Introduction

The OpenAPI specification is a language-agnostic definition format used to describe RESTful APIs. Nest provides a dedicated module which allows generating such a specification by leveraging decorators.

Installation

To begin using it, we first install the required dependency.

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

Bootstrap

Once the installation process is complete, open the `main.ts` file and initialize Swagger using the `SwaggerModule` class:

```
@@filename(main)
import { NestFactory } from '@nestjs/core';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { AppModule } from './app.module';

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

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

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

> info **Hint** `document` (returned by the `SwaggerModule#createDocument()` method) is a serializable object conforming to OpenAPI Document. Instead of hosting it via HTTP, you could also save it as a JSON/YAML file, and consume it in different ways.

The `DocumentBuilder` helps to structure a base document that conforms to the OpenAPI Specification. It provides several methods that allow setting such properties as title, description, version, etc. In order to create a full document (with all HTTP routes defined) we use the `createDocument()` method of the `SwaggerModule` class. This method takes two arguments, an application instance and a Swagger options object. Alternatively, we can provide a third argument, which should be of type `SwaggerDocumentOptions`. More on this in the Document options section.

Once we create a document, we can call the `setup()` method. It accepts:

1. The path to mount the Swagger UI
2. An application instance
3. The document object instantiated above
4. Optional configuration parameter (read more here)

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 the Swagger UI.

As you can see, the `SwaggerModule` automatically reflects all of your endpoints.

> info **Hint** To generate and download a Swagger JSON file, navigate to `http://localhost:3000/api-json` (assuming that your Swagger documentation is available under `http://localhost:3000/api`).

> warning **Warning** When using `fastify` and `helmet`, there may be a problem with CSP, to solve this collision, configure the CSP as shown below:
>
> ```
> app.register(helmet, {
>   contentSecurityPolicy: {
>     directives: {
>       defaultSrc: [`'self'`],
>       styleSrc: [`'self'`, `'unsafe-inline'`],
>       imgSrc: [`'self'`, 'data:', 'validator.swagger.io'],
>       scriptSrc: [`'self'`, `https: 'unsafe-inline'`],
>     },
>   },
> });
>
> // If you are not going to use CSP at all, you can use this:
> app.register(helmet, {
>   contentSecurityPolicy: false,
> });
> ```

Document options

When creating a document, it is possible to provide some extra options to fine tune the library's behavior. These options should be of type `SwaggerDocumentOptions`, which can be the following:

```
export interface SwaggerDocumentOptions {
  /**
   * List of modules to include in the specification
   */
  include?: Function[];

  /**
   * Additional, extra models that should be inspected and included in the
specification
   */
  extraModels?: Function[];

  /**
   * If `true`, swagger will ignore the global prefix set through
`setGlobalPrefix()` method
   */
  ignoreGlobalPrefix?: boolean;

  /**
   * If `true`, swagger will also load routes from the modules imported by
`include` modules
   */
  deepScanRoutes?: boolean;

  /**
   * Custom operationIdFactory that will be used to generate the
`operationId`
   * based on the `controllerKey` and `methodKey`
   * @default () => controllerKey_methodKey
   */
  operationIdFactory?: (controllerKey: string, methodKey: string) =>
string;
}
```

For example, if you want to make sure that the library generates operation names like `createUser` instead of `UserController_createUser`, you can set the following:

```
const options: SwaggerDocumentOptions =  {
  operationIdFactory: (
    controllerKey: string,
    methodKey: string
  ) => methodKey
};
const document = SwaggerModule.createDocument(app, config, options);
```

Setup options

You can configure Swagger UI by passing the options object which fulfills the `ExpressSwaggerCustomOptions` (if you use express) interface as a fourth argument of the `SwaggerModule#setup` method.

