소개

OpenAPI 사양은 언어에 구애받지 않는 정의 형식으로 RESTful API를 설명하는 데 사용됩니다. Nest는 데코레이터를 활용하여 이러한 사양을 생성할 수 있는 전용 모듈을 제공합니다.

설치

사용을 시작하려면 먼저 필요한 종속성을 설치합니다.

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

부트스트랩

설치 프로세스가 완료되면 `main.ts` 파일을 열고 다음을 사용하여 Swagger를 초기화합니다. `SwaggerModule` 클래스입니다:

```
@@파일명(메인)
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/swagger'에서 { SwaggerModule, DocumentBuilder }를 임포트하고,
'./app.module'에서 { AppModule }을 임포트합니다;

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

  const config = new DocumentBuilder()
    .setTitle('고양이 예시')
    .setDescription('고양이 API 설명')
    .setVersion('1.0')
    .addTag('cats')
    .build();
  const document = SwaggerModule.createDocument(app, config);
  SwaggerModule.setup('api', app, document);

  await app.listen(3000);
}
부트스트랩();
```

> 정보 힌트 문서(`SwaggerModule#createDocument()` 메서드에서 반환)는 OpenAPI Document를 준수하는 직렬화 가능한 객체입니다. HTTP를 통해 호스팅하는 대신 JSON/YAML 파일로 저장하여 다양한 방식으로 사용할 수도 있습니다.

문서 작성기는 OpenAPI 사양을 준수하는 기본 문서를 구조화하는 데 도움이 됩니다. 제목, 설명, 버전 등과 같은 속성을 설정할 수 있는 여러 메서드를 제공합니다. 전체 문서(모든 HTTP 경로가 정의된)를 생성하려면 `SwaggerModule` 클래스의 `createDocument()` 메서드를 사용합니다. 이 메서드는 애플리케이션 인스턴스와 Swagger 옵션이라는 두 개의 인수를 받습니다.

객체입니다. 또는 세 번째 인수를 제공할 수 있는데, 이 인수의 유형은 `Swagger도큐먼트옵션`. 이에 대한 자세한 내용은 문서 옵션 섹션을 참조하세요. 문

서를 생성하고 나면 `setup()` 메서드를 호출할 수 있습니다. 수락합니다:

. 스웨거 UI를 마운트하는 경로
. 애플리케이션 인스턴스
. 위에 인스턴스화된 문서 객체
. 선택적 구성 매개변수(자세한 내용은 여기를 참조하세요)

이제 다음 명령을 실행하여 HTTP 서버를 시작할 수 있습니다:

```
$ npm 실행 시작
```

애플리케이션이 실행되는 동안 브라우저를 열고 `http://localhost:3000/api` 으로 이동합니다. Swagger UI 가 표시됩니다.

보시다시피 `SwaggerModule은` 모든 엔드포인트를 자동으로 반영합니다.

> 정보 힌트 Swagger JSON 파일을 생성하고 다운로드하려면 http://localhost:3000/api-json(`http://localhost:3000/api)`로 이동합니다(Swagger 문서가 제공된다고 가정).

> 경고 패스트파이와 헬멧을 사용할 때 CSP에 문제가 있을 수 있으며, 이 충돌을 해결하려면 아래 그림과 같이 CSP를 구성하세요:

```
app.register(helmet, {
  contentSecurityPolicy: {
    지시어를 추가합니다: {
      defaultSrc: [`'self'`],
      styleSrc: [`'self'`, `'unsafe-inline'`],
      imgSrc: [`'self'`, 'data:', 'validator.swagger.io'],
      scriptSrc: [`'self'`, `https: 'unsafe-inline'`],
    },
  },
});

// CSP를 전혀 사용하지 않을 경우 다음과 같이 사용할 수 있습니다:
app.register(helmet, {
  contentSecurityPolicy: false,
});
```

문서 옵션

문서를 만들 때 라이브러리의 동작을 미세 조정하기 위해 몇 가지 추가 옵션을 제공할 수 있습니다. 이러한 옵션은 다음과 같은 `SwaggerDocumentOptions` 유형이어야 합니다:

```
내보내기 인터페이스 SwaggerDocumentOptions {
  /**
   * 사양에 포함할 모듈 목록
   */
  include? 함수[];

  /**
   * 검사하고 사양에 포함시켜야 하는 추가 추가 모델
   */
  extraModels?: Function[];

  /**
   * true`이면 스웨거는 글로벌 접두사 설정된
`setGlobalPrefix()` 메서드
   */
  무시 글로벌 접두사?: 부울;

  /**
   * true`인 경우, 스웨거는 다음에서 가져온 모듈의 경로도 로드합니다.
포함` 모듈
   */
  deepScanRoutes?: 부울;

  /**
   * 커스텀 오퍼레이션아이디팩토리를 생성하는 데 사용되는
`운영아이디`
   * 컨트롤러 키`와 `메소드 키`를 기반으로 합니다.
   * 기본값 () => 컨트롤러키_메소드키
   */
  operationIdFactory?: (컨트롤러키: 문자열, 메서드키: 문자열) => 문자열;
}
```

예를 들어 라이브러리에서 `UserController_createUser` 대신 `createUser와` 같은 작업 이름을 생성하도록 하려면 다음을 설정하면 됩니다:

```
const 옵션: SwaggerDocumentOptions ={
                                          o
  perationIdFactory: (
    컨트롤러키: 문자열, 메서드키:
    문자열
  ) => 메서드키
};
const document = SwaggerModule.createDocument(app, config, options);
```

설정 옵션

`SwaggerModule#setup` 메서드의 네 번째 인수로 `ExpressSwaggerCustomOptions`(express를 사용하는 경우) 인터페이스를 충족하는 옵션 오브젝트를 전달하여 Swagger UI를 구성할 수 있습니다.

```
내보내기 인터페이스 ExpressSwaggerCustomOptions { 탐색기
  ?: 부울;
  swaggerOptions? Record<string, any>;
  customCss?: 문자열;
  customCssUrl?: 문자열;
  customJs?: 문자열;
  customfavIcon?: 문자열;
  swaggerUrl?: 문자열;
  customSiteTitle?: 문자열;
  validatorUrl?: 문자열; url?:
  문자열입니다;
  urls?: Record<'url' | 'name', string>[];
  patchDocumentOnRequest?: <TRequest = any, TResponse = any> (req:
TRequest, res: TResponse, 문서: OpenAPIObject) => OpenAPIObject;
}
```

예

작동하는 예제는 여기에서 확인할 수 있습니다.

