OpenAPI (Swagger)

This chapter applies only to TypeScript

The **OpenAPI** (Swagger) specification is a powerful definition format to describe RESTful APIs. Nest provides a dedicated **module** to work with it.

Installation

Firstly, you have to install the required packages:

```
$ npm install --save @nestjs/swagger swagger-ui-express
```

If you are using fastify, you have to install `fastify-swagger` instead of `swagger-ui-express`:

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

Bootstrap

Once the installation process is done, open your bootstrap file (mostly `main.ts`) and initialize the Swagger using `SwaggerModule` class:

```
import { NestFactory } from '@nestjs/core';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { ApplicationModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(ApplicationModule);

  const options = new DocumentBuilder()
    .setTitle('Cats example')
    .setDescription('The cats API description')
    .setVersion('1.0')
    .addTag('cats')
    .build();
  const document = SwaggerModule.createDocument(app, options);
  SwaggerModule.setup('api', app, document);
```

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

The `DocumentBuilder` is a helper class that helps to structure a base document for the `SwaggerModule`. It contains several methods that allow setting such properties like title, description, version, and so on.

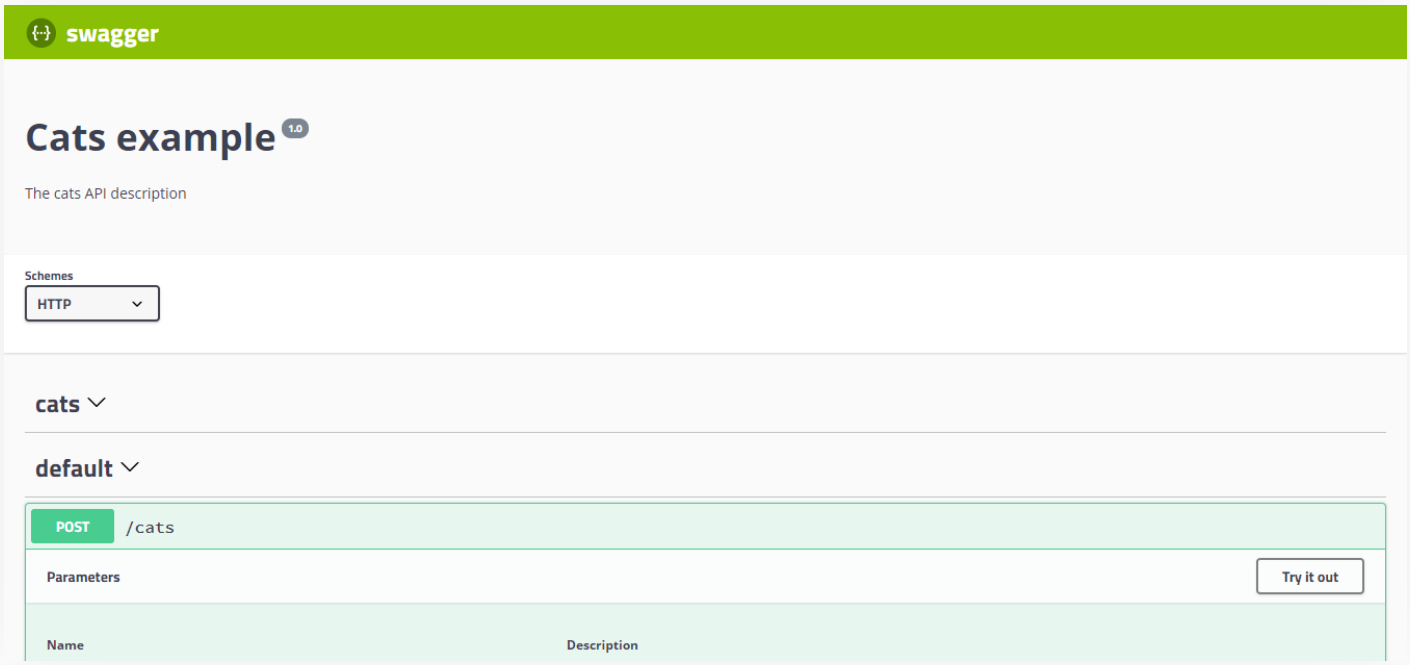
In order to create a full document (with defined HTTP routes) we use the `createDocument()` method of the `SwaggerModule` class. This method takes two arguments, the application instance and the base Swagger options respectively.