```
export interface ExpressSwaggerCustomOptions {
  explorer?: boolean;
  swaggerOptions?: Record<string, any>;
  customCss?: string;
  customCssUrl?: string;
  customJs?: string;
  customfavIcon?: string;
  swaggerUrl?: string;
  customSiteTitle?: string;
  validatorUrl?: string;
  url?: string;
  urls?: Record<'url' | 'name', string>[];
  patchDocumentOnRequest?: <TRequest = any, TResponse = any> (req:
TRequest, res: TResponse, document: OpenAPIObject) => OpenAPIObject;
}
```

Example

A working example is available here.

Types and parameters

The `SwaggerModule` searches for all `@Body()`, `@Query()`, and `@Param()` decorators in route handlers to generate the API document. It also creates corresponding model definitions by taking advantage of reflection. Consider the following code:

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

> info **Hint** To explicitly set the body definition use the `@ApiBody()` decorator (imported from the `@nestjs/swagger` package).

Based on the `CreateCatDto`, the following model definition Swagger UI 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 visible to the `SwaggerModule`, we have to either annotate them with the `@ApiProperty()` decorator or use the CLI plugin (read more in the **Plugin** section) which will do it automatically:

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

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

> info **Hint** Instead of manually annotating each property, consider using the Swagger plugin (see Plugin section) which will automatically provide this for you.

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

In addition, the `@ApiProperty()` decorator allows setting various Schema Object properties:

```
@ApiProperty({
  description: 'The age of a cat',
  minimum: 1,
  default: 1,
})
age: number;
```

> info **Hint** Instead of explicitly typing the `{{"@ApiProperty({ required: false })"}}` you can use the `@ApiPropertyOptional()` short-hand decorator.

In order to explicitly set the type of the property, use the `type` key:

```
@ApiProperty({
  type: Number,
})
age: number;
```

Arrays

When the property is an array, we must manually indicate the array type as shown below:

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

> info **Hint** Consider using the Swagger plugin (see Plugin section) which will automatically detect arrays.

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

Circular dependencies

When you have circular dependencies between classes, use a lazy function to provide the `SwaggerModule` with type information:

```
@ApiProperty({ type: () => Node })
node: Node;
```

> info **Hint** Consider using the Swagger plugin (see Plugin section) which will automatically detect circular dependencies.

Generics and interfaces

Since TypeScript does not store metadata about generics or interfaces, when you use them in your DTOs, `SwaggerModule` may not be able to properly generate model definitions at runtime. For instance, the following code won't be correctly inspected by the Swagger module:

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

In order to overcome this limitation, you can set the type explicitly:

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

Enums

To identify an `enum`, we must manually set the `enum` property on the `@ApiProperty` with an array of values.

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

Alternatively, define an actual TypeScript enum as follows:

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

You can then use the enum directly with the `@Query()` parameter decorator in combination with the `@ApiQuery()` decorator.

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

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

Enums schema

By default, the `enum` property will add a raw definition of Enum on the `parameter`.

```
- breed:
    type: 'string'
    enum:
      - Persian
      - Tabby
      - Siamese
```

The above specification works fine for most cases. However, if you are utilizing a tool that takes the specification as **input** and generates **client-side** code, you might run into a problem with the generated code containing duplicated `enums`. Consider the following code snippet:

```
// generated client-side code
export class CatDetail {
  breed: CatDetailEnum;
}

export class CatInformation {
  breed: CatInformationEnum;
}

export enum CatDetailEnum {
  Persian = 'Persian',
  Tabby = 'Tabby',
  Siamese = 'Siamese',
}

export enum CatInformationEnum {
  Persian = 'Persian',
  Tabby = 'Tabby',
  Siamese = 'Siamese',
}
```

info **Hint** The above snippet is generated using a tool called NSwag.

You can see that now you have two `enums` that are exactly the same. To address this issue, you can pass an `enumName` along with the `enum` property in your decorator.

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

The `enumName` property enables `@nestjs/swagger` to turn `CatBreed` into its own `schema` which in turns makes `CatBreed` enum reusable. The specification will look like the following:

```
CatDetail:
  type: 'object'
  properties:
    ...
```

```
      - breed:
          schema:
            $ref: '#/components/schemas/CatBreed'
  CatBreed:
    type: string
    enum:
      - Persian
      - Tabby
      - Siamese
```

> info **Hint** Any **decorator** that takes `enum` as a property will also take `enumName`.

Raw definitions

In some specific scenarios (e.g., deeply nested arrays, matrices), you may want to describe your type by hand.

