하면 `시간대를` 사용하는 대신 표준 시간대의 오프셋을 지정할 수 있습니다. 매개변수.
`disabled`	작업이 전혀 실행되지 않는지 여부를 나타냅니다.

```
'@nestjs/common'에서 { Injectable }을 가져옵니다;
'@nestjs/schedule'에서 { Cron, CronExpression }을 가져옵니다;

@Injectable()
export class NotificationService {
  @Cron('* * 0 * * *', {
    이름: '알림', 시간대: '유럽/파리',
  })
  triggerNotifications() {}
}
```

크론 작업이 선언된 후 크론 작업에 액세스하여 제어하거나, 동적 API를 사용하여 크론 작업(크론 패턴이 런타임에 정의되는 경우)을 동적으로 만들 수 있습니다. API를 통해 선언적 크론 작업에 액세스하려면 다음과 같이 하세요.

는 데코레이터의 두 번째 인수로 선택적 옵션 객체의 `이름` 속성을 전달하여 작업을 이름과 연관시켜야 합니다.

선언적 간격

메서드가 (반복적으로) 지정된 간격으로 실행되어야 한다고 선언하려면 메서드 정의 앞에 `@Interval()` 데코레이터를 붙입니다. 아래와 같이 간격 값을 밀리초 단위의 숫자로 데코레이터에 전달합니다:

```
@Interval(10000)
handleInterval() {
  this.logger.debug('10초마다 호출');
}
```

> 정보 힌트 이 메커니즘은 내부적으로 JavaScript `setInterval()` 함수를 사용합니다. 크론 작업을 활용하여 반복 작업을 예약할 수도 있습니다.

동적 API를 통해 선언 클래스 외부에서 선언 간격을 제어하려면 다음 구문을 사용하여 간격을 이름과 연결하세요:

```
@Interval('알림', 2500)
handleInterval() {}
```

또한 동적 API를 사용하면 런타임에 간격의 속성을 정의하는 동적 간격을 생성하고 이를 나열 및 삭제할 수 있습니다.

선언적 시간 초과

지정된 시간 초과 시 메서드가 (한 번) 실행되도록 선언하려면 메서드 정의 앞에 `@Timeout()` 데코레이터를 붙입니다. 아래와 같이 애플리케이션 시작부터 상대적인 시간 오프셋(밀리초 단위)을 데코레이터에 전달합니다:

```
@Timeout(5000)
handleTimeout() {
  this.logger.debug('5초 후 한 번 호출됨');
}
```

> 정보 힌트 이 메커니즘은 자바스크립트 `setTimeout()` 함수를 내부적으로 사용합니다.

동적 API를 통해 선언 클래스 외부에서 선언적 타임아웃을 제어하려면 다음 구문을 사용하여 타임아웃을 이름과 연결하세요:

```
@Timeout('알림', 2500)
handleTimeout() {}
```

또한 동적 API를 사용하면 런타임에 타임아웃의 속성을 정의하는 동적 타임아웃을 생성하고 이를 나열 및 삭제할 수 있습니다.

동적 일정 모듈 API

`nestjs/schedule` 모듈은 선언적 크론 작업, 시간 초과 및 간격을 관리할 수 있는 동적 API를 제공합니다. 이 API를 사용하면 런타임에 속성이 정의되는 동적 크론 작업, 시간 초과 및 간격을 생성하고 관리할 수도 있습니다.

동적 크론 작업

코드의 어느 곳에서나 이름으로 `CronJob` 인스턴스에 대한 참조를 가져옵니다.

`SchedulerRegistry` API를 사용합니다. 먼저 표준 생성자 주입을 사용하여 `SchedulerRegistry`를 주입합니다:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

> 정보 힌트 `@nestjs/schedule` 패키지에서 `SchedulerRegistry`를 가져옵니다.

그런 다음 다음과 같이 클래스에서 사용합니다. 다음 선언을 사용하여 크론 작업이 생성되었다고 가정합니다:

```
@Cron('* * 8 * * *', {
  name: '알림',
})
triggerNotifications() {}
```

다음을 사용하여 이 작업에 액세스합니다:

```
const job = this.schedulerRegistry.getCronJob('알림'); job.stop();
console.log(job.lastDate());
```

`getCronJob()` 메서드는 명명된 크론 작업을 반환합니다. 반환된 `CronJob` 객체에는 다음과 같은 메서드가 있습니다:

- `stop()` - 실행이 예약된 작업을 중지합니다.

- `start()` - 중지된 작업을 다시 시작합니다.
- `setTime(time: CronTime)` - 작업을 중지하고 새 시간을 설정한 다음 작업을 시작합니다.
- `lastDate()` - 작업이 실행된 마지막 날짜의 문자열 표현을 반환합니다. `nextDates(count: 숫자)` - 예정된 작업 실행 날짜를 나타내는 `모멘트` 객체의 배열(크기 `카운트`)을 반환합니다.

> 정보 힌트 사람이 읽을 수 있는 형태로 렌더링하려면 `모멘트` 객체에서 `toDate()`를 사용하세요.

다음과 같이 SchedulerRegistry#addCronJob 메서드를 사용하여 새 크론 작업을 동적으로 생성합니다:

```
addCronJob(name: 문자열, seconds: 문자열) {
  const job = new CronJob(`${초} * * * * *`, () => { this.logger.warn(`시간 (${
    초}) 동안 ${이름} 작업이 실행됩니다!`);
  });

  this.schedulerRegistry.addCronJob(name, job);
  job.start();

  this.logger.warn(
    매분 ${초}초마다 ${이름} 작업이 추가되었습니다!`,
  );
}
```

이 코드에서는 크론 패키지의 크론 잡 객체를 사용하여 크론 잡을 생성합니다. CronJob 생성자는 첫 번째 인자로 (@Cron() 데코레이터와 마찬가지로) 크론 패턴을, 두 번째 인자로 크론 타이머가 실행될 때 실행될 콜백을 받습니다. SchedulerRegistry#addCronJob 메서드는 두 개의 인수를 받습니다. CronJob의 이름과 CronJob 개체 자체입니다.

> 경고 경고 SchedulerRegistry에 액세스하기 전에 반드시 인젝션해야 합니다. Import 크론 패키지의 크론잡.

다음과 같이 SchedulerRegistry#deleteCronJob 메서드를 사용하여 명명된 크론 작업을 삭제합니다:

```
deleteCron(name: 문자열) {
  this.schedulerRegistry.deleteCronJob(name);
  this.logger.warn(`job ${name} deleted!`);
}
```

다음과 같이 SchedulerRegistry#getCronJobs 메서드를 사용하여 모든 크론 작업을 나열합니다:

```
getCrons() {
  const jobs = this.schedulerRegistry.getCronJobs();
  jobs.forEach((value, key, map) => {{
    let next;
    try {
      next = value.nextDates().toDate();
    } catch (e) {
      다음 = '오류: 다음 발사 날짜가 과거입니다!';
    }
    this.logger.log(`job: ${key} -> next: ${next}`);
  });
}
```

getCronJobs() 메서드는 맵을 반환합니다. 이 코드에서는 맵을 반복하여 각 CronJob의 nextDates() 메서드에 액세스하려고 시도합니다. CronJob API에서는 작업이 이미 실행되었고 향후 실행 날짜가 없는 경우 예외를 던집니다.

동적 간격

SchedulerRegistry#getInterval 메서드를 사용하여 간격에 대한 참조를 얻습니다. 위와 같이 표준 생성자 주입을 사용하여 스케줄러 레지스트리를 생성합니다:

```
constructor(private schedulerRegistry: SchedulerRegistry) {}
```

그리고 다음과 같이 사용하세요:

```
const interval = this.schedulerRegistry.getInterval('notifications');
clearInterval(interval);
```

다음과 같이 SchedulerRegistry#addInterval 메서드를 사용하여 새 간격을 동적으로 생성합니다:

```
addInterval(name: 문자열, milliseconds: 숫자) { const
  callback = () => {
    this.logger.warn(`시간 (${밀리초})에 ${이름} 간격으로 실행 중!`);
  };

  const interval = setInterval(callback, 밀리초);
  this.schedulerRegistry.addInterval(name, interval);
}
```

이 코드에서는 표준 자바스크립트 간격을 생성한 다음 이를 SchedulerRegistry#addInterval 메서드에 전달합니다. 이 메서드에는 간격의 이름과 간격 자체의 두 가지 인수가 필요합니다.

다음과 같이 SchedulerRegistry#deleteInterval 메서드를 사용하여 명명된 간격을 삭제합니다:

```
deleteInterval(name: string) {
  this.schedulerRegistry.deleteInterval(name);
  this.logger.warn(`Interval ${name} deleted!`);
}
```

다음과 같이 `SchedulerRegistry#getIntervals` 메서드를 사용하여 모든 인터벌을 나열합니다:

```
    intervals.forEach(key => this.logger.log(`Interval: ${key}`));
  }
```

동적 시간 초과

SchedulerRegistry#getTimeout 메서드를 사용하여 타임아웃에 대한 참조를 얻습니다. 위와 같이 표준 생성자 주입을 사용하여 스케줄러 레지스트리를 생성합니다:

```
constructor(private readonly schedulerRegistry: SchedulerRegistry) {}
```

그리고 다음과 같이 사용하세요:

```
const timeout = this.schedulerRegistry.getTimeout('알림');
clearTimeout(timeout);
```

다음과 같이 SchedulerRegistry#addTimeout 메서드를 사용하여 새 타임아웃을 동적으로 생성합니다:

```
addTimeout(name: 문자열, milliseconds: 숫자) { const
  callback = () => {
    this.logger.warn(`(${밀리초}) 이후 실행되는 ${이름} 타임
아웃!`);
  };

  const timeout = setTimeout(callback, 밀리초);
  this.schedulerRegistry.addTimeout(name, timeout);
}
```

이 코드에서는 표준 자바스크립트 시간 제한을 생성한 다음 이를 SchedulerRegistry#addTimeout 메서드에 전달합니다. 이 메서드에는 타임아웃의 이름과 타임아웃 자체의 두 가지 인수가 필요합니다.

다음과 같이 SchedulerRegistry#deleteTimeout 메서드를 사용하여 지정된 타임아웃을 삭제합니다:

```
deleteTimeout(name: string) {
  this.schedulerRegistry.deleteTimeout(name);
  this.logger.warn(`타임아웃 ${name} 삭제!`);
}
getTimeouts() {
  const timeouts = this.schedulerRegistry.getTimeouts();
```

다음과 같이 `SchedulerRegistry#getTimeouts` 메서드를 사용하여 모든 시간 초과를 나열합니다:

```
    timeouts.forEach(key => this.logger.log(`Timeout: ${key}`));
  }
```

예

작동하는 예제는 여기에서 확인할 수 있습니다.

대기열

큐는 일반적인 애플리케이션 확장 및 성능 문제를 해결하는 데 도움이 되는 강력한 디자인 패턴입니다. 다음은 큐를 통해 해결할 수 있는 몇 가지 문제 예시입니다:

- 처리 피크를 완화하세요. 예를 들어 사용자가 리소스 집약적인 작업을 임의의 시간에 시작할 수 있는 경우 이러한 작업을 동기적으로 수행하는 대신 대기열에 추가할 수 있습니다. 그런 다음 작업자 프로세스가 제어된 방식으로 대기열에서 작업을 가져오도록 할 수 있습니다. 애플리케이션이 확장됨에 따라 새로운 큐 소비자를 쉽게 추가하여 백엔드 작업 처리를 확장할 수 있습니다.
- 모놀리식 작업을 분할하여 Node.js 이벤트 루프를 차단할 수 있습니다. 예를 들어 사용자 요청에 오디오 트랜스코딩과 같은 CPU 집약적인 작업이 필요한 경우 이 작업을 다른 프로세스에 위임하여 사용자 대면 프로세스가 응답성을 유지할 수 있도록 여유를 확보할 수 있습니다.
- 다양한 서비스에서 안정적인 커뮤니케이션 채널을 제공하세요. 예를 들어, 한 프로세스 또는 서비스에서 작업(작업)을 대기열에 추가하고 다른 서비스에서 사용할 수 있습니다. 모든 프로세스 또는 서비스에서 작업 수명 주기의 완료, 오류 또는 기타 상태 변경 시 상태 이벤트를 수신하여 알림을 받을 수 있습니다. 큐 생산자 또는 소비자가 실패해도 상태가 유지되며 노드가 다시 시작될 때 작업 처리가 자동으로 다시 시작될 수 있습니다.

Nest는 인기 있고 잘 지원되는 고성능 Node.js 기반 큐 시스템 구현인 Bull 위에 추상화/래퍼로 `@nestjs/bull` 패키지를 제공합니다. 이 패키지를 사용하면 애플리케이션에 Nest 친화적인 방식으로 Bull 큐를 쉽게 통합할 수 있습니다.

Bull은 Redis를 사용하여 작업 데이터를 유지하므로 시스템에 Redis가 설치되어 있어야 합니다. Redis를 기반으로 하기 때문에, 큐 아키텍처는 완전히 분산되어 플랫폼에 독립적일 수 있습니다. 예를 들어, 일부 큐 생산자와 소비자 및 리스너는 Nest에서 하나(또는 여러) 노드에서 실행하고 다른 생산자, 소비자 및 리스너는 다른 네트워크 노드의 다른 Node.js 플랫폼에서 실행하도록 할 수 있습니다.

이 장에서는 `@nestjs/bull` 패키지를 다룹니다. 자세한 배경과 구체적인 구현 세부 사항은 Bull 설명서를 읽어보시기 바랍니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

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

설치 프로세스가 완료되면 `BullModule을` 루트 `앱모듈로` 가져올 수 있습니다.