The last step is to call `setup()`. It accepts sequentially **(1)** path to mount the Swagger, **(2)** application instance, and **(3)** the document that describes the Nest application.

Now you can run the following command to start the HTTP server:

```
$ npm run start
```

While the application is running, open your browser and navigate to `http://localhost:3000/api`. You should see a similar page:



The `SwaggerModule` automatically reflects all of your endpoints. In the background, it's making use of `swagger-ui-express` and creates a live documentation.

> **HINT**

If you want to download the corresponding Swagger JSON file, you can simply call `http://localhost:3000/api-json` in your browser (if your Swagger documentation is published under `http://localhost:3000/api`).

Body, query, path parameters

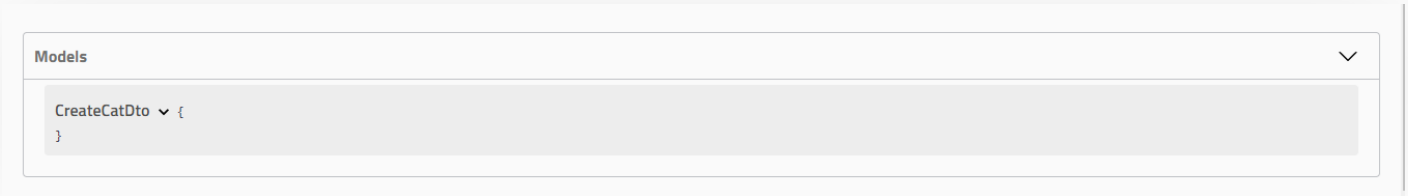
During the examination of the defined controllers, the `SwaggerModule` is looking for all used `@Body()`, `@Query()`, and `@Param()` decorators in the route handlers. Hence, the valid document can be created.

Moreover, the module creates the **models definitions** by taking advantage of the reflection. Take a look at the following code:

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

> **NOTICE**
> To implicitly set the body definition you can use the `@ApiImplicitBody()` decorator (`@nestjs/swagger` package).

Based on the `CreateCatDto`, the module definition will be created:

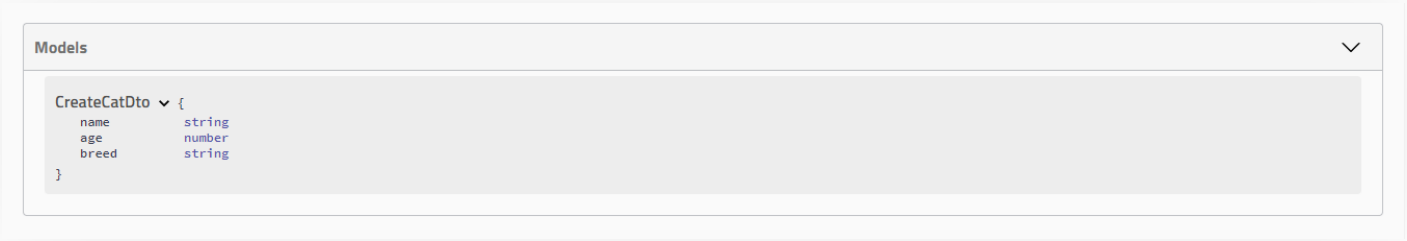


As you can see, the definition is empty although the class has a few declared properties. In order to make the class properties accessible to the `SwaggerModule`, we have to mark all of them with `@ApiModelProperty()` decorator:

```
import { ApiModelProperty } from '@nestjs/swagger';

export class CreateCatDto {
  @ApiModelProperty()
  readonly name: string;

  @ApiModelProperty()
  readonly age: number;
```

```
  @ApiModelProperty()
  readonly breed: string;
}
```

Let's open the browser and verify the generated `CreateCatDto` model:



The `@ApiModelProperty()` decorator accepts options object:

```
export const ApiModelProperty: (metadata?: {
  description?: string;
  required?: boolean;
  type?: any;
  isArray?: boolean;
  collectionFormat?: string;
  default?: any;
  enum?: SwaggerEnumType;
  format?: string;
  multipleOf?: number;
  maximum?: number;
  exclusiveMaximum?: number;
  minimum?: number;
  exclusiveMinimum?: number;
  maxLength?: number;
  minLength?: number;
  pattern?: string;
  maxItems?: number;
  minItems?: number;
  uniqueItems?: boolean;
  maxProperties?: number;
  minProperties?: number;
  readOnly?: boolean;
  xml?: any;
  example?: any;
}) => PropertyDecorator;
```

> **HINT**
> There's an `@ApiModelPropertyOptional()` shortcut decorator which helps to avoid continuous typing `@ApiModelProperty(&#123 required: false &#125)`.

Thanks to that we can simply set the **default** value, determine whether the property is required or explicitly set the type.

Multiple specifications

Swagger module also provides a way to support multiple specifications. In other words, you can serve different documentations with different `SwaggerUI` on different endpoints.

In order to allow `SwaggerModule` to support multi-spec, your application must be written with modular approach. The `createDocument()` method takes in a 3rd argument: `extraOptions` which is an object where a property `include` expects an array of modules.

You can setup Multiple Specifications support as shown below:

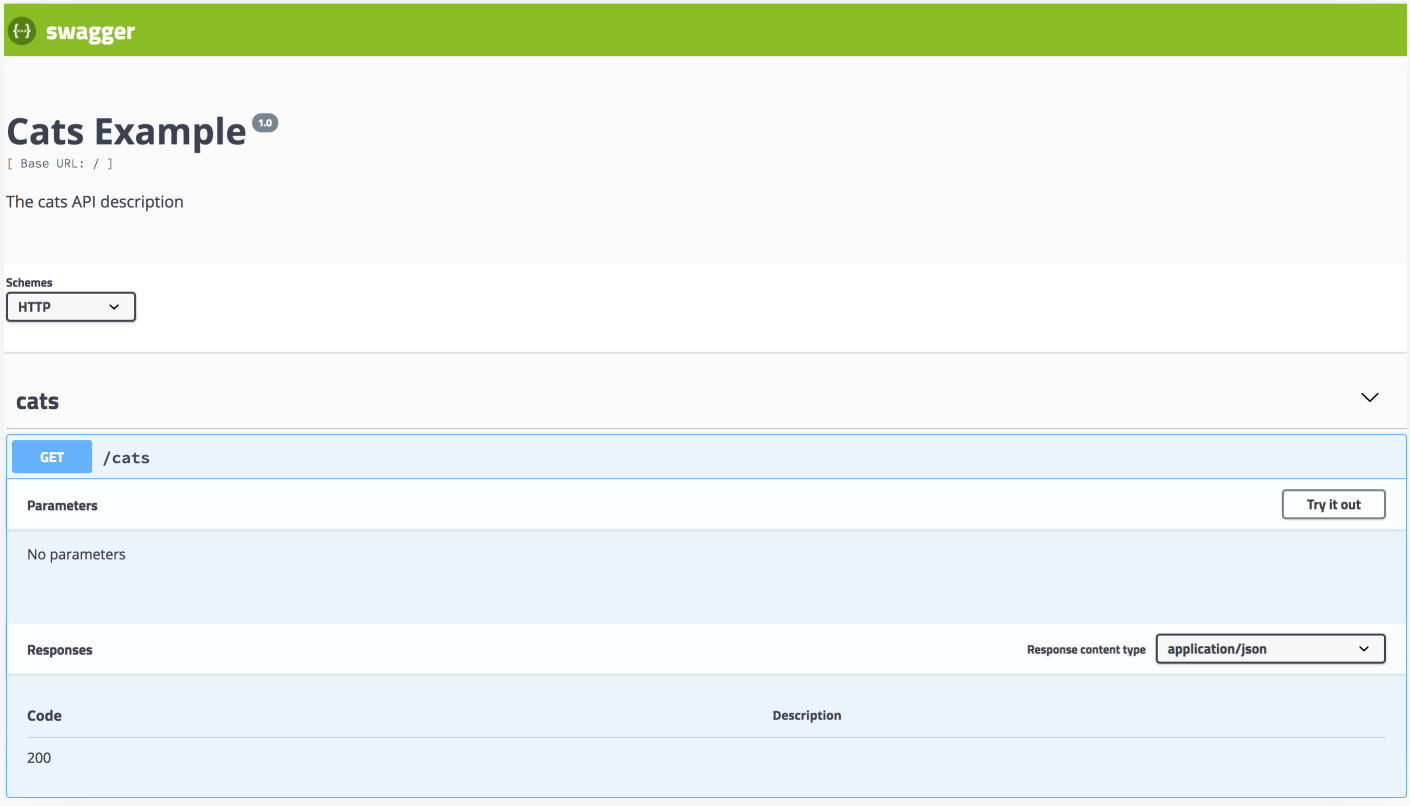
```
import { NestFactory } from '@nestjs/core';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { ApplicationModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(ApplicationModule);

  /**
   * createDocument(application, configurationOptions, extraOptions);
   *
   * createDocument method takes in an optional 3rd argument "extraOptions"
   * which is an object with "include" property where you can pass an Array
   * of Modules that you want to include in that Swagger Specification
   * E.g: CatsModule and DogsModule will have two separate Swagger Specifications which
   * will be exposed on two different SwaggerUI with two different endpoints.
   */

  const options = new DocumentBuilder()
    .setTitle('Cats example')
    .setDescription('The cats API description')
    .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('Dogs example')
    .setDescription('The dogs API description')
    .setVersion('1.0')
    .addTag('dogs')
    .build();

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

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

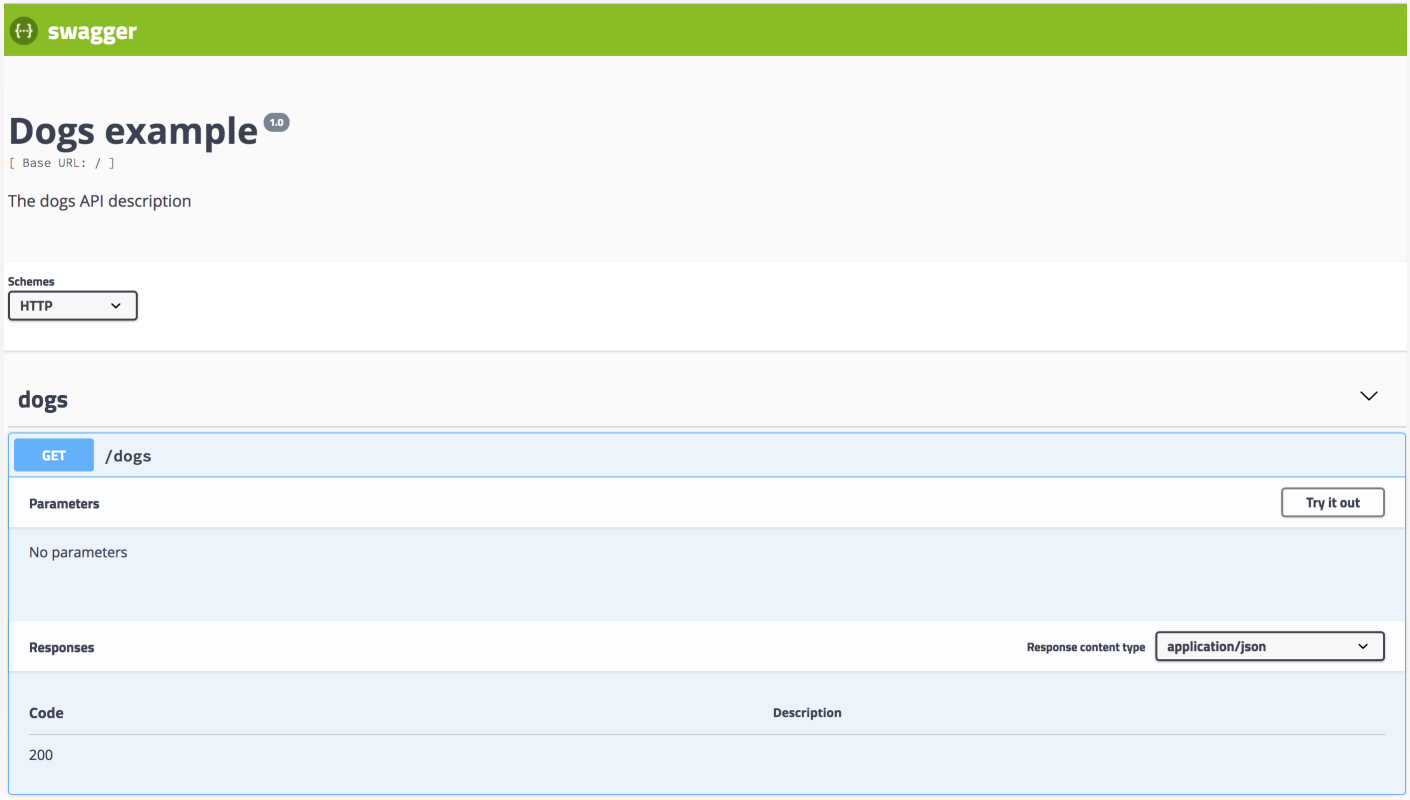
Now you can start your server with the following command:

```
$ npm run start
```

Navigate to `http://localhost:3000/api/cats` to see SwaggerUI for your cats:



While `http://localhost:3000/api/dogs` will expose a SwaggerUI for your dogs:



> **NOTICE**
> You have to construct a **SwaggerOptions** with `DocumentBuilder`, run `createDocument()` against newly constructed `options` then immediately "serve" it with `setup()` before you can start working on a second **SwaggerOptions** for a second Swagger Specification. This specific order is to prevent Swagger configurations being overridden by different options.

Working with enums

To be able for `SwaggerModule` to identify an `Enum`, we have to manually set the `enum` property on `@ApiModelProperty` with an array of values.

```
@ApiModelProperty({ enum: ['Admin', 'Moderator', 'User']})
role: UserRole;
```

`UserRole` enum can be defined as following:

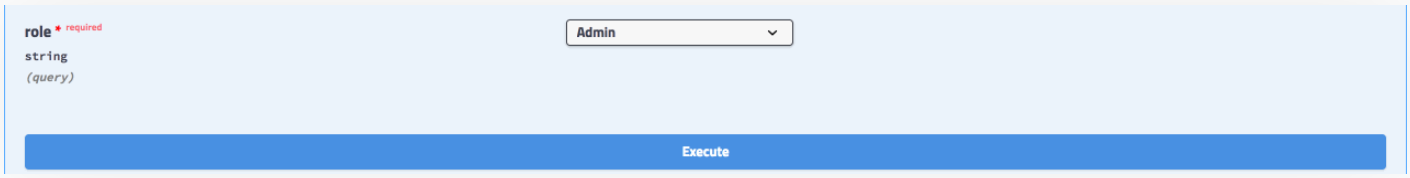
```
export enum UserRole {
  Admin = 'Admin',
  Moderator = 'Moderator',
```

```
  User = 'User',
}
```

> **NOTE**
> The above usage can only be applied to a **property** as part of a **model definition.**

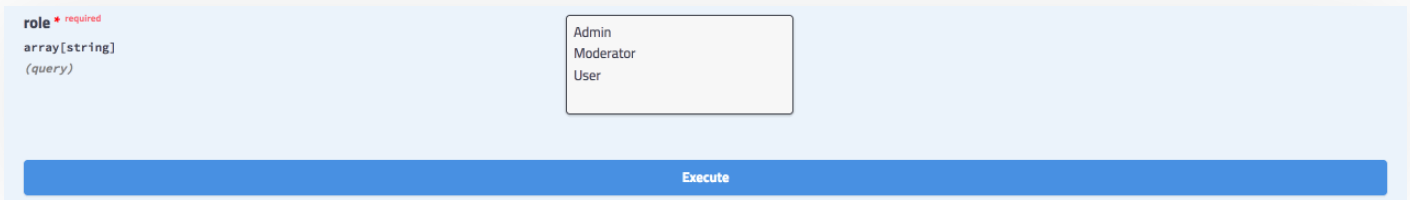
Enums can be used by itself with the `@Query()` parameter decorator in combination with the `@ApiImplicitQuery()` decorator.