```
@ApiProperty({
  type: 'array',
  items: {
    type: 'array',
    items: {
      type: 'number',
    },
  },
})
coords: number[][];
```

Likewise, in order to define your input/output content manually in controller classes, use the `schema` property:

```
@ApiBody({
  schema: {
    type: 'array',
    items: {
      type: 'array',
      items: {
        type: 'number',
      },
    },
  },
})
async create(@Body() coords: number[][]) {}
```

Extra models

To define additional models that are not directly referenced in your controllers but should be inspected by the Swagger module, use the `@ApiExtraModels()` decorator:

```
@ApiExtraModels(ExtraModel)
export class CreateCatDto {}
```

> info **Hint** You only need to use `@ApiExtraModels()` once for a specific model class.

Alternatively, you can pass an options object with the `extraModels` property specified to the `SwaggerModule#createDocument()` method, as follows:

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

To get a reference (`$ref`) to your model, use the `getSchemaPath(ExtraModel)` function:

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

oneOf, anyOf, allOf

To combine schemas, you can use the `oneOf`, `anyOf` or `allOf` keywords (read more).

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

If you want to define a polymorphic array (i.e., an array whose members span multiple schemas), you should use a raw definition (see above) to define your type by hand.

```
type Pet = Cat | Dog;

@ApiProperty({
  type: 'array',
  items: {
    oneOf: [
      { $ref: getSchemaPath(Cat) },
      { $ref: getSchemaPath(Dog) },
    ],
  },
})
pets: Pet[];
```

> info **Hint** The `getSchemaPath()` function is imported from `@nestjs/swagger`.

Both `Cat` and `Dog` must be defined as extra models using the `@ApiExtraModels()` decorator (at the class-level).

Operations

In OpenAPI terms, paths are endpoints (resources), such as `/users` or `/reports/summary`, that your API exposes, and operations are the HTTP methods used to manipulate these paths, such as `GET`, `POST` or `DELETE`.

Tags

To attach a controller to a specific tag, use the `@ApiTags(...tags)` decorator.

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

Headers

To define custom headers that are expected as part of the request, use `@ApiHeader()`.

```
@ApiHeader({
  name: 'X-MyHeader',
  description: 'Custom header',
})
@Controller('cats')
export class CatsController {}
```

Responses

To define a custom HTTP response, use the `@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);
}
```

Nest provides a set of short-hand **API response** decorators that inherit from the `@ApiResponse` decorator:

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

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

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

To specify a return model for a request, we must create a class and annotate all properties with the `@ApiProperty()` decorator.

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

  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

Then the `Cat` model can be used in combination with the `type` property of the response decorator.

```
@ApiTags('cats')
@Controller('cats')
export class CatsController {
  @Post()
  @ApiCreatedResponse({
    description: 'The record has been successfully created.',
    type: Cat,
  })
  async create(@Body() createCatDto: CreateCatDto): Promise<Cat> {
    return this.catsService.create(createCatDto);
  }
}
```

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

File upload

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

```
@UseInterceptors(FileInterceptor('file'))
@ApiConsumes('multipart/form-data')
@ApiBody({
  description: 'List of cats',
  type: FileUploadDto,
})
uploadFile(@UploadedFile() file) {}
```

Where `FileUploadDto` is defined as follows:

```
class FileUploadDto {
  @ApiProperty({ type: 'string', format: 'binary' })
  file: any;
}
```

To handle multiple files uploading, you can define `FilesUploadDto` as follows:

```
class FilesUploadDto {
  @ApiProperty({ type: 'array', items: { type: 'string', format: 'binary'
} })
```

```
  files: any[];
}
```

Extensions

To add an Extension to a request use the `@ApiExtension()` decorator. The extension name must be prefixed with `x-`.

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

Advanced: Generic `ApiResponse`

With the ability to provide Raw Definitions, we can define Generic schema for Swagger UI. Assume we have the following DTO:

```
export class PaginatedDto<TData> {
  @ApiProperty()
  total: number;

  @ApiProperty()
  limit: number;

  @ApiProperty()
  offset: number;

  results: TData[];
}
```

We skip decorating `results` as we will be providing a raw definition for it later. Now, let's define another DTO and name it, for example, `CatDto`, as follows:

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

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

With this in place, we can define a `PaginatedDto<CatDto>` response, as follows:

```
@ApiOkResponse({
  schema: {
    allOf: [
      { $ref: getSchemaPath(PaginatedDto) },
      {
        properties: {
          results: {
            type: 'array',
            items: { $ref: getSchemaPath(CatDto) },
          },
        },
      },
    ],
  },
})
async findAll(): Promise<PaginatedDto<CatDto>> {}
```

In this example, we specify that the response will have allOf `PaginatedDto` and the `results` property will be of type `Array<CatDto>`.