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져오고,
'@nestjs/bull'에서 { BullModule }을 가져옵니다;

모듈({ import: [
    BullModule.forRoot({
      redis: {
```

```
          호스트: 'localhost',
          포트: 6379,
        },
      }),
    ],
  })
  내보내기 클래스 AppModule {}
```

`forRoot()` 메서드는 애플리케이션에 등록된 모든 큐에서 사용할 `불` 패키지 구성 객체를 등록하는 데 사용됩니다 (달리 명시되지 않는 한). 구성 객체는 다음과 같은 프로퍼티로 구성됩니다:

- `리미터: RateLimiter` - 대기열의 작업이 처리되는 속도를 제어하는 옵션입니다. 자세한 내용은 RateLimiter를 참조하세요. 선택 사항입니다.
- `redis: RedisOpts` - Redis 연결을 구성하는 옵션입니다. 자세한 내용은 RedisOpts를 참조하세요. 선택 사항입니다.
- `접두사: 문자열` - 모든 대기열 키의 접두사입니다. 선택 사항입니다.
- `기본 작업 옵션: JobOptions` - 새 작업의 기본 설정을 제어하는 옵션입니다. 자세한 내용은 JobOptions를 참조하세요. 선택 사항입니다.
- `설정을 변경합니다: 고급 설정` - 고급 대기열 구성 설정입니다. 이 설정은 일반적으로 변경하지 않는 것이 좋습니다. 자세한 내용은 고급 설정을 참조하세요. 선택 사항입니다.

모든 옵션은 선택 사항으로, 큐 동작을 세부적으로 제어할 수 있습니다. 이러한 옵션은 Bull `Queue` 생성자에 직접 전달됩니다. 이러한 옵션에 대한 자세한 내용은 여기를 참조하세요.

대기열을 등록하려면 다음과 같이 `BullModule.registerQueue()` 동적 모듈을 임포트합니다:

```
BullModule.registerQueue({
  name: 'audio,
});
```

> 정보 힌트 쉼표로 구분된 여러 개의 구성 개체를 전달하여 여러 개의 대기열을 생성합니다.
> `registerQueue()` 메서드를 호출합니다.

`registerQueue()` 메서드는 큐를 인스턴스화 및/또는 등록하는 데 사용됩니다. 큐는 동일한 자격 증명을 사용하여 동일한 기본 Redis 데이터베이스에 연결하는 모듈 및 프로세스 간에 공유됩니다. 각 큐는 이름 속성에 의해 고유합니다. 큐 이름은 컨트롤러/프로바이더에 큐를 주입하기 위한 주입 토큰과 소비자 클래스 및 리스너를 큐

와 연결하기 위한 데코레이터에 대한 인수로 사용됩니다.

다음과 같이 특정 대기열에 대해 미리 구성된 옵션 중 일부를 재정의할 수도 있습니다:

```
BullModule.registerQueue({
  name: 'audio,
  redis: {
    port: 6380,
```

```
  },
});
```

작업은 Redis에서 지속되므로, 특정 명명된 큐가 인스턴스화될 때마다(예: 앱이 시작/재시작될 때) 이전 완료되지 않은 세션에서 존재할 수 있는 모든 이전 작업을 처리하려고 시도합니다.

각 대기열에는 하나 또는 여러 개의 프로듀서, 소비자, 리스너가 있을 수 있습니다. 소비자는 특정 순서로 큐에서 작업을 검색합니다: FIFO(기본값), LIFO 또는 우선순위에 따라 검색합니다. 큐 처리 순서 제어에 대해서는 여기에서 설명합니다.

명명된 구성

큐가 여러 개의 서로 다른 Redis 인스턴스에 연결되는 경우, 명명된 구성이라는 기술을 사용할 수 있습니다. 이 기능을 사용하면 지정된 키 아래에 여러 구성을 등록한 다음 큐 옵션에서 참조할 수 있습니다.

예를 들어, 애플리케이션에 등록된 몇 개의 큐에서 기본 인스턴스 외에 추가로 사용하는 Redis 인스턴스가 있다고 가정하면 다음과 같이 구성을 등록할 수 있습니다:

```
BullModule.forRoot('alternative-config', {
  redis: {
    포트: 6381,
  },
});
```

위의 예에서 `'alternative-config'는` 구성 키일 뿐입니다(임의의 문자열일 수 있음). 이제 `등록큐큐()` 옵션

객체에서 이 구성을 가리킬 수 있습니다:

```
BullModule.registerQueue({
  configKey: 'alternative-config',
  name: 'video')
});
```

프로듀서

작업 생산자는 대기열에 작업을 추가합니다. 생산자는 일반적으로 애플리케이션 서비스(네스트 공급자)입니다.

큐에 작업을 추가하려면 먼저 다음과 같이 큐를 서비스에 주입합니다:

```
'@nestjs/common'에서 { Injectable }을 임포트하고,
'bull'에서 { Queue }를 임포트합니다;
'@nestjs/bull'에서 { InjectQueue }를 가져옵니다;

@Injectable()
내보내기 클래스 AudioService {
```

```
    생성자(@InjectQueue('audio') private audioQueue: Queue) {}
}
```

> 정보 힌트 `@InjectQueue()` 데코레이터는 이름에 제공된 대로 큐를 식별합니다.

`registerQueue()` 메서드 호출(예: `'audio'`).

이제 큐의 `add()` 메서드를 호출하여 사용자 정의 작업 객체를 전달하여 작업을 추가합니다. 작업은 직렬화 가능한 JavaScript 객체로 표현됩니다(Redis 데이터베이스에 저장되는 방식이므로). 전달하는 작업의 모양은 임의적이므로 작업 객체의 의미를 나타내는 데 사용하십시오.

```
const job = await this.audioQueue.add({
  foo: 'bar',
});
```

명명된 작업

작업에는 고유한 이름이 있을 수 있습니다. 이를 통해 특정 이름의 작업만 처리하는 전문 소비자를 만들 수 있습니다.

```
const job = await this.audioQueue.add('transcode', {
  foo: 'bar',
});
```

> 경고 경고 명명된 작업을 사용할 때는 대기열에 추가된 각 고유 이름에 대해 프로세서를 만들어야 하며, 그렇지 않으면 대기열에서 해당 작업에 대한 프로세서가 없다는 불만을 표시합니다. 명명된 작업 사용에 대한 자세한 내용은 여기를 참조하세요.

작업 옵션

작업에는 추가 옵션을 연결할 수 있습니다. `Queue.add()` 메서드에서 `작업` 인수 뒤에 옵션 개체를 전달합니다. 작업 옵션 속성은 다음과 같습니다:

- `우선순위`: `숫자` - 선택적 우선순위 값입니다. 범위는 1(가장 높은 우선순위)에서 MAX_INT(가장 낮은 우선
- 순위)까지입니다. 우선순위를 사용하면 성능에 약간의 영향을 미치므로 주의해서 사용하세요. `지연`: `숫자` - 이 작업을 처리할 수 있을 때까지 대기할 시간(밀리초)입니다. 정확한 지연을 위해 서버와 클라이언트의 시

계가 모두 동기화되어 있어야 합니다.

- `시도`: `수` - 작업이 완료될 때까지 시도한 총 시도 횟수입니다.
- `반복`: 반복 `옵션` - 크론 사양에 따라 작업을 반복합니다. `백오프`: `숫자 | 백오프 옵션` - 작업 실패 시 자동 재시도를 위한 백오프 설정입니다. 백오프옵션을 참조하세요.
- `lifo`: `부울` - 참이면 작업을 대기열의 왼쪽 끝이 아닌 오른쪽 끝에 추가합니다(기본값은 거짓). `timeout`: `숫자` - 시간 초과 오류로 작업이 실패할 때까지의 시간(밀리초) `jobId`: `숫자 | 문자열` - 작업 ID 재정의 - 기본적으로 작업 ID는 고유 정수이지만 이 설정을 사용하여 이를 재정의할 수 있습니다. 이 옵션을 사용하는 경우 jobId가 고유한지 확인하는 것은 사용자의 책임입니다. 이미 존재하는 ID로 작업을 추가하려고 하면 추가되지 않습니다.

- removeOnComplete: boolean | 숫자 - 참이면 작업이 성공적으로 완료되면 작업을 제거합니다. 숫자는 보관할 작업의 양을 지정합니다. 기본 동작은 완료된 세트에 작업을 유지하는 것입니다.
- removeOnFail: boolean | 숫자 - 참이면 모든 시도 후 실패하면 작업을 제거합니다. 숫자는 유지할 작업의 양을 지정합니다. 기본 동작은 실패한 세트에 작업을 유지하는 것입니다.
- stackTraceLimit: 숫자 - 스택트레이스에 기록될 스택 추적 줄의 양을 제한합니다.

다음은 작업 옵션으로 작업을 사용자 지정하는 몇 가지 예입니다. 작

업 시작을 지연시키려면 지연 구성 속성을 사용합니다.

```
const job = await this.audioQueue.add(
  {
    foo: 'bar',
  },
  { delay: 3000 }, // 3초 지연
);
```

대기열의 오른쪽 끝에 작업을 추가하려면(작업을 선입선출(LIFO)로 처리하려면), 다음과 같이 설정합니다.

속성을 true로 설정합니다.

```
const job = await this.audioQueue.add(
  {
    foo: 'bar',
  },
  { lifo: true },
);
```

작업의 우선순위를 지정하려면 우선순위 속성을 사용하세요.

```
const job = await this.audioQueue.add(
  {
    foo: 'bar',
  },
  { 우선순위: 2 },
);
```

소비자

컨슈머는 큐에 추가된 작업을 처리하거나 큐의 이벤트를 수신하는 메서드를 정의하는 클래스 또는 둘 다를 정의하는 클래스입니다. 다음과 같이 `@Processor()` 데코레이터를 사용하여 소비자 클래스를 선언합니다:

```
'@nestjs/bull'에서 { Processor }를 가져옵니다;
```

```
@Processor('audio')
내보내기 클래스 AudioConsumer {}
```

> 정보 힌트 소비자는 `공급자로` 등록해야 `@nestjs/bull` 패키지가 소비자를 받을 수 있습니다.

여기서 데코레이터의 문자열 인수(예: `'audio'`)는 클래스 메서드와 연결할 대기열의 이름입니다.

소비자 클래스 내에서 핸들러 메서드를 `@Process()`로 장식하여 작업 핸들러를 선언합니다.

데코레이터.

```
'@nestjs/bull'에서 { Processor, Process }를 가져오고,
'bull'에서 { Job }을 가져옵니다;

@Processor('audio')
내보내기 클래스 AudioConsumer {
  @Process()
  async transcode(job: Job<unknown>) {
    let progress = 0;
    for (i = 0; i < 100; i++) {
      await doSomething(job.data);
      progress += 1;
      await job.progress(progress);
    }
    반환 {};
  }
}
```

장식된 메서드(예: `transcode()`)는 워커가 유휴 상태이고 큐에 처리할 작업이 있을 때마다 호출됩니다. 이 핸들러 메서드는 `작업` 객체를 유일한 인수로 받습니다. 처리기 메서드가 반환하는 값은 작업 개체에 저장되며 나중에 완료된 이벤트의 리스너 등에서 액세스할 수 있습니다.

`작업` 객체에는 해당 상태와 상호작용할 수 있는 여러 메서드가 있습니다. 예를 들어 위 코드는 `progress()` 메서드를 사용하여 작업의 진행 상황을 업데이트합니다. 전체 `Job` 객체 API 참조는 여기를 참조하세요.

아래와 같이 `@Process()` 데코레이터에 해당 `이름을` 전달하여 작업 처리기 메서드가 특정 유형의 작업(특정 `이름의` 작업)만 처리하도록 지정할 수 있습니다. 지정된 소비자 클래스에는 각 작업 유형(`이름`)에 해당하는 여러 개의 `@Process()` 핸들러를 가질 수 있습니다. 명명된 작업을 사용하는 경우 각 이름에 해당하는 핸들러가 있어야 합니다.

```
@Process('트랜스코드')
async transcode(job: Job<unknown>) { ... }
```

> 경고 경고 동일한 대기열에 대해 여러 소비자를 정의할 때 `@Process({{ '{' }} 동시성: 1 {{ '}' }})`의 `동시성` 옵션이 적용되지 않습니다. 최소 `동시성은` 정의된 소비자 수와 일치합니다. 이는 `@Process()` 핸들러가 다른 `이름을` 사용하여 명명된 작업을 처리하는 경우에도 적용됩니다.

요청 범위가 지정된 소비자

소비자가 요청 범위로 플래그가 지정되면(여기에서 주입 범위에 대해 자세히 알아보세요), 각 작업에 대해 클래스의 새 인스턴스가 독점적으로 생성됩니다. 인스턴스는 작업이 완료된 후 가비지 수집됩니다.

```
@Processor({
  name: 'audio',
  scope: Scope.REQUEST,
})
```

요청 범위가 지정된 소비자 클래스는 동적으로 인스턴스화되고 단일 작업으로 범위가 지정되므로 표준 접근 방식을 사용하여 생성자를 통해 `JOB_REF를` 주입할 수 있습니다.