유형 및 매개변수

SwaggerModule은 라우트 핸들러에서 모든 @Body(), @Query(), @Param() 데코레이터를 검색하여 API 문서를 생성합니다. 또한 리플렉션을 활용하여 해당 모델 정의를 생성합니다. 다음 코드를 살펴보세요:

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

> 정보 힌트 본문 정의를 명시적으로 설정하려면 @ApiBody() 데코레이터를 사용하십시오 (nestjs/swagger 패키지).

CreateCatDto를 기반으로 다음과 같은 모델 정의 Swagger UI가 생성됩니다:

보시다시피 클래스에 몇 가지 선언된 프로퍼티가 있지만 정의는 비어 있습니다. 클래스 프로퍼티를 SwaggerModule에 표시하려면 @ApiProperty() 데코레이터로 주석을 달거나 자동으로 처리하는 CLI 플러그인(플러그인 섹션에서 자세히 읽어보세요)을 사용해야 합니다:

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기

클래스 CreateCatDto {
  @ApiProperty()
  이름: 문자열;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

> 정보 힌트 각 프로퍼티에 수동으로 주석을 다는 대신 이를 자동으로 제공하는 Swagger 플러그인(플러그인 섹션 참조)을 사용하는 것이 좋습니다.

브라우저를 열고 생성된 CreateCatDto 모델을 확인해 보겠습니다:

또한 `@ApiProperty()` 데코레이터를 사용하면 다양한 스키마 객체 속성을 설정할 수 있습니다:

```
@ApiProperty({
  설명: '고양이의 나이',
```

```
  최소: 1,
  기본값: 1,
})
나이: 숫자;
```

> 정보 힌트 {{"@ApiProperty({ 필수: false })"}}를 명시적으로 입력하는 대신 `@ApiPropertyOptional()` 단축 데코레이터를 사용할 수 있습니다.

속성의 유형을 명시적으로 설정하려면 `유형` 키를 사용합니다:

```
@ApiProperty({
  type: 숫자,
})
나이: 숫자;
```

배열

프로퍼티가 배열인 경우 아래와 같이 배열 유형을 수동으로 표시해야 합니다:

```
@ApiProperty({ 유형: [문자열] }) names:
문자열[];
```

> 정보 힌트 배열을 자동으로 감지하는 Swagger 플러그인(플러그인 섹션 참조)을 사용하는 것이 좋습니다.

위와 같이 유형을 배열의 첫 번째 요소로 포함하거나(위 그림 참조), `isArray` 속성을 다음과 같이 설정합니다. `true`.

순환 종속성

클래스 간에 순환 종속성이 있는 경우, 지연 함수를 사용하여 `SwaggerModule을` 제공하세요.

유형 정보와 함께:

```
@ApiProperty({ 유형: () => 노드 })
노드: 노드;
```

> 정보 힌트 순환 종속성을 자동으로 감지하는 Swagger 플러그인(플러그인 섹션 참조)을 사용하는 것이 좋습니다.

제네릭 및 인터페이스

TypeScript는 제네릭이나 인터페이스에 대한 메타데이터를 저장하지 않으므로 DTO에서 제네릭이나 인터페이스를 사용할 때 SwaggerModule이 런타임에 모델 정의를 제대로 생성하지 못할 수 있습니다. 예를 들어, 다음 코드는 Swagger 모듈에서 올바르게 검사되지 않습니다:

```
createBulk(@Body() usersDto: CreateUserDto[])
```

이 제한을 극복하기 위해 유형을 명시적으로 설정할 수 있습니다:

```
APIBody({ type: [CreateUserDto] })
createBulk(@Body() usersDto: CreateUserDto[])
```

열거형

`열거` 형을 식별하려면 값 배열을 사용하여 `@ApiProperty에서` 열거 형 속성을 수동으로 설정해야 합니다.

```
@ApiProperty({ 열거형: ['관리자', '진행자', '사용자']}) 역할:
UserRole;
```

또는 다음과 같이 실제 TypeScript 열거형을 정의합니다:

```
export enum UserRole {
  Admin = 'Admin',
  Moderator = 'Moderator',
  User = 'User',
}
```

그런 다음 `@Query()` 매개 변수 데코레이터와 함께 열거 형을 직접 사용할 수 있습니다.

`@ApiQuery()` 데코레이터.

```
@ApiQuery({ 이름: 'role', 열거형: UserRole })
async filterByRole(@Query('role') role: UserRole = UserRole.User) {}
```

`isArray를` true로 설정하면 열거형을 다중 선택으로 선택할 수 있습니다:

열거형 스키마

기본적으로 열거형 속성은 `매개변수에` 열거형에 대한 원시 정의를 추가합니다.

```
- 품종:
    유형: '문자열'
```

```
    열거형:
      - 페르시아어
      - Tabby
      - Siamese
```

위의 사양은 대부분의 경우 정상적으로 작동합니다. 그러나 사양을 입력으로 받아 클라이언트 측 코드를 생성하는 도구를 사용하는 경우 생성된 코드에 중복된 열거 형이 포함되어 있는 문제가 발생할 수 있습니다. 다음 코드 스니펫을 살펴보세요:

```
// 생성된 클라이언트 측 코드 내보내기 클래스 CatDetail {
  품종: CatDetailEnum;
}

내보내기 클래스 CatInformation {
  breed: CatInformationEnum;
}

export enum CatDetailEnum {
  페르시아어 = '페르시아어', 타비 = '타비',
  샴어 = '샴어',
}

내보내기 열거형 CatInformationEnum {
  페르시아어 = '페르시아어',
  Tabby = '얼룩말',
  Siamese = '샴',
}
```

> 정보 힌트 위의 스니펫은 NSwag라는 도구를 사용하여 생성되었습니다.