- `getSchemaPath()` function that returns the OpenAPI Schema path from within the OpenAPI Spec File for a given model.
- `allOf` is a concept that OAS 3 provides to cover various Inheritance related use-cases.

Lastly, since `PaginatedDto` is not directly referenced by any controller, the `SwaggerModule` will not be able to generate a corresponding model definition just yet. In this case, we must add it as an Extra Model. For example, we can use the `@ApiExtraModels()` decorator on the controller level, as follows:

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

If you run Swagger now, the generated `swagger.json` for this specific endpoint should have the following response defined:

```
"responses": {
  "200": {
    "description": "",
    "content": {
      "application/json": {
        "schema": {
          "allOf": [
            {
              "$ref": "#/components/schemas/PaginatedDto"
            },
            {
              "properties": {
                "results": {
                  "$ref": "#/components/schemas/CatDto"
                }
              }
            }
          ]
        }
      }
    }
  }
}
```

To make it reusable, we can create a custom decorator for `PaginatedDto`, as follows:

```
export const ApiPaginatedResponse = <TModel extends Type<any>>(
  model: TModel,
) => {
  return applyDecorators(
    ApiExtraModels(model),
    ApiOkResponse({
      schema: {
        allOf: [
          { $ref: getSchemaPath(PaginatedDto) },
          {
            properties: {
              results: {
                type: 'array',
                items: { $ref: getSchemaPath(model) },
              },
            },
          },
        ],
      },
    }),
  );
};
```

> info **Hint** `Type<any>` interface and `applyDecorators` function are imported from the `@nestjs/common` package.

To ensure that `SwaggerModule` will generate a definition for our model, we must add it as an extra model, like we did earlier with the `PaginatedDto` in the controller.

With this in place, we can use the custom `@ApiPaginatedResponse()` decorator on our endpoint:

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

For client generation tools, this approach poses an ambiguity in how the `PaginatedResponse<TModel>` is being generated for the client. The following snippet is an example of a client generator result for the above `GET /` endpoint.

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

As you can see, the **Return Type** here is ambiguous. To workaround this issue, you can add a `title` property to the `schema` for `ApiPaginatedResponse`:

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

Now the result of the client generator tool will become:

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

Security

To define which security mechanisms should be used for a specific operation, use the `@ApiSecurity()` decorator.

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

Before you run your application, remember to add the security definition to your base document using `DocumentBuilder`:

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

Some of the most popular authentication techniques are built-in (e.g., `basic` and `bearer`) and therefore you don't have to define security mechanisms manually as shown above.

Basic authentication

To enable basic authentication, use `@ApiBasicAuth()`.

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

Before you run your application, remember to add the security definition to your base document using `DocumentBuilder`:

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

Bearer authentication

To enable bearer authentication, use `@ApiBearerAuth()`.

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

Before you run your application, remember to add the security definition to your base document using `DocumentBuilder`:

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

OAuth2 authentication

To enable OAuth2, use `@ApiOAuth2()`.

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

Before you run your application, remember to add the security definition to your base document using `DocumentBuilder`:

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

Cookie authentication

To enable cookie authentication, use `@ApiCookieAuth()`.

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

Before you run your application, remember to add the security definition to your base document using `DocumentBuilder`:

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

Mapped types

As you build out features like **CRUD** (Create/Read/Update/Delete) it's often useful to construct variants on a base entity type. Nest provides several utility functions that perform type transformations to make this task more convenient.

Partial

When building input validation types (also called DTOs), it's often useful to build **create** and **update** variations on the same type. For example, the **create** variant may require all fields, while the **update** variant may make all fields optional.

Nest provides the `PartialType()` utility function to make this task easier and minimize boilerplate.