```
constructor(@Inject(JOB_REF) jobRef: Job) {
  console.log(jobRef);
}
```

> 정보 힌트 `JOB_REF` 토큰은 `@nestjs/bull` 패키지에서 가져옵니다.

이벤트 리스너

Bull은 대기열 및/또는 작업 상태 변경이 발생할 때 유용한 이벤트 세트를 생성합니다. Nest는 표준 이벤트의 핵심 집합을 구독할 수 있는 데코레이터 세트를 제공합니다. 이러한 데코레이터는 `@nestjs/bull` 패키지에서 내보낼 수 있습니다.

이벤트 리스너는 소비자 클래스 내에서 선언해야 합니다(즉, `@Processor()` 데코레이터로 장식된 클래스 내에서). 이벤트를 수신하려면 아래 표의 데코레이터 중 하나를 사용하여 이벤트에 대한 핸들러를 선언하세요. 예를 들어 작업이 `오디오` 대기열에서 활성 상태가 될 때 발생하는 이벤트를 수신하려면 다음 구문을 사용합니다:

```
'@nestjs/bull'에서 { Processor, Process, OnQueueActive }를 임포트하고,
'bull'에서 { Job }을 임포트합니다;

@Processor('audio')
내보내기 클래스 AudioConsumer {

  OnQueueActive()
  onActive(job: Job) {
    콘솔 로그(
      데이터로 ${job.name} 유형의 ${job.id} 작업을 처리 중입니다.
```

```
    ${job.data}...`,
      );
    }
    ...
```

Bull은 분산(다중 노드) 환경에서 작동하므로 이벤트 로컬리티라는 개념을 정의합니다. 이 개념은 이벤트가 단일 프로세스 내에서만 트리거되거나 다른 프로세스의 공유 큐에서 트리거될 수 있음을 인식합니다. 로컬 이벤트는 로컬 프로세스의 대기열에서 작업 또는 상태 변경이 트리거될 때 생성되는 이벤트입니다. 즉, 이벤트 생산자와 소비자가 단일 프로세스에 로컬인 경우 대기열에서 발생하는 모든 이벤트는 로컬 이벤트입니다.

대기열이 여러 프로세스에서 공유되는 경우 글로벌 이벤트가 발생할 가능성이 있습니다. 한 프로세스의 수신기가 다른 프로세스에 의해 트리거된 이벤트 알림을 수신하려면 글로벌 이벤트에 등록해야 합니다.

이벤트 핸들러는 해당 이벤트가 발생할 때마다 호출됩니다. 핸들러는 아래 표에 표시된 서명으로 호출되며, 이벤트와 관련된 정보에 액세스할 수 있습니다. 아래에서 로컬 이벤트 핸들러 서명과 글로벌 이벤트 핸들러 서명 간의 주요 차이점에 대해 설명합니다.

로컬 이벤트 리스너	글로벌 이벤트 리스너	핸들러 메서드 서명/시기 해고
온큐에러()	온글로벌 큐 에러()	핸들러(오류: 오류) - 오류 오류가 발생했습니다. 오류에는 트리거 오류가 포함되어 있습니다.
@OnQueueWaiting() 온큐큐액티브()	온글로벌 큐 대기() 온글로벌큐큐액티브()	핸들러(jobId: 숫자 | 문자열) - 작업자가 유휴 상태인 즉시 작업이 처리되기를 기다리는 중입니다. jobId에는 이 상태에 진입한 작업의 ID가 포함되어 있습니다.
@OnQueueStalled()	OnGlobalQueueStalled()	핸들러(작업: Job) - 작업이 시작되었습니다. 핸들러(job: 작업) - 작업 작업이 중

단된 것으로 표시되었습니다. 이벤트 루프를 충돌하거나 일시 중지하는 작업자를 디버깅할 때 유용합니다.

`온큐어프로그레스()`	`온글로벌 큐 진행률()`	`핸들러(job: 작업, 진행률: 숫자)` - 작업의 진행률이 `진행률` 값으로 업데이트되었습니다.
`OnQueueCompleted()`	`@OnGlobalQueueCompleted()`	`핸들러(job: Job, result: any)` `결과와` 함께 `작업이` 성공적으로 완료되었습니다.
`@OnQueueFailed()`	`온글로벌 큐 실패()`	`핸들러(job: 작업, err: 오류)` 이유 `오류로 작업이` 실패했습니다.

`@OnQueuePaused()`	`온글로벌 큐 일시정지()`	`핸들러()` 큐가 일시 중지되었습니다.
`대기열 재개()`	`온글로벌 큐 재개()`	`핸들러(작업: 작업)` 큐에 다음이 있습니다. 가 재개되었습니다.
`@OnQueueCleaned()`	`온글로벌 큐 청소()`	`핸들러(jobs: Job[], type: 문자열)` 대기열에서 오래된 작업이 정리되었습니다. jobs는 정리된 작업의 배열이고 `type은` 정리된 작업의 유형입니다.
`@OnQueueDrained()`	`OnGlobalQueueDrained()`	`핸들러()` 대기열이 대기 중인 모든 작업을 처리할 때마다 발생합니다(아직 처리되지 않은 지연된 작업이 있을 수 있음에도 불구하고).
`@OnQueueRemoved()`	`글로벌 대기열 제거()`	`핸들러(작업: Job)` `작업이` 성공적으로 제거되었습니다.

글로벌 이벤트를 수신할 때 메서드 서명은 로컬 버전과 약간 다를 수 있습니다. 특히, 로컬 버전에서 `작업` 객체를 수신하는 메서드 서명은 글로벌 버전에서 `jobId(숫자)`를 수신합니다. 이러한 경우 실제 `작업` 객체에 대한 참조를 얻으려면 `Queue#getJob` 메서드를 사용하세요. 이 호출은 대기 상태여야 하므로 핸들러를 `비동기` 상태로 선언해야 합니다. 예를 들어

```
온글로벌큐큐완료()
async onGlobalCompleted(jobId: number, result: any) {
  const job = await this.immediateQueue.getJob(jobId);
  console.log('(Global) on completed: job ', job.id, ' -> result: ',
result);
}
```

> 정보 힌트 `큐` 객체에 접근하려면(`getJob()` 호출을 위해) 당연히 큐 객체를 주입해야 합니다. 또한 큐를 주입하는 모듈에 큐가 등록되어 있어야 합니다.

특정 이벤트 리스너 데코레이터 외에도 일반 `@OnQueueEvent()` 데코레이터를 `BullQueueEvents` 또는

`BullQueueGlobalEvents` 열거형과 함께 사용할 수도 있습니다. 이벤트에 대한 자세한 내용은 여기를 참조하세요.

대기열 관리

대기열에는 일시 중지 및 재개, 다양한 상태의 작업 수 검색 등 여러 가지 관리 기능을 수행할 수 있는 API가 있습니다. 전체 큐 API는 여기에서 확인할 수 있습니다. 아래 일시 중지/재개 예시와 같이 `큐` 개체에서 직접 이러한 메서드를 호출할 수 있습니다.

`pause()` 메서드 호출로 큐를 일시 중지합니다. 일시 중지된 큐는 재개될 때까지 새 작업을 처리하지 않지만 처리 중인 현재 작업은 완료될 때까지 계속됩니다.

```
오디오큐브.일시정지()를 기다립니다;
```

일시 중지된 대기열을 다시 시작하려면 다음과 같이 `resume()` 메서드를 사용합니다:

```
오디오 큐를 기다립니다;
```

별도의 프로세스

작업 처리기는 별도의 (포크된) 프로세스(소스)에서 실행할 수도 있습니다. 여기에는 몇 가지 장점이 있습니다:

- 프로세스는 샌드박스가 적용되므로 충돌이 발생해도 작업자에게 영향을 미치
- 지 않습니다. 대기열에 영향을 주지 않고 차단 코드를 실행할 수 있습니다(작업
- 이 중단되지 않음). 멀티코어 CPU를 훨씬 더 잘 활용합니다.
- redis에 대한 연결이 줄어듭니다.

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져오고,
'@nestjs/bull'에서 { BullModule }을 가져오고,
'path'에서 { join }을 가져옵니다;

모듈({ import: [
    BullModule.registerQueue({
      name: 'audio,
      프로세서입니다: [join(__dirname, 'processor.js')],
    }),
  ],
})
내보내기 클래스 AppModule {}
```

함수가 포크된 프로세스에서 실행되고 있기 때문에 의존성 주입(및 IoC 컨테이너)을 사용할 수 없다는 점에 유의하세요. 즉, 프로세서 함수는 필요한 외부 종속성의 모든 인스턴스를 포함(또는 생성)해야 합니다.

```
@@파일명(프로세서)
'bull'에서 { Job, DoneCallback }을 가져옵니다;

export default function (job: Job, cb: DoneCallback) {
  console.log(`[${process.pid}] ${JSON.stringify(job.data)}`);
  cb(null, 'It works');
}
```

비동기 구성

정적이 아닌 비동기적으로 `불` 옵션을 전달하고 싶을 수도 있습니다. 이 경우 비동기 구성을 처리하는 여러 가지 방법을 제공하는 `forRootAsync()` 메서드를 사용하세요. 마찬가지로 큐 옵션을 비동기적으로 전달하려면 `registerQueueAsync()` 메서드를 사용하세요.

한 가지 방법은 팩토리 함수를 사용하는 것입니다:

```
BullModule.forRootAsync({
  useFactory: () => ({
    redis: {
      호스트: 'localhost', 포
      트: 6379,
    },
  }),
});
```

저희 팩토리는 다른 비동기 공급자처럼 동작합니다(예: `비동기일` 수 있고, `주입을` 통해 종속성을 주입할 수 있습니다).

```
BullModule.forRootAsync({
  import: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    redis: {
      host: configService.get('QUEUE_HOST'),
      port: configService.get('QUEUE_PORT'),
    },
  }),
  주입합니다: [구성 서비스],
});
```

또는 `useClass` 구문을 사용할 수도 있습니다:

```
BullModule.forRootAsync({
  useClass: BullConfigService,
});
```

위의 구성은 `BullModule` 내부에 `BullConfigService를` 인스턴스화하고 이를 사용하여 `createSharedConfiguration()을` 호출하여 옵션 객체를 제공합니다. 이는 아래 그림과 같이 BullConfigService가 `SharedBullConfigurationFactory` 인터페이스를 구현해야 한다는 것을 의미합니다:

```
@Injectable()
BullConfigService 클래스는 SharedBullConfigurationFactory를 구현합니다 {
  createSharedConfiguration(): BullModuleOptions {
    반환 {
      redis: {
```

```
          호스트: 'localhost',
          포트: 6379,
        },
      };
    }
  }
```

`BullModule` 내부에 `BullConfigService를` 생성하지 않고 다른 모듈에서 가져온 프로바이더를 사용하려면 `useExisting` 구문을 사용할 수 있습니다.

```
BullModule.forRootAsync({
  imports: [ConfigModule],
  useExisting: ConfigService,
});
```

이 구조는 `useClass와` 동일하게 작동하지만 한 가지 중요한 차이점이 있습니다. BullModule은 가져온 모듈을 조회하여 새 `구성 서비스를` 인스턴스화하는 대신 기존 `구성 서비스를` 재사용합니다.

예

작동하는 예제는 여기에서 확인할 수 있습니다.

로거

Nest에는 애플리케이션 부트스트래핑 및 잡힌 예외 표시(예: 시스템 로깅)와 같은 여러 상황에서 사용되는 텍스트 기반 로거가 내장되어 있습니다. 이 기능은 `@nestjs/common` 패키지의 `Logger` 클래스를 통해 제공됩니다. 다음 중 하나를 포함하여 로깅 시스템의 동작을 완전히 제어할 수 있습니다:

- 로깅을 완전히 비활성화합니다.
- 로그 세부 수준 지정(예: 오류, 경고, 디버그 정보 표시 등)• 기본 로거의 타임스탬프 재정의(예: 날짜 형식으로 ISO8601 표준 사용)• 기본 로거를 완전히 재정의합니다.
- 기본 로거를 확장하여 사용자 정의하기
- 종속성 주입을 사용하여 애플리케이션 작성 및 테스트를 간소화하세요.

기본 제공 로거를 사용하거나 사용자 정의 구현을 만들어 애플리케이션 수준 이벤트와 메시지를 직접 기록할 수도 있습니다.

고급 로깅 기능을 사용하려면 Winston과 같은 Node.js 로깅 패키지를 사용하여 완전히 사용자 정의된 프로덕션 등급 로깅 시스템을 구현할 수 있습니다.

기본 사용자 지정

로깅을 비활성화하려면 `NestFactory.create()` 메서드의 두 번째 인수로 전달되는 (선택 사항) Nest 애플리케이션 옵션 객체에서 `logger` 속성을 `false로` 설정합니다.

```
const app = await NestFactory.create(AppModule, {
  logger: false,
});
await app.listen(3000);
```

특정 로깅 수준을 활성화하려면 다음과 같이 표시할 로그 수준을 지정하는 문자열 배열로 `logger` 속성을 설정합니다:

```
const app = await NestFactory.create(AppModule, {
  logger: ['error', 'warn'],
});
await app.listen(3000);
```

배열의 값은 `'로그'`, `'오류'`, `'경고'`, `'디버그'`, `'상세'` 중 어떤 조합이든 사용할 수 있습니다.

> 정보 힌트 기본 로거의 메시지에서 색상을 사용하지 않으려면 `NO_COLOR` 환경 변수를 비어 있지 않은 문자열로 설정하세요.

사용자 지정 구현

로거 프로퍼티의 값을 LoggerService 인터페이스를 구현하는 객체로 설정하여 Nest에서 시스템 로깅에 사용할 사용자 정의 로거 구현을 제공할 수 있습니다. 예를 들어, 다음과 같이 Nest에 내장된 전역 JavaScript 콘솔 객체(LoggerService 인터페이스를 구현)를 사용하도록 지시할 수 있습니다:

```
const app = await NestFactory.create(AppModule, {
  logger: console,
});
await app.listen(3000);
```

나만의 커스텀 로거를 구현하는 방법은 간단합니다. 각 메서드를 구현하기만 하면 됩니다.

LoggerService 인터페이스와 같습니다.

```
'@nestjs/common'에서 { LoggerService } 임포트; 내보내기

클래스 MyLogger는 LoggerService를 구현합니다.
  /**
   * '로그' 수준의 로그를 작성합니다.
   */
  log(message: any, ...optionalParams: any[]) {}

  /**
   * '오류' 수준 로그를 작성합니다.
   */
  error(message: any, ...optionalParams: any[]) {}

  /**
   * '경고' 수준 로그를 작성합니다.
   */
  warn(message: any, ...optionalParams: any[]) {}

  /**
   * '디버그' 수준 로그를 작성합니다.
   */
  debug?(message: any, ...optionalParams: any[]) {}

  /**
   * '상세' 수준 로그를 작성합니다.
   */
  verbose?(message: any, ...optionalParams: any[]) {}
}
```

그런 다음 Nest 애플리케이션 옵션 개체의 logger 속성을 통해 MyLogger 인스턴스를 제공할 수 있습니다.

```
const app = await NestFactory.create(AppModule, {
  logger: new MyLogger(),
```

```
  });
  await app.listen(3000);
```

이 기법은 간단하지만 `MyLogger` 클래스에 대한 의존성 주입을 활용하지 않습니다. 이는 특히 테스트 시 몇 가지 문제를 야기할 수 있으며 `MyLogger의` 재사용성을 제한할 수 있습니다. 더 나은 해결 방법은 아래의 의존성 주입 섹션을 참조하세요.