이제 완전히 동일한 열거형 두 개가 생겼음을 알 수 있습니다. 이 문제를 해결하려면 데코레이터의 enum 속성과 함께 enumName을 입력합니다.

```
내보내기 클래스 CatDetail {
  @ApiProperty({ enum: CatBreed, enumName: 'CatBreed' })
  breed: CatBreed;
}
```

enumName 속성을 사용하면 @nestjs/swagger가 CatBreed를 자체 스키마로 전환할 수 있으며, 이를 통해

`CatBreed` 열거형을 재사용할 수 있게 됩니다. 사양은 다음과 같습니다:

```
CatDetail: 유형:
  '객체' 속성:
    ...
```

```
    - 품종:
        스키마:
          참조: '#/components/schemas/CatBreed'
CatBreed: 유형:
  문자열 열거형입
  니다:
    - 페르시아어
    - Tabby
    - Siamese
```

> 정보 힌트 `enum을` 속성으로 취하는 데코레이터는 `enumName도` 취합니다.

원시 정의

일부 특정 시나리오(예: 깊게 중첩된 배열, 행렬)에서는 유형을 직접 설명해야 할 수도 있습니다.

```
@ApiProperty({
  type: 'array',
  items: {
    유형: '배열', 항목: {
      유형: '숫자',
    },
  },
})
좌표: 숫자[][];
```

마찬가지로 컨트롤러 클래스에서 입력/출력 콘텐츠를 수동으로 정의하려면 `스키마를` 사용하세요.

속성입니다:

```
@ApiBody({
  schema: {
    유형: '배열', 항목: {
      유형: '배열', 항목:
      {
        유형: '숫자',
      },
    },
  },
})
async create(@Body() coords: number[][]) {}
```

추가 모델

컨트롤러에서 직접 참조되지는 않지만 Swagger 모듈에서 검사해야 하는 추가 모델을 정의하려면 `@ApiExtraModels()` 데코레이터를 사용합니다:

```
@ApiExtraModels(ExtraModel)
내보내기 클래스 CreateCatDto {}
```

> 정보 힌트 특정 모델 클래스에 대해 `@ApiExtraModels()`를 한 번만 사용하면 됩니다.

또는 `추가 모델` 속성이 지정된 옵션 객체를 `SwaggerModule#createDocument()` 메서드를 다음과 같이 호출합니다:

```
const document = SwaggerModule.createDocument(app, options, {
  extraModels: [ExtraModel],
});
```

모델에 대한 참조(`$ref`)를 가져오려면 `getSchemaPath(ExtraModel)` 함수를 사용합니다:

```
'application/vnd.api+json': {
    스키마를 추가합니다: { $ref: getSchemaPath(ExtraModel) },
},
```

oneOf, anyOf, allOf

스키마를 결합하려면 `oneOf`, `anyOf` 또는 `allOf` 키워드를 사용할 수 있습니다(자세히 보기).

```
@ApiProperty({
  oneOf: [
    { $ref: getSchemaPath(Cat) },
    { $ref: getSchemaPath(Dog) },
  ],
})
반려동물: 고양이 | 개;
```

다형성 배열(즉, 멤버가 여러 스키마에 걸쳐 있는 배열)을 정의하려면 원시 정의(위 참조)를 사용하여 유형을 수동으로 정의해야 합니다.

```
유형 애완동물 = 고양이 | 개;

@ApiProperty({
  유형: '배열', 항
  목: {
    oneOf: [
      { $ref: getSchemaPath(Cat) },
```

```
        { $ref: getSchemaPath(Dog) },
      ],
    },
  })
  애완 동물: 애완동물[];
```

> 정보 힌트 `getSchemaPath()` 함수는 `@nestjs/swagger`에서 가져온 것입니다.

Cat과 Dog는 모두 (클래스 수준에서) `@ApiExtraModels()` 데코레이터를 사용하여 추가 모델로 정의해야 합니다.

운영

OpenAPI 용어에서 경로는 API가 노출하는 `/users` 또는 `/reports/summary`와 같은 엔드포인트(리소스)이며, 작업은 이러한 경로를 조작하는 데 사용되는 HTTP 메서드(예: `GET`, `POST` 또는 `DELETE`)입니다.

태그

특정 태그에 컨트롤러를 연결하려면 `@ApiTags(...태그)` 데코레이터를 사용합니다.

```
@ApiTags('cats')
@Controller('cats')
내보내기 클래스 CatsController {}
```

헤더

요청의 일부로 예상되는 사용자 정의 헤더를 정의하려면 `@ApiHeader()를` 사용합니다.

```
@ApiHeader({
  이름: 'X-MyHeader', 설명: '사용자
  정의 헤더',
})
@Controller('cats')
내보내기 클래스 CatsController {}
```

응답

사용자 정의 HTTP 응답을 정의하려면 `@ApiResponse()` 데코레이터를 사용합니다.

```
@Post()
@ApiResponse({ status: 201, description: '레코드가 성공적으로 생성되었습니다.'})
APIResponse({ status: 403, description: 'Forbidden.'})
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Nest는 `@ApiResponse를` 상속하는 짧은 API 응답 데코레이터 세트를 제공합니다.

데코레이터:

- `@ApiOkResponse()`
- `@ApiCreatedResponse()`
- `@ApiAcceptedResponse()`
- `@ApiNoContentResponse()`

- APIMovedPermanentlyResponse()
- @ApiFoundResponse()
- @ApiBadRequestResponse()
- @ApiUnauthorizedResponse()
- @ApiNotFoundResponse()
- @ApiForbiddenResponse()
- @ApiMethodNotAllowedResponse()
- @ApiNotAcceptableResponse()
- @ApiRequestTimeoutResponse()
- @ApiConflictResponse() @ApiPre-
- conditionFailedResponse()
- @ApiTooManyRequestsResponse()
- @ApiGoneResponse()
- @ApiPayloadTooLargeResponse()
- @ApiUnsupportedMediaTypeResponse()
- @ApiUnprocessableEntityResponse()
- @ApiInternalServerErrorResponse()
- @ApiNotImplementedResponse()
- @ApiBadGatewayResponse()
- @ApiServiceUnavailableResponse()
- @ApiGatewayTimeoutResponse()
- @ApiDefaultResponse()