```
@ApiImplicitQuery({ name: 'role', enum: ['Admin', 'Moderator', 'User'] })
async filterByRole(@Query('role') role: UserRole = UserRole.User) {
  // role returns: UserRole.Admin, UserRole.Moderator OR UserRole.User
}
```



> **HINT**
> `enum` and `isArray` can also be used in combination in `@ApiImplicitQuery()`

With `isArray` set to **true**, the `enum` can be selected as a **multi-select**:



Working with arrays

We have to manually indicate a type when the property is actually an array:

```
@ApiModelProperty({ type: [String] })
```

```
@ApiModelProperty({ type: [String] })
readonly names: string[];
```

Simply put your type as the first element of an array (as shown above) or set an `isArray` property to `true`.

Tags

At the beginning, we created a `cats` tag (by making use of `DocumentBuilder`). In order to attach the controller to the specified tag, we need to use `@ApiUseTags(...tags)` decorator.

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

Responses

To define a custom HTTP response, we use `@ApiResponse()` decorator.

```
@Post()
@ApiResponse({ status: 201, description: 'The record has been successfully created.'})
@ApiResponse({ status: 403, description: 'Forbidden.'})
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

Same as common HTTP exceptions defined in Exception Filters section, Nest also provides a set of usable **API responses** that inherits from the core `@ApiResponse` decorator:

- `@ApiOkResponse()`
- `@ApiCreatedResponse()`
- `@ApiBadRequestResponse()`
- `@ApiUnauthorizedResponse()`
- `@ApiNotFoundResponse()`
- `@ApiForbiddenResponse()`
- `@ApiMethodNotAllowedResponse()`
- `@ApiNotAcceptableResponse()`
- `@ApiRequestTimeoutResponse()`
- `@ApiConflictResponse()`
- `@ApiGoneResponse()`
- `@ApiPayloadTooLargeResponse()`
- `@ApiUnsupportedMediaTypeResponse()`

- `@ApiUnprocessableEntityResponse()`
- `@ApiInternalServerErrorResponse()`
- `@ApiNotImplementedResponse()`
- `@ApiBadGatewayResponse()`
- `@ApiServiceUnavailableResponse()`
- `@ApiGatewayTimeoutResponse()`

In addition to the available HTTP exceptions, Nest provides short-hand decorators for: `HttpStatus.OK`, `HttpStatus.CREATED` and `HttpStatus.METHOD_NOT_ALLOWED`