내장 로거 확장

로거를 처음부터 작성하는 대신 기본 제공 로거를 확장하여 요구 사항을 충족할 수 있습니다. `ConsoleLogger` 클래스 및 기본 구현의 선택된 동작을 재정의합니다.

```
'@nestjs/common'에서 { ConsoleLogger } 임포트; 내보내기

클래스 MyLogger extends ConsoleLogger {
  error(message: any, stack?: string, context?: string) {
    // 여기에 맞춤형 로직을 추가하세요;
  }
}
```

아래의 애플리케이션 로깅을 위한 로거 사용 섹션에 설명된 대로 기능 모듈에서 이러한 확장 로거를 사용할 수 있습니다.

위의 사용자 정의 구현 섹션에 표시된 것처럼 애플리케이션 옵션 객체의 `logger` 속성을 통해 인스턴스를 전달하거나 아래의 종속성 주입 섹션에 표시된 기술을 사용하여 Nest가 시스템 로깅에 확장 로거를 사용하도록 지시할 수 있습니다. 이 경우 위의 샘플 코드에 표시된 것처럼 `super를` 호출하여 특정 로그 메서드 호출을 부모(내장) 클래스에 위임하여 Nest가 예상되는 내장 기능을 사용할 수 있도록 해야 합니다.

종속성 주입

고급 로깅 기능을 사용하려면 종속성 주입을 활용하고 싶을 것입니다. 예를 들어 로거에 `컨피그서비스를` 주입하여 사용자 정의한 다음 다른 컨트롤러 및/또는 프로바이더에 사용자 정의 로거를 주입할 수 있습니다. 사용자 정의 로거에 종속성 주입을 사용하려면 `LoggerService를` 구현하는 클래스를 생성하고 해당 클래스를 일부 모듈에 프로바이더로 등록하세요. 예를 들어 다음과 같이 할 수 있습니다.

. 이전 섹션에 표시된 대로 기본 제공 ConsoleLogger를 확장하거나 완전히 재정의하는 `MyLogger` 클래스

를 정의합니다. `LoggerService` 인터페이스를 구현해야 합니다.

. 아래와 같이 `LoggerModule`을 생성하고 해당 모듈에서 `MyLogger`를 제공합니다.

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./my-logger.service'에서 { MyLogger }를 가져옵니다;

모듈({
  공급자: [MyLogger],
```

```
  내보내기: [MyLogger],
})
내보내기 클래스 LoggerModule {}
```

이 구성을 사용하면 이제 다른 모듈에서 사용할 수 있도록 사용자 정의 로거를 제공하게 됩니다. `MyLogger` 클래스는 모듈의 일부이므로 의존성 주입을 사용할 수 있습니다(예: `컨피그서비스` 주입). Nest에서 시스템 로깅(예: 부트스트랩 및 오류 처리)에 사용할 수 있도록 이 사용자 정의 로거를 제공하는 데 필요한 기술이 한 가지 더 있습니다.

애플리케이션 인스턴스화(`NestFactory.create()`)는 모듈의 컨텍스트 외부에서 발생하기 때문에 초기화의 일반적인 의존성 주입 단계에 참여하지 않습니다. 따라서 적어도 하나의 애플리케이션 모듈이 `LoggerModule`을 임포트하여 Nest가 `MyLogger` 클래스의 싱글톤 인스턴스를 인스턴스화하도록 트리거하도록 해야 합니다.

그런 다음 Nest에 다음과 같은 구성으로 동일한 싱글톤 인스턴스를 사용하도록 지시할 수 있습니다:

```
const app = await NestFactory.create(AppModule, {
  bufferLogs: true,
});
app.useLogger(app.get(MyLogger));
await app.listen(3000);
```

> 정보 참고 위의 예제에서는 사용자 정의 로거(이 경우 `MyLogger`)가 첨부되고 애플리케이션 초기화 프로세스가 완료되거나 실패할 때까지 모든 로그가 버퍼링되도록 `bufferLogs를 true로` 설정했습니다. 초기화 프로세스가 실패하면 Nest는 원래 `ConsoleLogger로` 폴백하여 보고된 오류 메시지를 인쇄합니다.
>
> 또한, `자동 플러시 로그를 거짓`(기본값 `참`)으로 설정하여 로그를 수동으로 플러시할 수 있습니다 (`Logger#flush()` 메서드 사용).

여기서는 `NestApplication` 인스턴스의 `get()` 메서드를 사용하여 `MyLogger` 객체의 싱글톤 인스턴스를 검색합니다. 이 기술은 본질적으로 Nest에서 사용할 로거 인스턴스를 "주입"하는 방법입니다. `app.get()` 호출은 `MyLogger의` 싱글톤 인스턴스를 검색하며, 위에서 설명한 대로 해당 인스턴스가 다른 모듈에 먼저 주입되는지에 따라 달라집니다.

기능 클래스에 이 `MyLogger` 공급자를 삽입하여 Nest 시스템 로깅과 애플리케이션 로깅 모두에서 일관된 로깅 동작을 보장할 수도 있습니다. 자세한 내용은 아래에서 애플리케이션 로깅에 로거 사용 및 사용자 정의 로거 삽입하기를 참조하세요.

애플리케이션 로깅에 로거 사용

위의 몇 가지 기술을 결합하여 Nest 시스템 로깅과 자체 애플리케이션 이벤트/메시지 로깅 모두에서 일관된 동작과 형식을 제공할 수 있습니다.

좋은 방법은 각 서비스에서 `@nestjs/common의 Logger` 클래스를 인스턴스화하는 것입니다. 다음과 같이 `Logger` 생성자에서 서비스 이름을 `컨텍스트` 인수로 제공할 수 있습니다:

```
'@nestjs/common'에서 { 로거, 인젝터블 }을 임포트합니다;
```

```
주입 가능() 클래스
MyService {
  비공개 읽기 전용 로거 = 새로운 로거(MyService.name);

  doSomething() {
    this.logger.log('작업 중...');
  }
}
```

기본 로거 구현에서 컨텍스트는 아래 예제의 NestFactory와 같이 대괄호 안에 인쇄됩니다:

```
Nest] 19096     12/08/2019, 7:12:59 AM          [NestFactory]Nest 애플리케이션 시
작 중...
```

app.useLogger()를 통해 사용자 정의 로거를 제공하면 실제로 Nest 내부에서 사용하게 됩니다. 즉, 코드가 구현에 구애받지 않고 유지되며, app.useLogger()를 호출하여 기본 로거를 사용자 정의 로거로 쉽게 대체할 수 있습니다.

이렇게 하면 이전 섹션의 단계를 따라 app.useLogger(app.get(MyLogger))를 호출하면 MyService에서 이.logger.log()를 다음과 같이 호출하면 MyLogger 인스턴스에서 메서드 log를 호출하게 됩니다.

이 방법이 대부분의 경우에 적합합니다. 그러나 사용자 지정 메서드 추가 및 호출과 같은 추가 사용자 지정이 필요한 경우 다음 섹션으로 이동하세요.

사용자 지정 로거 삽입

시작하려면 다음과 같은 코드를 사용하여 기본 제공 로거를 확장합니다. 각 기능 모듈에 고유한 MyLogger 인스턴스를 갖도록 하기 위해 일시적인 범위를 지정하는 ConsoleLogger 클래스에 대한 구성 메타데이터로 범위 옵션을 제공합니다. 이 예제에서는 개별 ConsoleLogger 메서드(로그(), 경고() 등)를 확장하지 않지만 원하는 경우 확장할 수 있습니다.

```
'@nestjs/common'에서 { Injectable, Scope, ConsoleLogger } import;

@Injectable({ scope: Scope.TRANSIENT })
export class MyLogger extends ConsoleLogger {
  customLog() {
    this.log('고양이에게 밥을 주세요!');
  }
}
```

다음으로, 다음과 같은 구조의 `LoggerModule`을 생성합니다:

```
'@nestjs/common'에서 { Module }을 가져옵니다;
'./my-logger.service'에서 { MyLogger }를 가져옵니다;
```

```
모듈({
  제공자: [MyLogger], 내보내기
  : [MyLogger],
})
내보내기 클래스 LoggerModule {}
```

다음으로 `LoggerModule`을 기능 모듈로 가져옵니다. 기본 `로거`를 확장했기 때문에 `setContext` 메서드를 편리하게 사용할 수 있습니다. 이제 다음과 같이 컨텍스트 인식 커스텀 로거를 사용할 수 있습니다:

```
'@nestjs/common'에서 { Injectable }을 임포트하고,
'./my-logger.service'에서 { MyLogger }를 임포트합
니다;

@Injectable()
내보내기 클래스 CatsService {
  비공개 읽기 전용 고양이: Cat[] = [];

  constructor(private myLogger: MyLogger) {
    // 일시적인 범위로 인해 CatsService에는 고유한 MyLogger 인스턴스가 있습니다,
    // 따라서 여기서 컨텍스트를 설정해도 다른 서비스의 다른 인스턴스에는 영향을 미치지 않습니다.
    this.myLogger.setContext('CatsService');
  }

  findAll(): Cat[] {
    //  모든  기본  메소드를  호출할  수  있습니다
    this.myLogger.warn('고양이를  반환하려고  합니다
    !');
    //  그리고  사용자  정의  메서드
    this.myLogger.customLog();
    반환 this.cats;
  }
}
```

마지막으로, 아래와 같이 `메인.ts` 파일에서 사용자 정의 로거의 인스턴스를 사용하도록 Nest에 지시합니다. 물론 이 예제에서는 `log()`, `warn()` 등과 같은 `로거` 메서드를 확장하여 로거 동작을 실제로 사용자 정의하지 않았으므로 이 단계는 실제로 필요하지 않습니다. 하지만 해당 메서드에 사용자 정의 로직을 추가하고 Nest가 동일한 구현을 사용하도록 하려면 이 단계가 필요합니다.

```
const app = await NestFactory.create(AppModule, {
  bufferLogs: true,
});
app.useLogger(new MyLogger());
await app.listen(3000);
```

> 정보 힌트 또는 `bufferLogs를 true로` 설정하는 대신 `logger: false` 명령어를 사용하여 일시적으로 로거를 비활성화할 수 있습니다. `NestFactory.create에 logger: false를` 제공하면 `useLogger를` 호출할 때까지 아무 것도 기록되지 않으므로 중요한 초기화 오류를 놓칠 수 있다는 점에 유의하세요. 초기 메시지 중 일부가 기본 로거로 기록되어도 괜찮다면 `logger: false` 옵션을 생략할 수 있습니다.

외부 로거 사용

프로덕션 애플리케이션에는 고급 필터링, 서식 지정, 중앙 집중식 로깅 등 특정 로깅 요구 사항이 있는 경우가 많습니다. Nest의 기본 제공 로거는 Nest 시스템 동작을 모니터링하는 데 사용되며 개발 중 기능 모듈에서 기본 형식의 텍스트 로깅에도 유용할 수 있지만, 프로덕션 애플리케이션은 Winston과 같은 전용 로깅 모듈을 활용하는 경우가 많습니다. 다른 표준 Node.js 애플리케이션과 마찬가지로 Nest에서도 이러한 모듈을 최대한 활용할 수 있습니다.

쿠키

HTTP 쿠키는 사용자의 브라우저에 저장되는 작은 데이터 조각입니다. 쿠키는 웹사이트가 상태 정보를 기억할 수 있는 신뢰할 수 있는 메커니즘으로 설계되었습니다. 사용자가 웹사이트를 다시 방문하면 요청과 함께 쿠키가 자동으로 전송됩니다.

Express와 함께 사용(기본값)

먼저 필요한 패키지(TypeScript 사용자의 경우 해당 유형)를 설치합니다:

```
$ npm i 쿠키 파서
$ npm i -D @타입스/쿠키-파서
```

설치가 완료되면 `쿠키 파서` 미들웨어를 글로벌 미들웨어로 적용합니다(예: `main.ts` 파일).