```
@Post()
@ApiCreatedResponse({ description: '레코드가 성공적으로 생성되었습니다.'})
APIForbiddenResponse({ description: 'Forbidden.'})
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

요청에 대한 반환 모델을 지정하려면 클래스를 생성하고 모든 프로퍼티에

`@ApiProperty()` 데코레이터.

```
export class Cat {
  @ApiProperty()
  id: number;

  @ApiProperty()
  이름: 문자열;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

그런 다음 `Cat` 모델을 응답 데코레이터의 `유형` 속성과 함께 사용할 수 있습니다.

```
@ApiTags('cats')
@Controller('cats')
export class CatsController {
  @Post()
  @ApiCreatedResponse({
    설명: '레코드가 성공적으로 생성되었습니다.', 유형: Cat,
  })
  async create(@Body() createCatDto: CreateCatDto): Promise<Cat> {
    return this.catsService.create(createCatDto);
  }
}
```

브라우저를 열고 생성된 `고양이` 모델을 확인해 보겠습니다:

파일 업로드

다음과 함께 `@ApiBody` 데코레이터를 사용하여 특정 방법에 대한 파일 업로드를 활성화할 수 있습니다.

`를 호출합니다`. 다음은 파일 업로드 기법을 사용한 전체 예제입니다:

```
사용 인터셉터(파일인터셉터('파일'))
@ApiConsumes('multipart/form-data')
@ApiBody({
  설명: '고양이 목록', 유형:
  FileUploadDto,
})
uploadFile(@UploadedFile() 파일) {}
```

여기서 FileUploadDto는 다음과 같이 정의됩니다:

```
FileUploadDto 클래스 {
  @ApiProperty({ 유형: '문자열', 형식: '바이너리' }) 파일:
  any;
}
```

여러 파일 업로드를 처리하려면 다음과 같이 `FilesUploadDto를` 정의할 수 있습니다:

```
클래스 FilesUploadDto {
  @ApiProperty({ 유형: '배열', 항목: { 유형: '문자열', 형식: '바이너리'
} })
```

```
  파일: any[];
}
```

확장 기능

요청에 확장자를 추가하려면 `@ApiExtension()` 데코레이터를 사용합니다. 확장자 이름 앞에는 `x-`를 붙여야 합니다.

```
@ApiExtension('x-foo', { hello: 'world' })
```

고급 일반 `API 응답`

원시 정의를 제공하는 기능을 통해 Swagger UI에 대한 일반 스키마를 정의할 수 있습니다. 다음과 같은 DTO가 있다고 가정합니다:

```
export class PaginatedDto<TData> {
  @ApiProperty()
  합계: 숫자;

  @ApiProperty()
  제한: 숫자;

  @ApiProperty() 오프셋
  : 숫자;

  결과를 반환합니다: TData[];
}
```

나중에 원시 정의를 제공할 것이므로 `결과` 꾸미기는 생략하겠습니다. 이제 다른 DTO를 정의하고 이름을 다음과 같이 CatDto로 지정해 보겠습니다:

```
export class CatDto
  { @ApiProperty()
  name: string;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

이를 통해 다음과 같이 `PaginatedDto<CatDto>` 응답을 정의할 수 있습니다:

```
@ApiOkResponse({
  schema: {
    allOf: [
      { $ref: getSchemaPath(PaginatedDto) },
      {
        속성을 추가합니다: {
          results: {
            유형: '배열',
            항목을 반환합니다: { $ref: getSchemaPath(CatDto) },
          },
        },
      },
    ],
  },
})
비동기 findAll(): Promise<PaginatedDto<CatDto>> {}
```

이 예제에서는 응답에 allOf `PaginatedDto가` 있고 `결과` 속성이 `Array<CatDto>` 유형이 되도록 지정합니다.

- 주어진 모델에 대한 OpenAPI 사양 파일 내에서 OpenAPI 스키마 경로를 반환하는 `getSchemaPath()` 함수를 호출합니다.
- `allOf`는 다양한 상속 관련 사용 사례를 다루기 위해 OAS 3에서 제공하는 개념입니다.

마지막으로, PaginatedDto는 컨트롤러에서 직접 참조하지 않기 때문에 `SwaggerModule`은 아직 해당 모델 정의를 생성할 수 없습니다. 이 경우 추가 모델로 추가해야 합니다. 예를 들어 다음과 같이 컨트롤러 수준에서 `@ApiExtraModels()` 데코레이터를 사용할 수 있습니다:

```
@Controller('cats')
@ApiExtraModels(PaginatedDto)
export class CatsController {}
```

지금 Swagger를 실행하면 이 특정 엔드포인트에 대해 생성된 `swagger.json에` 다음과 같은 응답이 정의되어 있어야 합니다:

```
"응답": {
  "200": {
    "설명": "", "내용": {
      "application/json": {
        "스키마": {
          "allOf": [
            {
              "$ref": "#/components/schemas/PaginatedDto"
            },
            {
              "properties": {
                "results": {
```

```
                  "$ref": "#/components/schemas/CatDto"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

재사용할 수 있도록 하기 위해 다음과 같이 `PaginatedDto에` 대한 사용자 지정 데코레이터를 만들 수 있습니다:

```
export const ApiPaginatedResponse = <TModel extends Type<any>>(
  model: TModel,
) => {
  return applyDecorators(
    ApiExtraModels(model),
    ApiOkResponse({
      스키마: {
        allOf: [
          { $ref: getSchemaPath(PaginatedDto) },
          {
            속성을 추가합니다: {
              results: {
                유형: '배열',
                항목을 반환합니다: { $ref: getSchemaPath(모델) },
              },
            },
          },
        ],
      },
    }),
  );
};
```

> 정보 힌트 `유형<any>` 인터페이스와 `applyDecorators` 함수는
>
> `nestjs/common` 패키지.

SwaggerModule이 모델에 대한 정의를 생성하도록 하려면 앞서 컨트롤러의 `PaginatedDto에서` 한 것처럼 추가 모델로 추가해야 합니다.

이제 엔드포인트에서 사용자 정의 `@ApiPaginatedResponse()` 데코레이터를 사용할 수 있습니다:

```
@ApiPaginatedResponse(CatDto)
비동기 findAll(): Promise<PaginatedDto<CatDto>> {}
```

클라이언트 생성 도구의 경우, 이 접근 방식은 클라이언트에 대해 페이지 매김된 응답<TModel>이 생성되는 방식에 모호함을 야기합니다. 다음 코드조각은 위의 GET/엔드포인트에 대한 클라이언트 생성기 결과의 예시입니다.