The `PartialType()` function returns a type (class) with all the properties of the input type set to optional. For example, suppose we have a **create** type as follows:

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

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

By default, all of these fields are required. To create a type with the same fields, but with each one optional, use `PartialType()` passing the class reference (`CreateCatDto`) as an argument:

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

> info **Hint** The `PartialType()` function is imported from the `@nestjs/swagger` package.

Pick

The `PickType()` function constructs a new type (class) by picking a set of properties from an input type. For example, suppose we start with a type like:

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

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

We can pick a set of properties from this class using the `PickType()` utility function:

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

> info **Hint** The `PickType()` function is imported from the `@nestjs/swagger` package.

Omit

The `OmitType()` function constructs a type by picking all properties from an input type and then removing a particular set of keys. For example, suppose we start with a type like:

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

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  age: number;

  @ApiProperty()
  breed: string;
}
```

We can generate a derived type that has every property **except** `name` as shown below. In this construct, the second argument to `OmitType` is an array of property names.

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

> info **Hint** The `OmitType()` function is imported from the `@nestjs/swagger` package.

Intersection

The `IntersectionType()` function combines two types into one new type (class). For example, suppose we start with two types like:

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

export class CreateCatDto {
  @ApiProperty()
  name: string;

  @ApiProperty()
  breed: string;
}

export class AdditionalCatInfo {
  @ApiProperty()
  color: string;
}
```

We can generate a new type that combines all properties in both types.

```
export class UpdateCatDto extends IntersectionType(
  CreateCatDto,
  AdditionalCatInfo,
) {}
```

> info **Hint** The `IntersectionType()` function is imported from the `@nestjs/swagger` package.

Composition

The type mapping utility functions are composable. For example, the following will produce a type (class) that has all of the properties of the `CreateCatDto` type except for `name`, and those properties will be set to optional:

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

Decorators

All of the available OpenAPI decorators have an `Api` prefix to distinguish them from the core decorators. Below is a full list of the exported decorators along with a designation of the level at which the decorator may be applied.

`@ApiBasicAuth()`	Method / Controller
`@ApiBearerAuth()`	Method / Controller
`@ApiBody()`	Method
`@ApiConsumes()`	Method / Controller
`@ApiCookieAuth()`	Method / Controller
`@ApiExcludeController()`	Controller
`@ApiExcludeEndpoint()`	Method
`@ApiExtension()`	Method
`@ApiExtraModels()`	Method / Controller
`@ApiHeader()`	Method / Controller
`@ApiHideProperty()`	Model
`@ApiOAuth2()`	Method / Controller
`@ApiOperation()`	Method
`@ApiParam()`	Method
`@ApiProduces()`	Method / Controller
`@ApiProperty()`	Model
`@ApiPropertyOptional()`	Model
`@ApiQuery()`	Method
`@ApiResponse()`	Method / Controller
`@ApiSecurity()`	Method / Controller
`@ApiTags()`	Method / Controller

CLI Plugin

TypeScript's metadata reflection system has several limitations which make it impossible to, for instance, determine what properties a class consists of or recognize whether a given property is optional or required. However, some of these constraints can be addressed at compilation time. Nest provides a plugin that enhances the TypeScript compilation process to reduce the amount of boilerplate code required.

> info **Hint** This plugin is **opt-in**. If you prefer, you can declare all decorators manually, or only specific decorators where you need them.

Overview

The Swagger plugin will automatically:

- annotate all DTO properties with `@ApiProperty` unless `@ApiHideProperty` is used
- set the `required` property depending on the question mark (e.g. `name?: string` will set `required: false`)
- set the `type` or `enum` property depending on the type (supports arrays as well)
- set the `default` property based on the assigned default value
- set several validation rules based on `class-validator` decorators (if `classValidatorShim` set to `true`)
- add a response decorator to every endpoint with a proper status and `type` (response model)
- generate descriptions for properties and endpoints based on comments (if `introspectComments` set to `true`)
- generate example values for properties based on comments (if `introspectComments` set to `true`)

Please, note that your filenames **must have** one of the following suffixes: `['.dto.ts', '.entity.ts']` (e.g., `create-user.dto.ts`) in order to be analysed by the plugin.