```
'쿠키 파서'에서 쿠키 파서로 *를 가져옵니다;
// 초기화 파일 어딘가에 앱.사용(쿠키파서());
```

`쿠키파서` 미들웨어에 몇 가지 옵션을 전달할 수 있습니다:

- `비밀` 쿠키를 서명하는 데 사용되는 문자열 또는 배열입니다. 이 옵션은 선택 사항이며 지정하지 않으면 서명된 쿠키를 구문 분석하지 않습니다. 문자열이 제공되면 이 문자열이 비밀로 사용됩니다. 배열을 제공하면 각 비밀을 순서대로 사용하여 쿠키의 서명을 해제하려고 시도합니다.
- 옵션은 두 번째 옵션으로 `cookie.parse에` 전달되는 객체입니다. 자세한 내용은 쿠키를 참조하세요.

미들웨어는 요청에 대한 `쿠키` 헤더를 구문 분석하여 쿠키 데이터를 `req.cookies` 속성으로, 비밀이 제공된 경우 `req.signedCookies` 속성으로 노출합니다. 이러한 속성은 쿠키 이름과 쿠키 값의 이름 값 쌍입니다.

비밀이 제공되면 이 모듈은 서명된 쿠키 값의 서명을 해제하고 유효성을 검사한 후 해당 이름 값 쌍을 `req.cookies에서 req.signedCookies로` 이동합니다. 서명된 쿠키는 값 앞에 `s:` 접두사가 붙은 쿠키입니다. 서명 유효성 검사에 실패한 서명된 쿠키는 변조된 값 대신 `false` 값을 갖습니다.

이제 다음과 같이 라우트 핸들러 내에서 쿠키를 읽을 수 있습니다:

```
@Get()
findAll(@Req() request: 요청) {
  console.log(request.cookies); // 또는 "request.cookies['cookieKey']"
  // 또는 console.log(request.signedCookies);
}
```

> 정보 힌트 `@Req()` 데코레이터는 @nestjs/common에서 가져온 것이고, `요청은 익스프레스` 패키지.

발신 응답에 쿠키를 첨부하려면 `응답#쿠키()` 메서드를 사용합니다:

```
@Get()
findAll(@Res({ 패스스루: true }) response: Response) { response.cookie('key',
  'value')
}
```

> 경고 경고 응답 처리 로직을 프레임워크에 맡기려면 위와 같이 `패스스루` 옵션을 `true로` 설정해야 합니다. 자세한 내용은 여기를 참조하세요.

> 정보 힌트 `@Res()` 데코레이터는 @nestjs/common에서 가져오고, 응답은 `익스프레스` 패키지.

Fastify와 함께 사용

먼저 필요한 패키지를 설치합니다:

```
$ npm i @fastify/cookie
```

설치가 완료되면 `@fastify/cookie` 플러그인을 등록합니다:

```
'@fastify/cookie'에서 fastifyCookie를 가져옵니다;

// 초기화 파일 어딘가에
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  새로운 FastifyAdapter(),
);
await app.register(fastifyCookie, {
  secret: 'my-secret', // 쿠키 서명용
});
```

이제 다음과 같이 라우트 핸들러 내에서 쿠키를 읽을 수 있습니다:

```
@Get()
findAll(@Req() request: FastifyRequest) {
  console.log(request.cookies); // 또는
  "request.cookies['cookieKey']"
}
```

> 정보 힌트 `@Req()` 데코레이터는 `@nestjs/common`에서 가져온 반면, `FastifyRequest`는 를 `Fastify` 패키지에 추가합니다.

발신 응답에 쿠키를 첨부하려면 `FastifyReply#setCookie()` 메서드를 사용합니다:

```
@Get()
findAll(@Res({ 패스스루: true }) 응답: FastifyReply) { response.setCookie('key',
  'value')
}
```

`FastifyReply#setCookie()` 메서드에 대해 자세히 알아보려면 이 페이지를 확인하세요.

> 경고 경고 응답 처리 로직을 프레임워크에 맡기려면 위와 같이 `패스스루` 옵션을 `true로` 설정해야 합니다. 자세한 내용은 여기를 참조하세요.

> 정보 힌트 `@Res()` 데코레이터는 `@nestjs/common`에서, `FastifyReply`는 `fastify` 패키지에서 가져옵니다.

사용자 지정 데코레이터 만들기(크로스 플랫폼)

들어오는 쿠키에 액세스하는 편리하고 선언적인 방법을 제공하기 위해 사용자 지정 데코레이터를 만들 수 있습니다.

```
import { createParamDecorator, ExecutionContext } from '@nestjs/common';

export const Cookies = createParamDecorator(
  (데이터: 문자열, ctx: 실행 컨텍스트) => {
    const request = ctx.switchToHttp().getRequest();
    반환 데이터 ? 요청.쿠키?.[데이터] : 요청.쿠키;
  },
);
```

`쿠키()` 데코레이터는 모든 쿠키 또는 `req.cookies` 객체에서 명명된 쿠키를 추출하여 데코레이션된 매개변수를 해당 값으로 채웁니다.

이제 다음과 같이 경로 핸들러 시그니처에 데코레이터를 사용할 수 있습니다:

```
@Get()
findAll(@Cookies('name') name: string) {}
```

이벤트

이벤트 이미터 패키지(@nestjs/event-emitter)는 간단한 옵저버 구현을 제공하여 애플리케이션에서 발생하는 다양한 이벤트를 구독하고 수신할 수 있도록 합니다. 이벤트는 하나의 이벤트에 서로 의존하지 않는 여러 리스너를 가질 수 있으므로 애플리케이션의 다양한 측면을 분리하는 좋은 방법입니다.

EventEmitterModule은 내부적으로 eventemitter2 패키지를 사용합니

다. 시작하기

먼저 필요한 패키지를 설치합니다:

```
npm i --save @nestjs/event-emitter
```

설치가 완료되면 이벤트이미터모듈을 루트 앱모듈로 가져와서

forRoot() 정적 메서드를 호출합니다:

```
@@파일명(앱.모듈)
'@nestjs/common'에서 { Module }을 가져옵니다;
'@nestjs/event-emitter'에서 { EventEmitterModule }을 임포트합니다;

모듈({ import: [
    EventEmitterModule.forRoot()
  ],
})
내보내기 클래스 AppModule {}
```

.forRoot() 호출은 이벤트 이미터를 초기화하고 앱 내에 존재하는 모든 선언적 이벤트 리스너를 등록합니다. 등록은 onApplicationBootstrap 수명 주기 후크가 발생할 때 발생하며, 모든 모듈이 모든 예약된 작업을 로드하고 선언했는지 확인합니다.

기본 이벤트이미터 인스턴스를 구성하려면 구성 객체를 .forRoot() 메서드에 전달합니다.

메서드에 대해 다음과 같이 설명합니다:

```
EventEmitterModule.forRoot({
  // 와일드카드를 사용하려면 이것을 'true'로 설정합니다
  . 와일드카드: false,
  // 네임스페이스를 구분하는 데 사용되는 구분 기호 구분 기호:
  '.',
  // 새로운 리스너 이벤트를 발생시키려면 이것을 `true`로 설정합니다,
  // 제거 리스너 이벤트를 발생시키려면 이 값을 `true`로 설정합니다,
  // 이벤트에 할당할 수 있는 최대 리스너 수입니다.
```

```
  최대 청취자: 10,
  // 최대 리스너 수보다 많은 리스너가 할당된 경우 메모리 누수 메시지에 이벤트 이름 표시
  verboseMemoryLeak: false,
  // 오류 이벤트가 발생하고 리스너가 없는 경우 불포함 예외 발생을 비활성화합니다.
  무시 오류: 거짓,
});
```

이벤트 발송

이벤트를 디스패치(즉, 발동)하려면 먼저 표준 생성자 주입을 사용하여 `EventEmitter2를` 주입합니다:

```
constructor(private eventEmitter: EventEmitter2) {}
```

> 정보 힌트 `@nestjs/event-emitter` 패키지에서 `EventEmitter2를` 가져옵니다.

그런 다음 다음과 같이 수업에서 사용하세요:

```
this.eventEmitter.emit(
  'order.created',
  새로운 주문 생성 이벤트({
    orderId: 1,
    페이로드: {},
  }),
);
```

이벤트 듣기

이벤트 리스너를 선언하려면 다음과 같이 실행할 코드가 포함된 메서드 정의 앞에 `@OnEvent()` 데코레이터를 사용하여 메서드를 장식합니다:

```
OnEvent('order.created')
handleOrderCreatedEvent(payload: OrderCreatedEvent) {
  // "주문 생성 이벤트" 이벤트 처리 및 처리
}
```

> 경고 경고 이벤트 구독자는 요청 범위를 지정할 수 없습니다.

첫 번째 인수는 단순 이벤트 이미터의 경우 `문자열` 또는 심볼, 와일드카드 이미터의 경우 `문자열 | 심볼 |`

`Array<string | 심볼>`일 수 있습니다. 두 번째 인수(선택 사항)는 리스너 옵션 객체입니다(자세히 보기).

```
OnEvent('order.created', { async: true })
handleOrderCreatedEvent(payload: OrderCreatedEvent) {
  // "주문 생성 이벤트" 이벤트 처리 및 처리
}
```

네임스페이스/와일드카드를 사용하려면 이벤트이미터모듈#forRoot() 메서드에 와일드카드 옵션을 전달하세요. 네임스페이스/와일드카드가 활성화되면 이벤트는 구분 기호로 구분된 문자열(foo.bar)이거나 배열(['foo', 'bar'])이 될 수 있습니다. 구분 기호는 구성 속성(구분 기호)으로도 구성할 수 있습니다. 네임스페이스 기능을 활성화하면 와일드카드를 사용하여 이벤트를 구독할 수 있습니다:

```
@OnEvent('order.*')
handleOrderEvents(payload: OrderCreatedEvent | OrderRemovedEvent |
OrderUpdatedEvent) {
  // 이벤트 처리 및 처리
}
```

이러한 와일드카드는 하나의 블록에만 적용됩니다. 예를 들어 order.* 인수는 order.created 및 order.shipped 이벤트와 일치하지만 order.delayed.out_of_stock 이벤트와는 일치하지 않습니다. 이러한 이벤트를 수신하려면 EventEmitter2 문서에 설명된 다중 레벨 와일드카드 패턴(예: **)을 사용하세요.

예를 들어 이 패턴을 사용하면 모든 이벤트를 수신하는 이벤트 리스너를 만들 수 있습니다.

```
@OnEvent('**')
handleEverything(payload: any) {
  // 이벤트 처리 및 처리
}
```

> 정보 힌트 이벤트이미터2 클래스는 다음과 같이 이벤트와 상호작용하는 데 유용한 몇 가지 메서드를 제공합니다.
>
> waitFor 및 onAny. 자세한 내용은 여기에서 확인할 수 있습니다.

예

작동하는 예제는 여기에서 확인할 수 있습니다.

압축

압축은 응답 본문의 크기를 크게 줄여 웹 앱의 속도를 높일 수 있습니다.

운영 중인 트래픽이 많은 웹사이트의 경우 일반적으로 역방향 프록시(예: Nginx)를 통해 애플리케이션 서버에서 압축을 오프로드하는 것이 좋습니다. 이 경우 압축 미들웨어를 사용해서는 안 됩니다.

Express와 함께 사용(기본값)

압축 미들웨어 패키지를 사용하여 gzip 압축을 활성화합니다. 먼저 필요한 패키

```
$ npm i --save 압축
```

지를 설치합니다:

설치가 완료되면 압축 미들웨어를 글로벌 미들웨어로 적용합니다.

```
'압축'에서 *를 압축으로 가져옵니다;
// 초기화 파일 어딘가에 앱.사용(압축());
```

Fastify와 함께 사용

FastifyAdapter를 사용하는 경우 fastify-compress를 사용하는 것이 좋습니다:

```
$ npm i --save @fastify/compress
```

설치가 완료되면 `@fastify/compress` 미들웨어를 글로벌 미들웨어로 적용합니다.

```
'@fastify/compress'에서 압축을 가져옵니다;
// 초기화 파일 어딘가에 await
app.register(compression);
```

브라우저가 인코딩을 지원한다고 표시하면 기본적으로 @fastify/compress는 (노드 >= 11.7.0에서) Brotli 압축

을 사용합니다. Brotli는 압축률 측면에서 매우 효율적이지만 속도도 상당히 느립니다. 따라서 응답을 압축할 때 deflate와 gzip만 사용하도록 fastify-compress에 지시하면 응답 크기가 커지지만 훨씬 더 빠르게 전달됩니다.

인코딩을 지정하려면 `app.register에` 두 번째 인수를 제공합니다:

```
await app.register(compression, { encodings: ['gzip', 'deflate'] });
```

위의 내용은 클라이언트가 두 인코딩을 모두 지원하는 경우 gzip과 디플리플레이트 인코딩만 사용하도록 fastify-compress에 지시하고 gzip을 선호합니다.

파일 업로드

파일 업로드를 처리하기 위해 Nest는 Express용 멀티 미들웨어 패키지를 기반으로 하는 내장 모듈을 제공합니다. 멀티는 주로 HTTP POST 요청을 통해 파일을 업로드하는 데 사용되는 멀티파트/폼 데이터 형식으로 게시된 데이터를 처리합니다. 이 모듈은 완전히 구성할 수 있으며 애플리케이션 요구 사항에 맞게 동작을 조정할 수 있습니다.

> 경고 멀티터는 지원되는 다중 파트 형식(다중 파트/양식 데이터)이 아닌 데이터를 처리할 수 없습니다. 또한 이 패키지는 FastifyAdapter와 호환되지 않습니다.

더 나은 타이핑 안전을 위해 다중 타이핑 패키지를 설치해 보겠습니다:

```
$ npm i -D @types/multer
```

이 패키지가 설치되었으므로 이제 Express.Multer.File 유형을 사용할 수 있습니다(이 유형을 다음과 같이 가져올 수 있습니다: import {{ '{' }} Express {{ '}' }} from 'express').

기본 예제

단일 파일을 업로드하려면 FileInterceptor() 인터셉터를 라우트 핸들러에 연결하고 다음을 추출하면 됩니다. 데코레이터를 사용하여 요청에서 파일을 업로드합니다.