```
// 각도
findAll(): Observable<{ total: 숫자, limit: 숫자, offset: 숫자, results:
CatDto[] }>
```

보시다시피 여기의 반환 유형이 모호합니다. 이 문제를 해결하려면 제목을 추가하면 됩니다.

속성을 ApiPaginatedResponse의 스키마에 추가합니다:

```
export const ApiPaginatedResponse = <TModel extends Type<any>>(model:
TModel) => {
  return applyDecorators(
    ApiOkResponse({
      스키마: {
        title: `PaginatedResponseOf${model.name}`
        allOf: [
          // ...
        ],
      },
    }),
  );
};
```

이제 클라이언트 생성기 도구의 결과가 나타납니다:

```
// 각도
findAll(): Observable<PaginatedResponseOfCatDto>
```

보안

특정 작업에 어떤 보안 메커니즘을 사용해야 하는지 정의하려면 `@ApiSecurity()`를 사용하십시오.

데코레이터.

```
@ApiSecurity('basic')
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

`DocumentBuilder`:

```
const options = new DocumentBuilder().addSecurity('basic', {
  type: 'http',
  체계: '기본',
});
```

가장 많이 사용되는 인증 기술 중 일부는 기본으로 제공되므로(예: `기본` 및 `무기명`) 위와 같이 보안 메커니즘을 수동으로 정의할 필요가 없습니다.

기본 인증

기본 인증을 사용하려면 `@ApiBasicAuth()를` 사용합니다.

```
@ApiBasicAuth()
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

`DocumentBuilder`:

```
const options = new DocumentBuilder().addBasicAuth();
```

무기명 인증

무기명 인증을 활성화하려면 `@ApiBearerAuth()`를 사용합니다.

```
APIBearerAuth()
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

`DocumentBuilder`:

```
const options = new DocumentBuilder().addBearerAuth();
```

OAuth2 인증

OAuth2를 활성화하려면 `@ApiOAuth2()`를 사용합니다.

```
@ApiOAuth2(['pets:write'])
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

`DocumentBuilder`:

```
const options = new DocumentBuilder().addOAuth2();
```

쿠키 인증

쿠키 인증을 활성화하려면 `@ApiCookieAuth()`를 사용합니다.

```
API쿠키인증()
@Controller('cats')
내보내기 클래스 CatsController {}
```

애플리케이션을 실행하기 전에 다음을 사용하여 기본 문서에 보안 정의를 추가하는 것을 잊지 마세요.

`DocumentBuilder`:

```
const options = new DocumentBuilder().addCookieAuth('optional-session-
id');
```

매핑된 유형

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

부분

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

Nest는 이 작업을 더 쉽게 수행하고 상용구를 최소화하기 위해 `PartialType()` 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:

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기

클래스 CreateCatDto {
  @ApiProperty()
  이름: 문자열;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

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

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

> 정보 힌트 `PartialType()` 함수는 `@nestjs/swagger` 패키지에서 가져온 것입니다.

선택

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

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기

클래스 CreateCatDto {
  @ApiProperty()
```

```
  이름: 문자열;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

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

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

> 정보 힌트 `PickType()` 함수는 `@nestjs/swagger` 패키지에서 가져온 것입니다.

생략

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

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기

클래스 CreateCatDto {
  @ApiProperty()
  이름: 문자열;

  @ApiProperty()
  age: 숫자;

  @ApiProperty()
  품종: 문자열;
}
```

아래와 같이 `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/swagger` 패키지에서 가져온 것입니다.

교차로

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

```
'@nestjs/swagger'에서 { ApiProperty } 임포트; 내보내기

클래스 CreateCatDto {
  @ApiProperty()
  이름: 문자열;

  @ApiProperty()
  품종: 문자열;
}

내보내기 클래스 AdditionalCatInfo {
  @ApiProperty()
  색상: 문자열;
}
```

두 유형의 모든 속성을 결합한 새로운 유형을 생성할 수 있습니다.

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

> 정보 힌트 `IntersectionType()` 함수는 `@nestjs/swagger` 패키지에서 가져온 것입니다.