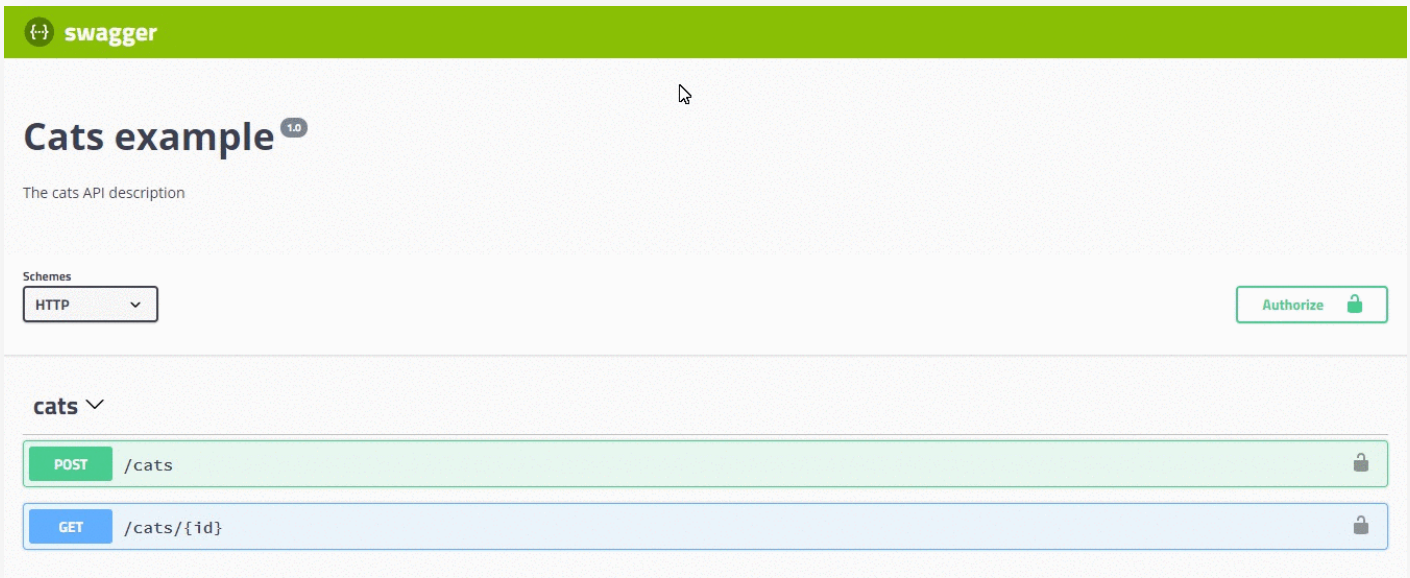
```
@Post()
@ApiCreatedResponse({ description: 'The record has been successfully created.'})
@ApiForbiddenResponse({ description: 'Forbidden.'})
async create(@Body() createCatDto: CreateCatDto) {
  this.catsService.create(createCatDto);
}
```

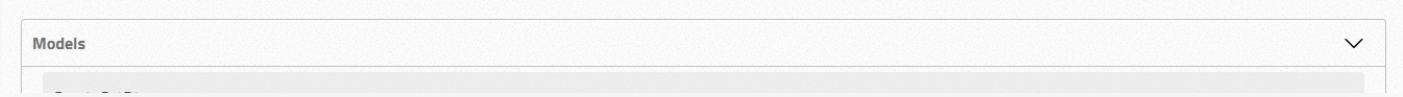
Authentication

You can enable the bearer authorization using `addBearerAuth()` method of the `DocumentBuilder` class. Then to restrict the chosen route or entire controller, use `@ApiBearerAuth()` decorator.

```
@ApiUseTags('cats')
@ApiBearerAuth()
@Controller('cats')
export class CatsController {}
```

That's how the OpenAPI documentation should look like now:





File upload

You can enable file upload for a specific method with the `@ApiImplicitFile` decorator together with `@ApiConsumes()`. Here's a full example using **File Upload** technique:

```
@UseInterceptors(FileInterceptor('file'))
@ApiConsumes('multipart/form-data')
@ApiImplicitFile({ name: 'file', required: true, description: 'List of cats' })
uploadFile(@UploadedFile() file) {}
```

Decorators

All of the available OpenAPI decorators has an `Api` prefix to be clearly distinguishable from the core decorators. Below is a full list of the exported decorators with a defined use-level (where might be applied).

`@ApiOperation()`	Method
`@ApiResponse()`	Method / Controller
`@ApiProduces()`	Method / Controller
`@ApiConsumes()`	Method / Controller
`@ApiBearerAuth()`	Method / Controller
`@ApiOAuth2Auth()`	Method / Controller
`@ApiImplicitBody()`	Method
`@ApiImplicitParam()`	Method
`@ApiImplicitQuery()`	Method

`@ApiImplicitHeader()`	Method
`@ApiImplicitFile()`	Method
`@ApiExcludeEndpoint()`	Method
`@ApiUseTags()`	Method / Controller
`@ApiModelProperty()`	Model
`@ApiModelPropertyOptional()`	Model

A working example is available here.

Support us

Nest is an MIT-licensed open source project. It can grow thanks to the support by these awesome people. If you'd like to join them, please read more here.

Principal Sponsor

Sponsors / Partners