```
@@파일명()
@Post('upload')
사용 인터셉터(FileInterceptor('file')) 업로드 파일(@UploadedFile() 파
일: Express.Multer.File) {
  콘솔 로그(파일);
}
스위치 @포스트('업로드')
사용 인터셉터(FileInterceptor('file')) 바인드(업로드된 파일())
uploadFile(file) {
  console.log(file);
}
```

> 정보 힌트 FileInterceptor() 데코레이터는 @nestjs/platform-express에서 내보냅니다. 패키지로 내보냅니다. 업로드된 파일() 데코레이터는 @nestjs/common에서 내보냅니다.

`FileInterceptor()` 데코레이터는 두 개의 인수를 받습니다:

- `fieldName`: 파일을 담고 있는 HTML 양식의 필드 이름을 제공하는 문자열 `옵션`: `다중 옵션` 유형의 선택
- 적 객체입니다. 이 객체는 다중 생성자에서 사용하는 것과 동일한 객체입니다(자세한 내용은 여기를 참조하세요).

> 경고 `FileInterceptor()`는 Google FireBase 등의 타사 클라우드 제공업체와 호환되지 않을 수 있습니다.

파일 유효성 검사

종종 파일 크기나 파일 밈 유형과 같은 수신 파일 메타데이터의 유효성을 검사하는 것이 유용할 수 있습니다. 이를 위해 자신만의 파이프를 만들고 `UploadedFile` 데코레이터로 주석이 달린 매개변수에 바인딩할 수 있습니다. 아래 예시는 기본 파일 크기 유효성 검사기 파이프를 구현하는 방법을 보여줍니다:

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

@Injectable()
export class FileSizeValidationPipe implements PipeTransform {
  transform(value: any, metadata: ArgumentMetadata) {
    // "value"는 파일의 속성과 메타데이터를 포함하는 객체 const oneKb = 1000입니다;
    반환 값.크기 < 1Kb;
  }
}
```

Nest는 일반적인 사용 사례를 처리하고 새로운 사용 사례의 추가를 용이하게/표준화하기 위한 내장 파이프를 제공합니다. 이 파이프는 `ParseFilePipe`라고 불리며 다음과 같이 사용할 수 있습니다:

```
@Post('file')
uploadFileAndPassValidation(
  Body() 본문: SampleDto,
  @UploadedFile(
    새로운 ParseFilePipe({ 유효성
      검사기: [
        // ... 여기에 파일 유효성 검사기 인스턴스 집합
      ]
    })
  )
  파일에 저장합니다: Express.Multer.File,
) {
  반환 { body,
    파일: file.buffer.toString(),
  };
}
```

보시다시피, `ParseFilePipe에서` 실행할 파일 유효성 검사기 배열을 지정해야 합니다. 유효성 검사기의 인터페이스에 대해서는 나중에 설명하겠지만, 이 파이프에는 두 가지 추가 옵션도 있다는 점을 언급할 필요가 있습니다:

`errorHttpStatusCode` 유효성 검사기가 실패할 경우 throw할 HTTP 상태 코드입니다. 기본값은 `400`(잘못된 요청)

예외 팩토리	오류 메시지를 수신하고 오류를 반환하는 팩토리입니다.

이제 `FileValidator` 인터페이스로 돌아갑니다. 유효성 검사기를 이 파이프와 통합하려면 기본 제공 구현을 사용하거나 사용자 지정 `FileValidator`를 제공해야 합니다. 아래 예시를 참조하세요:

```
내보내기 추상 클래스 FileValidator<TValidationOptions = Record<string, any>>
{
  생성자(보호된 읽기 전용 유효성 검사 옵션: TValidationOptions) {}

  /**
   * 생성자에서 전달된 옵션에 따라 이 파일을 유효한 파일로 간주할지 여부를 나타냅니다.
   * 매개변수 요청 객체의 파일을 파일로 저장합니다.
   */
  추상 isValid(file?: any): boolean | Promise<boolean>;

  /**
   * 유효성 검사에 실패할 경우 오류 메시지를 작성합니다.
   * 매개변수 요청 객체의 파일을 파일로 저장합니다.
   */
  추상 빌드 에러 메시지(파일: any): 문자열;
}
```

> 정보 힌트 `FileValidator` 인터페이스는 `isValid` 함수를 통해 비동기 유효성 검사를 지원합니다. 유형 보안을 활용하려면 Express(기본값)를 드라이버로 사용하는 경우 `파일` 매개 변수를 `Express.Multer.File`로 입력할 수도 있습니다.

FileValidator는 파일 객체에 액세스하고 클라이언트가 제공한 옵션에 따라 유효성을 검사하는 일반 클래스입니다. Nest에는 프로젝트에서 사용할 수 있는 두 가지 `FileValidator` 구현이 내장되어 있습니다:

- `MaxFileSizeValidator` - 주어진 파일의 크기가 제공된 값보다 작은지 확인합니다(`바이트`)
- `FileTypeValidator` - 지정된 파일의 밈 유형이 지정된 값과 일치하는지 확인합니다.

> 경고 경고 파일 유형을 확인하기 위해 FileTypeValidator 클래스는 멀티가 감지한 유형을 사용합니다. 기본적으로 멀티는 사용자 디바이스의 파일 확장자로부터 파일 유형을 도출합니다. 하지만 실제 파일 내용은 확인하지 않습니다. 파일의 이름을 임의의 확장자로 변경할 수 있으므로 앱에 더 안전한 솔루션이 필요한 경우 파일의 매직넘버 확인과 같은 사용자 정의 구현을 사용하는 것이 좋습니다.

앞서 언급한 `FileParsePipe`와 함께 어떻게 사용할 수 있는지 이해하기 위해 지난번 제시된 예제의 변경된 스

니펫을 사용하겠습니다:

```
업로드된 파일(
  새로운 ParseFilePipe({ 유효성 검
    사기: [
      새로운 MaxFileSizeValidator({ maxSize: 1000 }),
      새로운 FileTypeValidator({ fileType: 'image/jpeg' }),
```

```
      ],
    }),
  )
  파일에 저장합니다: Express.Multer.File,
```

> 정보 힌트 유효성 검사기 수가 크게 증가하거나 옵션이 파일을 복잡하게 만드는 경우, 이 배열을 별도의 파일에 정의한 다음 `fileValidators와` 같은 명명된 상수로 가져올 수 있습니다.

마지막으로, 유효성 검사기를 작성하고 구성할 수 있는 특별한 `ParseFilePipeBuilder` 클래스를 사용할 수 있습니다. 아래 그림과 같이 이 클래스를 사용하면 각 유효성 검사기의 수동 인스턴스화를 피하고 옵션을 직접 전달할 수 있습니다:

```
  업로드된 파일(
    새로운 ParseFilePipeBuilder()
      .addFileTypeValidator({
        fileType: 'jpeg',
      })
      .addMaxSizeValidator({
        maxSize: 1000
      })
      .build({
        errorHttpStatusCode: HttpStatus.UNPROCESSABLE_ENTITY
      }),
  )
  파일에 저장합니다: Express.Multer.File,
```

파일 배열

파일 배열(단일 필드명으로 식별됨)을 업로드하려면 `FilesInterceptor()` 데코레이터를 사용합니다(데코레이터 이름에 복수의 파일이 있는 것을 참고하세요). 이 데코레이터는 세 개의 인수를 받습니다:

- `fieldName`: 위에 설명된 대로
- `최대` 개수: 허용할 최대 파일 수를 정의하는 선택적 숫자입니다.
- `옵션`: 위에서 설명한 대로 선택적 `다중 옵션` 객체입니다.

`FilesInterceptor()를` 사용하는 경우, `@UploadedFiles()`를 사용하여 `요청에서` 파일을 추출합니다. 데코레이터.

```
@@파일명()
@Post('upload')
사용 인터셉터(FilesInterceptor('files')) 업로드 파일(@UploadedFiles() 파일:
Array<Express.Multer.File>) {
  콘솔 로그(파일);
}
스위치 @포스트('업로드')
사용 인터셉터(FilesInterceptor('files')) 바인드(업로드된 파일
())
```

```
uploadFile(files) {
  console.log(files);
}
```

> 정보 힌트 `FilesInterceptor()` 데코레이터는 `@nestjs/platform- express` 패키지에서 내보냅니다. `업로드된 파일()` 데코레이터는 `@nestjs/common에서` 내보냅니다.

여러 파일

여러 파일(모두 필드 이름 키가 다른)을 업로드하려면 `FileFieldsInterceptor()` 데코레이터. 이 데코레이터는 두 개의 인자를 받습니다:

- `uploadedFields`: 객체 배열로, 각 객체는 위에서 설명한 대로 필드 이름을 지정하는 문자열 값과 함께 필수 `이름` 속성을 지정하고 위에서 설명한 대로 선택적 `maxCount` 속성을 지정합니다.
- `옵션`: 위에서 설명한 대로 선택적 `다중 옵션` 객체입니다.

FileFieldsInterceptor`()를` 사용하는 경우, `@UploadedFiles()`를 사용하여 `요청에서` 파일을 추출합니다. 데코레이터.

```
@@파일명()
@Post('upload')
@UseInterceptors(FileFieldsInterceptor([
  { name: 'avatar', maxCount: 1 },
  { name: 'background', maxCount: 1 },
]))
업로드 파일(@UploadedFiles() 파일: { avatar?: Express.Multer.File[],
background?: Express.Multer.File[] }) {
  콘솔 로그(파일);
}
스위치 @Post('업로드')
@Bind(업로드된파일())
@UseInterceptors(FileFieldsInterceptor([
  { name: 'avatar', maxCount: 1 },
  { name: 'background', maxCount: 1 },
]))
uploadFile(files) {
  console.log(files);
}
```

모든 파일

임의의 필드 이름 키가 있는 모든 필드를 업로드하려면 `AnyFilesInterceptor()` 데코레이터를 사용합니다. 이 데코레이터는 위에서 설명한 대로 선택적 `옵션` 객체를 받을 수 있습니다.

`AnyFilesInterceptor()를` 사용하는 경우, `@UploadedFiles()`를 사용하여 `요청에서` 파일을 추출합니다. 데코레이터.

```
@@파일명()
@Post('upload')
사용 인터셉터(AnyFilesInterceptor()) 업로드 파일(@UploadedFiles() 파일:
Array<Express.Multer.File>) {
  콘솔 로그(파일);
}
스위치 @Post('업로드')
@Bind(업로드된파일())
UseInterceptors(AnyFilesInterceptor())
uploadFile(files) {
  콘솔 로그(파일);
}
```

파일 없음

`멀티파트/양식` 데이터는 허용하지만 파일 업로드를 허용하지 않으려면 `NoFilesInterceptor를` 사용하세요. 이렇게 하면 요청 본문에 멀티파트 데이터를 속성으로 설정합니다. 요청과 함께 전송된 모든 파일은 `BadRequestException을` 던집니다.

```
Post('upload')
@UseInterceptors(NoFilesInterceptor())
handleMultiPartData(@Body() body) {
  콘솔 로그(본문)
}
```

기본 옵션

위에서 설명한 대로 파일 인터셉터에서 멀티 옵션을 지정할 수 있습니다. 기본 옵션을 설정하려면, `멀티모듈을` 임포트할 때 정적 `register()` 메서드를 호출하여 지원되는 옵션을 전달하면 됩니다. 여기에 나열된 모든 옵션을 사용할 수 있습니다.

```
Multimodule.register({
  dest: './upload',
});
```

> 정보 힌트 `멀티모듈` 클래스는 `@nestjs/플랫폼-익스프레스` 패키지에서 내보냅니다.

비동기 구성

`멀티모듈` 옵션을 정적이 아닌 비동기적으로 설정해야 하는 경우 `registerAsync()` 메서드를 사용하세요. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성을 처리하는 몇 가지 기술을 제공합니다.

한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
Multimodule.registerAsync({
  useFactory: () => ({
    목적지: './업로드',
  }),
});
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 비동기화될 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.
주입합니다.

```
MulterModule.registerAsync({
  import: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    dest: configService.get<string>('MULTER_DEST'),
  }),
  주입합니다: [구성 서비스],
});
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 멀티모듈을 구성할 수도 있습니다:

```
MulterModule.registerAsync({
  useClass: MulterConfigService,
});
```

위의 구조는 MulterModule 내부에 MulterConfigService를 인스턴스화하여 이를 사용하여 필요한 옵션 객체를 생성합니다. 이 예제에서 MulterConfigService는 아래와 같이 MulterOptionsFactory 인터페이스를 구현해야 한다는 점에 유의하세요. 제공된 클래스의 인스턴스화된 객체에서 MulterModule이 createMulterOptions() 메서드를 호출합니다.

```
@Injectable()
MulterConfigService 클래스는 MulterOptionsFactory를 구현합니다 {
  createMulterOptions(): MulterModuleOptions {
    반환 {
      목적지: './업로드',
    };
  }
}
```

내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려는 경우
다중 모듈을 사용하려면 useExisting 구문을 사용합니다.

```
MulterModule.registerAsync({
  import: [ConfigModule],
```

```
    사용Existing: ConfigService,
  });
```

예

작동하는 예제는 여기에서 확인할 수 있습니다.

스트리밍 파일

> 정보 참고 이 장에서는 HTTP 애플리케이션에서 파일을 스트리밍하는 방법을 설명합니다. 예제 아래에 제시된 내용은 GraphQL 또는 마이크로서비스 애플리케이션에는 적용되지 않습니다.

REST API에서 클라이언트로 파일을 다시 보내고 싶을 때가 있을 수 있습니다. Nest에서 이 작업을 수행하려면 일반적으로 다음을 수행합니다:

```
컨트롤러('파일')
내보내기 클래스 FileController {
  @Get()
  getFile(@Res() res: Response) {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    file.pipe(res);
  }
}
```

하지만 이렇게 하면 컨트롤러 이후 인터셉터 로직에 액세스할 수 없게 됩니다. 이를 처리하기 위해 `StreamableFile` 인스턴스를 반환하면 프레임워크가 내부에서 응답을 파이핑하는 작업을 처리합니다.

스트리밍 가능한 파일 클래스

StreamableFile은 반환할 스트림을 보유하는 클래스입니다. 새로운 `StreamableFile` 생성자에 `버퍼` 또는 `스트림을` 전달할 수 있습니다.

> 정보 힌트 `StreamableFile` 클래스는 `@nestjs/common에서` 가져올 수 있습니다.

크로스 플랫폼 지원

Fastify는 기본적으로 `stream.pipe(res)`를 호출할 필요 없이 파일 전송을 지원할 수 있으므로 `StreamableFile` 클래스를 전혀 사용할 필요가 없습니다. 그러나 Nest는 두 플랫폼 유형 모두에서 `StreamableFile` 사용을 지원하므로 Express와 Fastify를 전환하는 경우 두 엔진 간의 호환성에 대해 걱정할 필요가 없습니다.

예

아래에서 `패키지.json`을 JSON 대신 파일로 반환하는 간단한 예제를 찾을 수 있지만 이 아이디어는 이미지, 문서 및 기타 모든 파일 유형으로 자연스럽게 확장됩니다.