구성

유형 매핑 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),
) {}
```

데코레이터

사용 가능한 모든 OpenAPI 데코레이터에는 핵심 데코레이터와 구분하기 위해 `API` 접두사가 붙습니다. 아래는 내보낸 데코레이터의 전체 목록과 데코레이터가 적용될 수 있는 레벨을 지정한 것입니다.

`@ApiBasicAuth()`	메서드 / 컨트롤러
`@ApiBearerAuth()`	메서드 / 컨트롤러
`@ApiBody()`	메서드
`@ApiConsumes()`	메서드 / 컨트롤러
`@ApiCookieAuth()`	메서드/컨트롤러
`@ApiExcludeController()`	컨트롤러
`@ApiExcludeEndpoint()`	메서드
`@ApiExtension()`	메서드
`@ApiExtraModels()`	메서드 / 컨트롤러
`@ApiHeader()`	메서드 / 컨트롤러
`@ApiHideProperty()`	Model
`@ApiOAuth2()`	메서드 / 컨트롤러
`@ApiOperation()`	메서드
`@ApiParam()`	메서드
`@ApiProduces()`	메서드 / 컨트롤러
`@ApiProperty()`	모델
`@ApiPropertyOptional()`	모델
`@ApiQuery()`	메서드

`@ApiResponse()`	메서드 / 컨트롤러
`@ApiSecurity()`	메서드 / 컨트롤러
`@ApiTags()`	메서드 / 컨트롤러

CLI 플러그인

TypeScript의 메타데이터 반영 시스템에는 몇 가지 제약이 있어 클래스가 어떤 프로퍼티로 구성되어 있는지 확인하거나 특정 프로퍼티가 선택 사항인지 필수 사항인지 인식할 수 없습니다. 하지만 이러한 제약 조건 중 일부는 컴파일 시점에 해결할 수 있습니다. Nest는 필요한 상용구 코드의 양을 줄이기 위해 TypeScript 컴파일 프로세스를 개선하는 플러그인을 제공합니다.

> 정보 힌트 이 플러그인은 옵트인입니다. 원하는 경우 모든 데코레이터를 수동으로 선언하거나 필요한 곳에 특정 데코레이터만 선언할 수 있습니다.

개요

스웨거 플러그인이 자동으로 실행됩니다:

- annotate all DTO properties with `@ApiProperty` unless `@ApiHideProperty` is used
- 물음표에 따라 `필요한` 속성을 설정합니다(예: `이름?: 문자열이` 설정됨). `필수: 거짓`)
- 유형에 따라 `유형` 또는 열거형 속성을 설정합니다(배열도 지원)• 지정된 기본값에 따라 `기본` 속성을 설정합니다.
- `클래스 유효성` 검사기 데코레이터를 기반으로 몇 가지 유효성 검사 규칙을 설정합니다(`클래스 유효성` 검사기 `심이 true로` 설정된 경우).
- 적절한 상태와 `유형`(응답 모델)으로 모든 엔드포인트에 응답 데코레이터를 추가합니다.
- 코멘트를 기반으로 프로퍼티 및 엔드포인트에 대한 설명을 생성합니다(`introspectComments 참으로` 설정)
- 코멘트를 기반으로 프로퍼티에 대한 예제 값을 생성합니다(`introspectComments가 true로` 설정된 경우).

파일 이름에는 다음 접미사 중 하나가 포함되어야 합니다: `['.dto.ts', '.entity.ts']` (예: `create-user.dto.ts)`를 생성하여 플러그인에서 분석할 수 있도록 합니다.

다른 접미사를 사용하는 경우 플러그인의 동작을 조정하려면 `dtoFileNameSuffix` 옵션(아래 참조).

이전에는 Swagger UI로 인터랙티브한 경험을 제공하려면 모델/컴포넌트를 사양에 어떻게 선언해야 하는지 패키지에 알리기 위해 많은 코드를 복제해야 했습니다. 예를 들어 다음과 같이 간단한 `CreateUserDto` 클래스를 정

의할 수 있습니다:

```
export 클래스 CreateUserDto {
  @ApiProperty()
  이메일: 문자열;

  @ApiProperty() 비밀번호
  : 문자열;

  @ApiProperty({ 열거형: RoleEnum, 기본값: [], isArray: true }) 역할:
  RoleEnum[] = [];

  @ApiProperty({ 필수: false, 기본값: true }) isEnabled?:
  boolean = true;
}
```

중간 규모의 프로젝트에서는 큰 문제가 되지 않지만, 클래스의 수가 많아지면 장황해지고 유지 관리가 어려워집니다.

Swagger 플러그인을 활성화하면 위의 클래스 정의를 간단하게 선언할 수 있습니다:

```
export 클래스 CreateUserDto {
  이메일: 문자열;
  비밀번호: 문자열; 역할:
  RoleEnum[] = [];
  isEnabled?: boolean = true;
}
```

> 정보 참고 Swagger 플러그인은 TypeScript 유형 및 클래스 유효성 검사기 데코레이터에서 @ApiProperty() 어노테이션을 파생합니다. 이는 생성된 Swagger UI 문서에 대한 API를 명확하게 설명하는 데 도움이 됩니다. 그러나 런타임에 유효성 검사는 여전히 클래스 유효성 검사기 데코레이터에 의해 처리됩니다. 따라서 `IsEmail()`, `IsNumber()` 등과 같은 유효성 검사기를 계속 사용해야 합니다.

따라서 문서 생성을 위해 자동 어노테이션에 의존하면서도 런타임 유효성 검사를 원하는 경우 클래스 유효성 검사기 데코레이터가 여전히 필요합니다.

> 정보 힌트 DTO에서 매핑된 유형 유틸리티(예: `PartialType`)를 사용할 때는 다음에서 가져옵니다. 플러그인이 스키마를 가져올 수 있도록 `@nestjs/mapped-types` 대신 `@nestjs/swagger`를 입력하세요.

플러그인은 추상 구문 트리를 기반으로 적절한 데코레이터를 즉시 추가합니다. 따라서 코드 곳곳에 흩어져 있는 `@ApiProperty` 데코레이터로 고생할 필요가 없습니다.

> 정보 힌트 플러그인은 누락된 스웨거 프로퍼티를 자동으로 생성하지만, 재정의해야 하는 경우 `@ApiProperty()`를 통해 명시적으로 설정하기만 하면 됩니다.

댓글 성찰

댓글 인트로스펙션 기능을 활성화하면 CLI 플러그인이 댓글을 기반으로 속성에 대한 설명과 예제 값을 생성합니다.

예를 들어 역할 속성을 예로 들어 보겠습니다:

```
/**
 * 사용자 역할 목록
 * @example ['admin']
 */
@ApiProperty({
  설명: `사용자의 역할 목록`, 예시: ['admin'],
})
역할: RoleEnum[] = [];
```

설명과 예제 값을 모두 복제해야 합니다. `introspectComments를` 활성화하면 CLI 플러그인이 이러한 코멘트를 추출하여 속성에 대한 설명(정의된 경우 예제)을 자동으로 제공할 수 있습니다. 이제 위의 프로퍼티는 다음과 같이 간단하게 선언할 수 있습니다:

```
/**
 * 사용자 역할 목록
 * @example ['admin']
 */
역할: RoleEnum[] = [];
```

플러그인에서 각각 `ApiProperty` 및 `ApiOperation` 데코레이터의 값을 설정하는 방법을 사용자 정의하는 데 사용할 수 있는 `dtoKeyOfComment` 및 `controllerKeyOfComment` 플러그인 옵션이 있습니다. 다음 예시를 살펴보세요:

```
일부 컨트롤러 내보내기 클래스 {
  /**
   * 리소스 생성
   */
  @Post()
  create() {}
}
```

기본적으로 이 옵션은 `"설명"`으로 설정되어 있습니다. 즉, 플러그인은 `"일부 리소스 만들기"를 설명` 키에 할당하여 `ApiOperation` 연산자의 `설명` 키로 사용합니다. 이렇게 설정합니다:

```
@ApiOperation({ description: "일부 리소스 생성" })
```

> 정보 힌트 모델의 경우 동일한 로직이 적용되지만 대신 `ApiProperty` 데코레이터에 적용됩니다.

CLI 플러그인 사용

플러그인을 활성화하려면 `nest-cli.json`(Nest CLI를 사용하는 경우)을 열고 다음 `플러그인을` 추가합니다. 구성합니다:

```
{
  "collection": "@nestjs/schematics",
  "sourceRoot": "src", "컴파일러옵션": {
    "플러그인": ["@nestjs/swagger"]
  }
}
```

옵션 속성을 사용하여 플러그인의 동작을 사용자 정의할 수 있습니다.

```
"플러그인": [
  {
    "name": "@nestjs/swagger", "옵
    션": {
      "classValidatorShim": false,
      "introspectComments": true
    }
  }
]
```

옵션 속성은 다음 인터페이스를 충족해야 합니다:

```
export interface PluginOptions {
  dtoFileNameSuffix?: string[];
  controllerFileNameSuffix?: string[];
  classValidatorShim?: boolean;
  dtoKeyOfComment?: string;
  controllerKeyOfComment?: string;
  introspectComments?: boolean;
}
```

옵션	기본값	설명
dtoFileName접미사	['.dto.ts', '.entity.ts']	DTO(데이터 전송 개체) 파일 접미사
컨트롤러 파일 이름 접미사 .controller.ts		컨트롤러 파일 접미사
클래스 유효성 검사기 심	참	true로 설정하면 모듈은 클래스 유효성 검사기 유효성 검사 데코레이터를 재사용합니다(예 Max(10)은 스키마 정의에 최대: 10을 스키마 정의에 추가합니다)
dtoKeyOfComment	'설명'	댓글 텍스트를 커짐으로 설정하는 속성 키입니다. ApiProperty.
컨트롤러 키 코멘트	'설명'	댓글 텍스트를 커짐으로 설정하는 속성 키입니다. ApiOperation.
introspectComments	false	true로 설정하면 플러그인이 댓글을 기반으로 속성에 대한 설명과 예제 값을 생성합니다.

플러그인 옵션이 업데이트될 때마다 `/dist` 폴더를 삭제하고 애플리케이션을 다시 빌드해야 합니다. CLI를 사용하지 않고 사용자 정의 `웹팩을` 구성한 경우 이 플러그인을 `ts-loader와` 함께 사용할 수 있습니다:

```
  getCustomTransformers: (program: any) => ({{
    before: [require('@nestjs/swagger/plugin').before({}, program)]
  }),
```

SWC 빌더

표준 설정(비모노레포)의 경우 SWC 빌더에서 CLI 플러그인을 사용하려면 여기에 설명된 대로 유형 검사를 사용하도록 설정해야 합니다.

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

모노레포 설정의 경우 여기 지침을 따르세요.

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

이제 아래와 같이 직렬화된 메타데이터 파일을 `SwaggerModule#loadPluginMetadata` 메서드를 통해 로드해야 합니다:

```
'./metadata'에서 메타데이터 가져오기; // <-- "플러그인 메타데이터 생성기"에 의해 자동 생성된 파일입니다.

await SwaggerModule.loadPluginMetadata(metadata); // <-- 여기
const document = SwaggerModule.createDocument(app, config);
```

`ts-jest`와 통합(e2e 테스트)

`ts-jest`는 e2e 테스트를 실행하기 위해 소스 코드 파일을 메모리에서 즉석에서 컴파일합니다. 즉, Nest CLI 컴파일러를 사용하지 않으며 플러그인을 적용하거나 AST 변환을 수행하지 않습니다.

플러그인을 활성화하려면 e2e 테스트 디렉토리에 다음 파일을 생성합니다:

```
const transformer = require('@nestjs/swagger/plugin');

module.exports.name = 'nestjs-swagger-transformer';
// 아래 구성을 변경할 때마다 버전 번호를 변경해야 합니다. 그렇지 않으면 jest가 변경
사항을 감지하지 못합니다 module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
      // @nestjs/swagger/플러그인 옵션(비워둘 수 있음)
```

```
    },
    cs.program, // 이전 버전의 Jest의 경우 "cs.tsCompiler.program" (<=
v27)
  );
};
```

이 설정이 완료되면 `jest` 구성 파일에서 AST 트랜스포머를 가져옵니다. 기본적으로(스타터 애플리케이션에서) e2e 테스트 구성 파일은 `테스트` 폴더 아래에 있으며 이름은 `jest-e2e.json`입니다.

```
{
  ... // 기타 구성 "전역": {
    "ts-jest": {
      "astTransformers": {
        "전에": ["<위에서 생성한 파일 경로>"]
      }
    }
  }
}
```

jest@^29를 사용하는 경우 이전 접근 방식이 더 이상 사용되지 않으므로 아래 스니펫을 사용하세요.