```
'@nestjs/common'에서 { Controller, Get, StreamableFile }을 임포트하고,
'fs'에서 { createReadStream }을 임포트합니다;
'경로'에서 { join }을 가져옵니다;

컨트롤러('파일')
내보내기 클래스 FileController {
  @Get()
```

```
  getFile(): StreamableFile {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    return new StreamableFile(file);
  }
}
```

기본 콘텐츠 유형은 애플리케이션/옥텟 스트림이며, 응답을 사용자 정의해야 하는 경우 다음과 같이 res.set 메서드 또는 @Header() 데코레이터를 사용할 수 있습니다:

```
'@nestjs/common'에서 { Controller, Get, StreamableFile, Res }를 임포트하고,
'fs'에서 { createReadStream }을 임포트합니다;
'경로'에서 { join }을 가져옵니다;
'express'에서 { Response } 유형을 가져옵니다;

컨트롤러('파일')
내보내기 클래스 FileController {
  @Get()
  getFile(@Res({ 패스스루: true }) res: Response): StreamableFile { const
    file = createReadStream(join(process.cwd(), 'package.json'));
    res.set({
      '콘텐츠 유형': '애플리케이션/json',
      '콘텐츠-처분': '첨부파일; 파일명="package.json"',
    });
    새로운 StreamableFile(file)을 반환합니다;
  }

  // 또는 짝수:
  @Get()
  헤더('콘텐츠 유형', '애플리케이션/json')
  @Header('Content-Disposition', 'attachment; filename="package.json"')
  getStaticFile(): StreamableFile {
    const file = createReadStream(join(process.cwd(), 'package.json'));
    return new StreamableFile(file);
  }
}
```

HTTP 모듈

Axios는 널리 사용되는 풍부한 기능을 갖춘 HTTP 클라이언트 패키지입니다. Nest는 Axios를 래핑하여 내장된 `HttpModule을` 통해 노출합니다. HttpModule은 HTTP 요청을 수행하기 위한 Axios 기반 메서드를 노출하는 `HttpService` 클래스를 내보냅니다. 또한 라이브러리는 결과 HTTP 응답을 `Observable로` 변환합니다.

> 정보 힌트 got 또는 undici를 포함한 모든 범용 Node.js HTTP 클라이언트 라이브러리를 직접 사용할 수도 있습니다.

설치

사용을 시작하려면 먼저 필수 종속성을 설치합니다.

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

시작하기

설치 프로세스가 완료되면 `HttpService를` 사용하려면 먼저 `HttpModule을` 가져옵니다.

```
모듈({
  수입: [HttpModule], 공급자:
  [CatsService],
})
내보내기 클래스 CatsModule {}
```

다음으로 일반 생성자 주입을 사용하여 `HttpService를` 주입합니다.

> 정보 힌트 HttpModule과 HttpService는 `@nestjs/axios` 패키지에서 가져옵니다.

```
@@파일명()
@Injectable()
내보내기 클래스 CatsService {
  생성자(비공개 읽기 전용 httpService: HttpService) {}

  findAll(): Observable<AxiosResponse<Cat[]>> {
    이.httpService.get('http://localhost:3000/cats')을 반환합니다;
  }
}
@@스위치 @Injectable()
@Dependencies(HttpService)
내보내기 클래스 CatsService {
  constructor(httpService) {
    this.httpService = httpService;
  }
```

```
  findAll() {
    이.httpService.get('http://localhost:3000/cats')을 반환합니다;
  }
}
```

> 정보 힌트 AxiosResponse는 `axios` 패키지에서 내보낸 인터페이스입니다(`$ npm i axios`).

모든 `HttpService` 메서드는 `관찰 가능한` 객체로 래핑된 `AxiosResponse를` 반환합니다.

구성

Axios는 다양한 옵션으로 구성하여 `HttpService의` 동작을 사용자 정의할 수 있습니다. Read
자세한 내용은 여기를 참조하세요. 기본 Axios 인스턴스를 구성하려면 인스턴스를 가져올 때 선택적 옵션 객체를
`HttpModule의 register()` 메서드에 전달합니다. 이 옵션 객체는 기본 Axios 생성자에게 직접 전달됩니다.

```
모듈({ import: [
    HttpModule.register({
      timeout: 5000,
      최대 리디렉션: 5,
    }),
  ],
  공급자: [CatsService],
})
내보내기 클래스 CatsModule {}
```

비동기 구성

모듈 옵션을 정적이 아닌 비동기적으로 전달해야 하는 경우, `registerAsync()`
메서드를 사용합니다. 대부분의 동적 모듈과 마찬가지로 Nest는 비동기 구성을 처리하는 몇 가지 기술을 제공합니

다. 한 가지 기법은 팩토리 함수를 사용하는 것입니다:

```
HttpModule.registerAsync({
  useFactory: () => ({
    시간 초과: 5000,
    최대 리디렉션: 5,
  }),
});
```

다른 팩토리 공급자와 마찬가지로 팩토리 함수는 비동기화될 수 있으며 다음을 통해 종속성을 주입할 수 있습니다.

주입합니다.

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useFactory: async (configService: ConfigService) => ({
    timeout: configService.get('HTTP_TIMEOUT'),
    maxRedirects: configService.get('HTTP_MAX_REDIRECTS'),
  }),
  주입합니다: [구성 서비스],
});
```

또는 아래와 같이 팩토리 대신 클래스를 사용하여 HttpModule을 구성할 수도 있습니다.

```
HttpModule.registerAsync({
  useClass: HttpConfigService,
});
```

위의 구조는 `HttpModule` 내부에 `HttpConfigService를` 인스턴스화하여 이를 사용하여 옵션 객체를 생성합니다. 이 예제에서 `HttpConfigService는` 아래와 같이 `HttpModuleOptionsFactory` 인터페이스를 구현해야 한다는 점에 유의하세요. `HttpModule은` 제공된 클래스의 인스턴스화된 객체에서 `createHttpOptions()` 메서드를 호출합니다.

```
@Injectable()
HttpConfigService 클래스는 HttpModuleOptionsFactory를 구현합니다 {
  createHttpOptions(): HttpModuleOptions {
    반환 { 시간 초과:
      5000,
      최대 리디렉션: 5,
    };
  }
}
```

`HttpModule` 내부에 비공개 복사본을 만드는 대신 기존 옵션 공급자를 재사용하려면 `useExisting` 구문을 사용하세요.

```
HttpModule.registerAsync({
  imports: [ConfigModule],
  useExisting: HttpConfigService,
});
```

Axios 직접 사용

`HttpModule.`register의 옵션이 충분하지 않다고 생각되거나 `@nestjs/axios`에서 생성한 기본 Axios 인스턴스에만 액세스하려는 경우, 다음과 같이 `HttpService#axiosRef`를 통해 액세스할 수 있습니다:

```
  @Injectable()
  내보내기 클래스 CatsService {
    생성자(비공개 읽기 전용 httpService: HttpService) {}

    findAll(): Promise<AxiosResponse<Cat[]>> {
      이.httpService.axiosRef.get('http://localhost:3000/cats')을 반환합니다;
                                //^ AxiosInstance 인터페이스
    }
  }
```

전체 예제

`HttpService` 메서드의 반환값이 Observable이기 때문에, rxjs를 사용할 수 있습니다.

`첫 번째 값` 또는 `마지막` 값에서 요청의 데이터를 프로미스 형태로 검색합니다.

```
  'rxjs'에서 { catchError, firstValueFrom } import;

  @Injectable()
  내보내기 클래스 CatsService {
    private readonly logger = new Logger(CatsService.name);
    constructor(private readonly httpService: HttpService) {}

    비동기 findAll(): Promise<Cat[]> {
      const { data } = await firstValueFrom(
        this.httpService.get<Cat[]>('http://localhost:3000/cats').pipe(
          catchError((error: AxiosError) => {
            this.logger.error(error.response.data);
            throw '오류가 발생했습니다!';
          }),
        ),
      );
      데이터를 반환합니다;
    }
  }
```

> 정보 힌트 `첫 번째` 값과 `마지막` 값의 차이점은 RxJS의 `첫 번째 값과 마지막 값에` 대한 설명서를 참조하세요.

세션

HTTP 세션은 여러 요청에 걸쳐 사용자에 대한 정보를 저장할 수 있는 방법을 제공하며, 특히 MVC 애플리케이션에 유용합니다.

Express와 함께 사용(기본값)

먼저 필요한 패키지(TypeScript 사용자의 경우 해당 유형)를 설치합니다:

```
$ npm i express-session
$ npm i -D @types/express-session
```

설치가 완료되면 익스프레스 세션 미들웨어를 전역 미들웨어로 적용합니다(예: main.ts 파일에).

```
'express-session'에서 *를 세션으로 가져옵니다;

// 초기화 파일 어딘가에 앱.사용(
  세션({
    secret: 'my-secret',
    resave: false,
    saveUnitialized: false,
  }),
);
```

> 경고 기본 서버 측 세션 저장소는 의도적으로 프로덕션 환경을 위해 설계되지 않았습니다. 대부분의 조건에서 메모리가 누수되고 단일 프로세스를 초과하여 확장되지 않으며 디버깅 및 개발용입니다. 공식 리포지토리에서 자세히 알아보세요.

비밀은 세션 ID 쿠키에 서명하는 데 사용됩니다. 이는 단일 비밀에 대한 문자열이거나 여러 비밀의 배열일 수 있습니다. 시크릿 배열이 제공되면 첫 번째 요소만 세션 ID 쿠키에 서명하는 데 사용되며, 요청에서 서명을 확인할 때 모든 요소가 고려됩니다. 시크릿 자체는 사람이 쉽게 파싱할 수 없어야 하며 임의의 문자 집합을 사용하는 것이 가장 좋습니다.

다시 저장 옵션을 활성화하면 요청 중에 세션을 수정하지 않은 경우에도 세션이 세션 저장소에 강제로 저장됩니다. 기본값은 true이지만 기본값은 향후 변경될 예정이므로 기본값을 사용하는 것은 더 이상 권장되지 않습니다.

마찬가지로 `저장` 초기화 옵션을 활성화하면 "초기화되지 않은" 세션이 스토어에 강제로 저장됩니다. 세션은 새 세션이지만 수정되지 않은 경우 초기화되지 않습니다. `false를` 선택하면 로그인 세션을 구현하거나 서버 스토리지 사용량을 줄이거나 쿠키를 설정하기 전에 허가가 필요한 법률을 준수하는 데 유용합니다. `false를` 선택하면 클라이언트가 세션(소스) 없이 여러 개의 병렬 요청을 하는 경쟁 조건에도 도움이 됩니다.

`세션` 미들웨어에 몇 가지 다른 옵션을 전달할 수 있으며, 이에 대한 자세한 내용은 API 설명서를 참조하세요.

> 정보 힌트 `보안: 참이` 권장 옵션입니다. 단, 보안 쿠키를 사용하려면 HTTPS가 활성화된 웹사이트가 필요합니다. 보안이 설정되어 있고 HTTP를 통해 사이트에 액세스하면 쿠키가 설정되지 않습니다. 프록시 뒤에 node.js가 있고 `보안: true를` 사용하는 경우 익스프레스에서 `"신뢰 프록시"를` 설정해야 합니다.

이제 다음과 같이 라우트 핸들러 내에서 세션 값을 설정하고 읽을 수 있습니다:

```
@Get()
findAll(@Req() request: 요청) {
  요청.세션.방문 = 요청.세션.방문 ? 요청.세션.방문
+ 1 : 1;
}
```

> 정보 힌트 `@Req()` 데코레이터는 @nestjs/common에서 가져온 것이고, `요청은 익스프레스` 패키지.

또는 다음과 같이 `@Session()` 데코레이터를 사용하여 요청에서 세션 객체를 추출할 수 있습니다:

```
@Get()
findAll(@Session() 세션: Record<string, any>) { session.visits =
  세션.방문 ? 세션.방문 + 1 : 1;
}
```

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

Fastify와 함께 사용

먼저 필요한 패키지를 설치합니다:

```
$ npm i @fastify/secure-session
```

설치가 완료되면 `fastify-secure-session` 플러그인을 등록합니다:

```
'@fastify/secure-session'에서 secureSession을 가져옵니다;

// 초기화 파일 어딘가에
const app = await NestFactory.create<NestFastifyApplication>(
  AppModule,
  새로운 FastifyAdapter(),
);
await app.register(secureSession, {
  비밀: 'averylogphrasebiggerthanthirtytwochars', 소금:
  'mq9hDxBVDbspDR6n',
});
```

> 정보 힌트 키를 미리 생성하거나(지침 참조) 키 회전을 사용할 수도 있습니다.

공식 리포지토리에서 사용 가능한 옵션에 대해 자세히 알아보세요.

이제 다음과 같이 라우트 핸들러 내에서 세션 값을 설정하고 읽을 수 있습니다:

```
@Get()
findAll(@Req() 요청: FastifyRequest) { const
  visits = request.session.get('visits');
  요청.세션.set('방문', 방문 ? 방문 + 1 : 1);
}
```

또는 다음과 같이 `@Session()` 데코레이터를 사용하여 요청에서 세션 객체를 추출할 수 있습니다:

```
@Get()
findAll(@Session() session: secureSession.Session) {
  const visits = session.get('visits');
  session.set('visits', visits ? visits + 1 : 1);
}
```

> 정보 힌트 `@Session()` 데코레이터는 `@nestjs/common`에서, `secureSession.Session`은 `@fastify/sure-session` 패키지에서 가져옵니다(가져오기 문: `'@fastify/sure-session'`에서 `*`를 `secureSession`으로 가져오기).

모델-보기-컨트롤러

Nest는 기본적으로 Express 라이브러리를 내부적으로 사용합니다. 따라서 Express에서 MVC(모델-뷰-컨트롤러) 패턴을 사용하는 모든 기술은 Nest에도 적용됩니다.

먼저 CLI 도구를 사용하여 간단한 Nest 애플리케이션을 스캐폴딩해 보겠습니다:

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

MVC 앱을 만들려면 HTML 뷰를 렌더링할 템플릿 엔진도 필요합니다:

```
$ npm install --save hbs
```

여기서는 `hbs`(핸들바) 엔진을 사용했지만 요구 사항에 맞는 엔진을 사용할 수 있습니다. 설치 프로세스가 완료되면 다음 코드를 사용하여 익스프레스 인스턴스를 구성해야 합니다:

```
@@파일명(메인)
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/platform-express'에서 { NestExpressApplication } 임포트; '경로'
에서 { join } 임포트;
'./app.module'에서 { AppModule }을 가져옵니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.create<NestExpressApplication>(
    AppModule,
  );

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
  app.setViewEngine('hbs');

  await app.listen(3000);
}
부트스트랩(); @@스
위치
'@nestjs/core'에서 { NestFactory }를 가져오고,
'path'에서 { join }를 가져옵니다;
'./app.module'에서 { AppModule }을 가져옵니다;

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

  app.useStaticAssets(join(__dirname, '..', 'public'));
  app.setBaseViewsDir(join(__dirname, '..', 'views'));
```

```
  app.setViewEngine('hbs');

  await app.listen(3000);
}
부트스트랩();
```

공용 디렉토리는 정적 에셋을 저장하는 데 사용되고, 뷰에는 템플릿이 포함되며, HTML 출력을 렌더링하는 데는 hbs 템플릿 엔진을 사용해야 한다고 Express에 설명했습니다.

템플릿 렌더링

이제 뷰 디렉터리와 그 안에 index.hbs 템플릿을 만들어 보겠습니다. 템플릿에서 메시지를 전달합니다:

```
<!DOCTYPE html>
<html>
  <head>
    <meta-charset="utf-8" />
    <title>앱</title>
  </head>
  <body>
    {{ "{{ 메시지 }\}" }}
  </body>
</html>
```

그런 다음 app.controller 파일을 열고 root() 메서드를 다음 코드로 바꿉니다:

```
@@파일명(앱.컨트롤러)
'@nestjs/common'에서 { Get, Controller, Render }를 가져옵니다;

컨트롤러()
내보내기 클래스 AppController {
  @Get()
  @Render('index')
  root() {
    반환 { 메시지: '안녕하세요!' };
  }
}
```

이 코드에서는 @Render() 데코레이터에서 사용할 템플릿을 지정하고 있으며, 라우트 핸들러 메서드의 반환값은 렌더링을 위해 템플릿으로 전달됩니다. 반환 값은 템플릿에서 생성한 메시지 자리 표시자와 일치하는 속성 메

`시지가` 있는 객체입니다.

애플리케이션이 실행되는 동안 브라우저를 열고 `http://localhost:3000` 으로 이동합니다. `안녕하세요!` 메시지가 표시될 것입니다.

동적 템플릿 렌더링

애플리케이션 로직이 렌더링할 템플릿을 동적으로 결정해야 하는 경우, `@Res()`를 사용해야 합니다.

데코레이터를 사용하고 `@Render()` 데코레이터가 아닌 경로 핸들러에 뷰 이름을 지정합니다:

> 정보 힌트 Nest가 `@Res()` 데코레이터를 감지하면 라이브러리별 `응답` 객체를 삽입합니다. 이 객체를 사용하여 템플릿을 동적으로 렌더링할 수 있습니다. 여기에서 `응답` 객체 API에 대해 자세히 알아보세요.

```
@@파일명(앱.컨트롤러)
'@nestjs/common'에서 { Get, Controller, Res, Render }를 가져오고,
'express'에서 { Response }를 가져옵니다;
'./app.service'에서 { AppService }를 가져옵니다;

컨트롤러()
내보내기 클래스 AppController {
  constructor(private appService: AppService) {}

  @Get()
  root(@Res() res: Response) {
    return res.render(
      이.앱서비스.getViewName(),
      { 메시지: 'Hello world!' },
    );
  }
}
```

예

작동 예제는 여기에서 확인할 수 있습

니다. Fastify

이 장에서 언급했듯이 호환 가능한 모든 HTTP 공급자를 Nest와 함께 사용할 수 있습니다. One
이러한 라이브러리가 바로 Fastify입니다. Fastify를 사용하여 MVC 애플리케이션을 생성하려면 다음 패키지를 설치해야 합니다:

```
$ npm i --save @fastify/static @fastify/view 핸들바 저장
```

다음 단계는 플랫폼에 따라 약간의 차이가 있지만 Express에서 사용되는 프로세스와 거의 동일합니다. 설치 프로

세스가 완료되면 `main.ts` 파일을 열고 내용을 업데이트합니다:

```
@@파일명(메인)
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/platform- fastify'에서 { NestFastifyApplication, FastifyAdapter }를 가
져옵니다;
'./app.module'에서 { AppModule }을 임포트하고, '경
로'에서 { join }을 임포트합니다;
```

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    새로운 FastifyAdapter(),
  );
  app.useStaticAssets({
    root: join(__dirname, '..', 'public'),
    접두사: '/public/',
  });
  app.setViewEngine({ 엔
    진: {
      핸들바: require('핸들바'),
    },
    템플릿: 조인(________dirname, '..', 'views'),
  });
  await app.listen(3000);
}
부트스트랩(); @@스
위치
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/platform-fastify'에서 { FastifyAdapter }를 임포트하고,
'./app.module'에서 { AppModule }을 임포트합니다;
'경로'에서 { join }을 가져옵니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.create(AppModule, new FastifyAdapter());
  app.useStaticAssets({
    root: join(__dirname, '..', 'public'),
    접두사: '/public/',
  });
  app.setViewEngine({ 엔
    진: {
      핸들바: require('핸들바'),
    },
    템플릿: 조인(________dirname, '..', 'views'),
  });
  await app.listen(3000);
}
부트스트랩();
```

Fastify API는 약간 다르지만 메서드 호출의 최종 결과는 동일하게 유지됩니다. Fastify의 한 가지 차이점은 `@Render()` 데코레이터에 전달되는 템플릿 이름에 파일 확장자가 포함되어야 한다는 점입니다.

```
@@파일명(앱.컨트롤러)
'@nestjs/common'에서 { Get, Controller, Render }를 가져옵니다;

컨트롤러()
export class AppController
  { @Get()
  @Render('index.hbs')
```

```
  root() {
    반환 { 메시지: '안녕하세요!' };
  }
}
```

애플리케이션이 실행되는 동안 브라우저를 열고 `http://localhost:3000` 으로 이동합니다. `안녕하세요!` 메시지가 표시될 것입니다.

예

작동하는 예제는 여기에서 확인할 수 있습니다.

성능(Fastify)

기본적으로 Nest는 Express 프레임워크를 사용합니다. 앞서 언급했듯이 Nest는 Fastify와 같은 다른 라이브러리와의 호환성도 제공합니다. Nest는 미들웨어와 핸들러를 적절한 라이브러리별 구현으로 프록시하는 것을 주요 기능으로 하는 프레임워크 어댑터를 구현함으로써 이러한 프레임워크 독립성을 달성합니다.

> 정보 힌트 프레임워크 어댑터를 구현하려면 대상 라이브러리가 Express에서와 유사한 요청/응답 파이프라인 처리 기능을 제공해야 합니다.

Fastify는 Express와 비슷한 방식으로 디자인 문제를 해결하기 때문에 Nest를 위한 좋은 대체 프레임워크를 제공합니다. 그러나 Fastify는 Express보다 훨씬 빠르며 벤치마크 결과도 거의 두 배 더 우수합니다. 그렇다면 Nest가 기본 HTTP 공급자로 Express를 사용하는 이유는 무엇일까요? 그 이유는 Express가 널리 사용되고 잘 알려져 있으며 Nest 사용자가 즉시 사용할 수 있는 호환 가능한 미들웨어 세트가 방대하기 때문입니다.

그러나 Nest는 프레임워크 독립성을 제공하므로 두 프레임워크 간에 쉽게 마이그레이션할 수 있습니다. 매우 빠른 성능을 중시하는 경우 Fastify가 더 나은 선택이 될 수 있습니다. Fastify를 사용하려면 이 챕터에 표시된 대로 기본 제공되는 `FastifyAdapter`를 선택하기만 하면 됩니다.

설치

먼저 필요한 패키지를 설치해야 합니다:

```
npm i --save @nestjs/platform-fastify
```

어댑터

Fastify 플랫폼이 설치되면 `FastifyAdapter`를 사용할 수 있습니다.

```
@@파일명(메인)
'@nestjs/core'에서 { NestFactory }를 가져옵니다.
  FastifyAdapter,
  NestFastifyApplication,
}를 '@nestjs/platform-fastify'에서 가져옵니다;
'./app.module'에서 { AppModule }을 가져옵니다;

비동기 함수 부트스트랩() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    새로운 FastifyAdapter()
  );
  await app.listen(3000);
}
부트스트랩();
```

기본적으로 Fastify는 `localhost 127.0.0.1` 인터페이스에서만 수신 대기합니다(자세히 보기). 다른 호스트에서 연결을 수락하려면 `listen()` 호출에 `'0.0.0.0'을` 지정해야 합니다:

```
비동기 함수 부트스트랩() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    새로운 FastifyAdapter(),
  );
  await app.listen(3000, '0.0.0.0');
}
```

플랫폼별 패키지

`FastifyAdapter를` 사용할 때 Nest는 HTTP 공급자로 Fastify를 사용한다는 점에 유의하세요. 즉, Express에 의존하는 각 레시피가 더 이상 작동하지 않을 수 있습니다. 대신 Fastify와 동등한 패키지를 사용해야 합니다.

응답 리디렉션

Fastify는 리디렉션 응답을 Express와 약간 다르게 처리합니다. Fastify로 올바른 리디렉션을 수행하려면 다음과 같이 상태 코드와 URL을 모두 반환하세요:

```
@Get()
index(@Res() res) {
  res.status(302).redirect('/login');
}
```

단축 옵션

`FastifyAdapter` 생성자를 통해 Fastify 생성자에 옵션을 전달할 수 있습니다. 예를 들어

```
새로운 FastifyAdapter({ logger: true });
```

미들웨어

미들웨어 함수는 Fastify의 래퍼 대신 원시 `요청` 및 `res` 객체를 검색합니다. 이것이 `미디` 패키지가 작동하는 방식(내부에서 사용되는 방식)이며, 자세한 내용은 이 페이지를 참조하세요,

```
@@파일명(logger.middleware)
'@nestjs/common'에서 { Injectable, NestMiddleware }를 임포트하고,
'fastify'에서 { FastifyRequest, FastifyReply }를 임포트합니다;

@Injectable()
```

```
내보내기 클래스 LoggerMiddleware는 NestMiddleware를 구현합니다 {
  use(req: FastifyRequest['raw'], res: FastifyReply['raw'], next: () =>
void) {
    console.log('요청...'); next();
  }
}
@@switch
'@nestjs/common'에서 { Injectable }을 임포트합니다;

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

경로 구성

Fastify의 경로 구성 기능을 `@RouteConfig()` 데코레이터와 함께 사용할 수 있습니다.

```
@RouteConfig({ 출력: 'hello world' })
@Get()
index(@Req() req) {
  req.routeConfig.output을 반환합니다;
}
```

> 정보 힌트 `@RouteConfig()`는 `@nestjs/platform-fastify`에서 가져온 것입니다.

예

작동하는 예제는 여기에서 확인할 수 있습니다.

서버 전송 이벤트

서버 전송 이벤트(SSE)는 클라이언트가 HTTP 연결을 통해 서버로부터 자동 업데이트를 수신할 수 있도록 하는 서버 푸시 기술입니다. 각 알림은 한 쌍의 개행으로 끝나는 텍스트 블록으로 전송됩니다(여기에서 자세히 알아보기).

사용법

라우트(컨트롤러 클래스 내에 등록된 라우트)에서 서버 전송 이벤트를 활성화하려면 메서드 핸들러에 `@Sse()` 데코레이터를 주석으로 추가합니다.

```
@Sse('sse')
sse(): Observable<MessageEvent> {
  반환 간격(1000).pipe(map((_) => ({ 데이터: { hello: 'world' } })));
}
```

> 정보 힌트 `@Sse()` 데코레이터와 `메시지 이벤트` 인터페이스는 `에서` 가져오고, `관찰 가능`, `간격` 및 지도는 `rxjs` 패키지에서 가져옵니다.

> 경고 서버에서 보낸 이벤트 경로는 `관찰 가능한` 스트림을 반환해야 합니다.

위의 예제에서는 실시간 업데이트를 전파할 수 있는 `sse라는` 경로를 정의했습니다. 이러한 이벤트는 EventSource API를 사용하여 수신할 수 있습니다.

`sse` 메서드는 여러 `MessageEvent를` 방출하는 `Observable을` 반환합니다(이 예제에서는 매초마다 새로운 `MessageEvent를` 방출합니다). `MessageEvent` 객체는 사양과 일치하도록 다음 인터페이스를 준수해야 합니다:

```
내보내기 인터페이스 MessageEvent { 데이
  터: 문자열 | 객체;
  id?: 문자열; 유형?:
  문자열; 재시도?: 숫자
  ;
}
```

이제 클라이언트 측 애플리케이션에서 `이벤트 소스` 클래스의 인스턴스를 생성할 수 있으며, 생성자 인수로 (위에서 `@Sse()` 데코레이터에 전달한 엔드포인트와 일치하는) `/sse` 경로를 전달할 수 있습니다.

`EventSource` 인스턴스는 `텍스트/이벤트 스트림` 형식으로 이벤트를 전송하는 HTTP 서버에 대한 영구 연결을 엽니다. 연결은 `EventSource.close()`를 호출하여 닫을 때까지 열린 상태로 유지됩니다.

연결이 열리면 서버에서 들어오는 메시지가 이벤트 형태로 코드에 전달됩니다. 수신 메시지에 이벤트 필드가 있는 경우 트리거된 이벤트는 이벤트 필드 값과 동일합니다. 이벤트 필드가 없는 경우 일반 `메시지` 이벤트가 발생합니다(소스).

```
const eventSource = new EventSource('/sse');
eventSource.onmessage = ({ data }) => {
  console.log('새 메시지', JSON.parse(data));
};
```

예

작동하는 예제는 여기에서 확인할 수 있습니다.