```
{
  ... // 기타 구성 "변환": {
    "^.+\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "전에": ["<위에서 생성한 파일 경로>"]
        }
      }
    ]
  }
}
```

농담 문제 해결(e2e 테스트)

Jest가 설정 변경 사항을 적용하지 않는 것 같다면 이미 빌드 결과를 캐시한 것일 수 있습니다. 새 구성을 적용하려면 Jest의 캐시 디렉터리를 지워야 합니다.

캐시 디렉터리를 지우려면 NestJS 프로젝트 폴더에서 다음 명령을 실행합니다:

```
$ npx jest --clearCache
```

자동 캐시 지우기가 실패하는 경우에도 다음 명령을 사용하여 캐시 폴더를 수동으로 제거할 수 있습니다:

```
# 농담 캐시 디렉터리 찾기(보통 /tmp/jest_rs)
# NestJS 프로젝트 루트에서 다음 명령을 실행합니다.
$ npx jest --showConfig | grep cache
# ex 결과:
#   "캐시": true,
#   "캐시디렉토리": "/tmp/jest_rs"

# Jest 캐시 디렉터리 제거 또는 비우기
$ rm -rf  <cacheDirectory값> #
```

예제:

```
# rm -rf /tmp/jest_rs
```

기타 기능

이 페이지에는 유용하게 사용할 수 있는 다른 모든 기능이 나열되어 있습니다. 글

로벌 접두사

`setGlobalPrefix()`를 통해 설정된 경로의 글로벌 접두사를 무시하려면 `ignoreGlobalPrefix`를 사용합니다:

```
const document = SwaggerModule.createDocument(app, options, {
  ignoreGlobalPrefix: true,
});
```

전역 매개 변수

`문서 작성기`를 사용하여 모든 경로에 매개변수 정의를 추가할 수 있습니다:

```
const options = new DocumentBuilder().addGlobalParameters({
  name: 'tenantId',
  in: '헤더',
});
```

다양한 사양

SwaggerModule은 여러 사양을 지원하는 방법을 제공합니다. 즉, 서로 다른 엔드포인트에서 서로 다른 UI로 서로 다른 문서를 제공할 수 있습니다.

여러 사양을 지원하려면 애플리케이션을 모듈 방식으로 작성해야 합니다. `createDocument()` 메서드에는 `include라는` 속성을 가진 객체인 세 번째 인수 `extraOptions가 사용됩니다`. `include` 속성은 모듈의 배열인 값을 받습니다.

아래와 같이 여러 사양 지원을 설정할 수 있습니다:

```
'@nestjs/core'에서 { NestFactory }를 가져옵니다;
'@nestjs/swagger'에서 { SwaggerModule, DocumentBuilder }를 임포트하고,
'./app.module'에서 { AppModule }을 임포트합니다;
'./cats/cats.module'에서 { CatsModule }을 임포트하고,
'./dogs/dogs.module'에서 { DogsModule }을 임포트합니다;

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

  /**
   * createDocument(애플리케이션, 구성옵션, 추가옵션);
   *
   * createDocument 메서드는 선택적 세 번째 인자 "extraOptions"를 받습니다.
   * 를 전달할 수 있는 "포함" 속성을 가진 객체입니다.
```

```
  배열
    * 해당 스웨거 사양에 포함하려는 모듈의 개수
    * 예시 예를 들어, 캣츠모듈과 독스모듈에는 다음과 같은 두 개의 별도 스웨거 사양
이 있습니다.
    * 는 두 개의 서로 다른 엔드포인트가 있는 두 개의 서로 다른 SwaggerUI에 노출
됩니다.
    */

  const options = new DocumentBuilder()
    .setTitle('고양이 예시')
    .setDescription('고양이 API 설명')
    .setVersion('1.0')
    .addTag('cats')
    .build();

  const catDocument = SwaggerModule.createDocument(app, options, {
    include: [CatsModule],
  });
  SwaggerModule.setup('api/cats', app, catDocument);

  const secondOptions = new DocumentBuilder()
    .setTitle('개 예시')
    .setDescription('개 API 설명')
    .setVersion('1.0')
    .addTag('dogs')
    .build();

  const dogDocument = SwaggerModule.createDocument(app, secondOptions, {
    include: [DogsModule],
  });
  SwaggerModule.setup('api/dogs', app, dogDocument);

  await app.listen(3000);
}
부트스트랩();
```

이제 다음 명령으로 서버를 시작할 수 있습니다:

```
$ npm 실행 시작
```

`http://localhost:3000/api/cats` 으로 이동하여 고양이를 위한 Swagger UI를 확인하세요:

그러면 `http://localhost:3000/api/dogs` 에 반려견용 Swagger UI가 노출됩니다:

마이그레이션 가이드

현재 `@nestjs/swagger@3.*`를 사용 중이라면 버전 4.0의 다음과 같은 중단/API 변경 사항을 참고하세요. 주요 변경 사항

다음 데코레이터가 변경/명칭이 변경되었습니다:

- 이제 @ApiModelProperty는 `@ApiProperty`입니다.
- @ApiModelPropertyOptional은 이제 `@ApiPropertyOptional`
- @ApiResponseModelProperty는 이제 `@ApiResponseProperty`
- @ApiImplicitQuery는 이제 `@ApiQuery`로 변경되었습니다.
- @ApiImplicitParam은 이제 `@ApiParam`
- @ApiImplicitBody는 이제 `@ApiBody`입니다.
- @ApiImplicitHeader는 이제 `@ApiHeader`입니다.
- 이제 `@ApiOperation({{ '{' }} title: 'test' {{ '}' }})`은 `@ApiOperation({{ '{' }} summary: 'test' {{ '}' }})`
- 이제 `@ApiUseTags`는 `@ApiTags`입니다.

`DocumentBuilder` 브레이킹 변경 사항(메서드 서명 업데이트):

- `추가 태그`
- `addBearerAuth`
- `addOAuth2`
- `setContactEmail`은 이제 `setContact`입니다.
- `setHost`가 제거되었습니다.
- setSchemes가 제거되었습니다(대신 addServer`('http://')`와 같은 추가 서버를 사용하세요

). 새로운 메서드

다음과 같은 방법이 추가되었습니다:

- 추가서버 추가Api
-
- 키 추가기본인증
- 추가보안
- 보안 요구 사항 추가