If you are using a different suffix, you can adjust the plugin's behavior by specifying the `dtoFileNameSuffix` option (see below).

Previously, if you wanted to provide an interactive experience with the Swagger UI, you had to duplicate a lot of code to let the package know how your models/components should be declared in the specification. For example, you could define a simple `CreateUserDto` class as follows:

```
export class CreateUserDto {
  @ApiProperty()
  email: string;

  @ApiProperty()
  password: string;

  @ApiProperty({ enum: RoleEnum, default: [], isArray: true })
  roles: RoleEnum[] = [];

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

While not a significant issue with medium-sized projects, it becomes verbose & hard to maintain once you have a large set of classes.

By enabling the Swagger plugin, the above class definition can be declared simply:

```
export class CreateUserDto {
  email: string;
  password: string;
  roles: RoleEnum[] = [];
  isEnabled?: boolean = true;
}
```

> info **Note** The Swagger plugin will derive the @ApiProperty() annotations from the TypeScript types and class-validator decorators. This helps in clearly describing your API for the generated Swagger UI documentation. However, the validation at runtime would still be handled by class-validator decorators. So, it is required to continue using validators like `IsEmail()`, `IsNumber()`, etc.

Hence, if you intend to relay on automatic annotations for generating documentations and still wish for runtime validations, then the class-validator decorators are still necessary.

> info **Hint** When using mapped types utilities (like `PartialType`) in DTOs import them from `@nestjs/swagger` instead of `@nestjs/mapped-types` for the plugin to pick up the schema.

The plugin adds appropriate decorators on the fly based on the **Abstract Syntax Tree**. Thus you won't have to struggle with `@ApiProperty` decorators scattered throughout the code.

> info **Hint** The plugin will automatically generate any missing swagger properties, but if you need to override them, you simply set them explicitly via `@ApiProperty()`.

Comments introspection

With the comments introspection feature enabled, CLI plugin will generate descriptions and example values for properties based on comments.

For example, given an example `roles` property:

```
/**
 * A list of user's roles
 * @example ['admin']
 */
@ApiProperty({
  description: `A list of user's roles`,
  example: ['admin'],
})
roles: RoleEnum[] = [];
```

You must duplicate both description and example values. With `introspectComments` enabled, the CLI plugin can extract these comments and automatically provide descriptions (and examples, if defined) for properties. Now, the above property can be declared simply as follows:

```
/**
 * A list of user's roles
 * @example ['admin']
 */
roles: RoleEnum[] = [];
```

There are `dtoKeyOfComment` and `controllerKeyOfComment` plugin options that you can use to customize how the plugin will set the value for `ApiProperty` and `ApiOperation` decorators respectively. Take a look at the following example:

```
export class SomeController {
  /**
   * Create some resource
   */
  @Post()
  create() {}
}
```

By default, these options are set to `"description"`. This means the plugin will assign `"Create some resource"` to `description` key on the `ApiOperation` operator. Like so:

```
@ApiOperation({ description: "Create some resource" })
```

> info **Hint** For models, the same logic applies but to `ApiProperty` decorator instead.

Using the CLI plugin

To enable the plugin, open `nest-cli.json` (if you use Nest CLI) and add the following `plugins` configuration:

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

You can use the `options` property to customize the behavior of the plugin.

```
"plugins": [
  {
    "name": "@nestjs/swagger",
    "options": {
      "classValidatorShim": false,
      "introspectComments": true
    }
  }
]
```

The `options` property has to fulfill the following interface:

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

Option	Default	Description
`dtoFileNameSuffix`	`['.dto.ts', '.entity.ts']`	DTO (Data Transfer Object) files suffix
`controllerFileNameSuffix`	`.controller.ts`	Controller files suffix
`classValidatorShim`	`true`	If set to true, the module will reuse `class-validator` validation decorators (e.g. `@Max(10)` will add `max: 10` to schema definition)
`dtoKeyOfComment`	`'description'`	The property key to set the comment text to on `ApiProperty`.
`controllerKeyOfComment`	`'description'`	The property key to set the comment text to on `ApiOperation`.
`introspectComments`	`false`	If set to true, plugin will generate descriptions and example values for properties based on comments

Make sure to delete the `/dist` folder and rebuild your application whenever plugin options are updated. If you don't use the CLI but instead have a custom `webpack` configuration, you can use this plugin in combination with `ts-loader`:

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

SWC builder

For standard setups (non-monorepo), to use CLI Plugins with the SWC builder, you need to enable type checking, as described here.

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

For monorepo setups, follow the instructions here.

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

Now, the serialized metadata file must be loaded by the `SwaggerModule#loadPluginMetadata` method, as shown below:

```
import metadata from './metadata'; // <-- file auto-generated by the
"PluginMetadataGenerator"

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

Integration with `ts-jest` (e2e tests)

To run e2e tests, `ts-jest` compiles your source code files on the fly, in memory. This means, it doesn't use Nest CLI compiler and does not apply any plugins or perform AST transformations.

To enable the plugin, create the following file in your e2e tests directory:

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

module.exports.name = 'nestjs-swagger-transformer';
// you should change the version number anytime you change the
configuration below - otherwise, jest will not detect changes
module.exports.version = 1;

module.exports.factory = (cs) => {
  return transformer.before(
    {
      // @nestjs/swagger/plugin options (can be empty)
    },
    cs.program, // "cs.tsCompiler.program" for older versions of Jest (<=
v27)
  );
};
```

With this in place, import AST transformer within your `jest` configuration file. By default (in the starter application), e2e tests configuration file is located under the `test` folder and is named `jest-e2e.json`.

```
{
  ... // other configuration
  "globals": {
    "ts-jest": {
      "astTransformers": {
        "before": ["<path to the file created above>"]
      }
    }
  }
}
```

If you use `jest@^29`, then use the snippet below, as the previous approach got deprecated.

```
{
  ... // other configuration
  "transform": {
    "^.+\\.(t|j)s$": [
      "ts-jest",
      {
        "astTransformers": {
          "before": ["<path to the file created above>"]
        }
      }
    ]
  }
}
```

Troubleshooting `jest` (e2e tests)

In case `jest` does not seem to pick up your configuration changes, it's possible that Jest has already **cached** the build result. To apply the new configuration, you need to clear Jest's cache directory.

To clear the cache directory, run the following command in your NestJS project folder:

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

In case the automatic cache clearance fails, you can still manually remove the cache folder with the following commands:

```
# Find jest cache directory (usually /tmp/jest_rs)
# by running the following command in your NestJS project root
$ npx jest --showConfig | grep cache
# ex result:
#   "cache": true,
#   "cacheDirectory": "/tmp/jest_rs"

# Remove or empty the Jest cache directory
$ rm -rf  <cacheDirectory value>
# ex:
# rm -rf /tmp/jest_rs
```

Other features

This page lists all the other available features that you may find useful.

Global prefix

To ignore a global prefix for routes set through `setGlobalPrefix()`, use `ignoreGlobalPrefix`:

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

Global parameters

You can add parameter definitions to all routes using `DocumentBuilder`:

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

Multiple specifications

The `SwaggerModule` provides a way to support multiple specifications. In other words, you can serve different documentation, with different UIs, on different endpoints.

To support multiple specifications, your application must be written with a modular approach. The `createDocument()` method takes a 3rd argument, `extraOptions`, which is an object with a property named `include`. The `include` property takes a value which is 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 { AppModule } from './app.module';
import { CatsModule } from './cats/cats.module';
import { DogsModule } from './dogs/dogs.module';

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

  /**
   * createDocument(application, configurationOptions, extraOptions);
   *
   * createDocument method takes 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 the Swagger UI for cats:

In turn, `http://localhost:3000/api/dogs` will expose the Swagger UI for dogs:

Migration guide

If you're currently using `@nestjs/swagger@3.*`, note the following breaking/API changes in version 4.0.

Breaking changes

The following decorators have been changed/renamed:

- `@ApiModelProperty` is now `@ApiProperty`
- `@ApiModelPropertyOptional` is now `@ApiPropertyOptional`
- `@ApiResponseModelProperty` is now `@ApiResponseProperty`
- `@ApiImplicitQuery` is now `@ApiQuery`
- `@ApiImplicitParam` is now `@ApiParam`
- `@ApiImplicitBody` is now `@ApiBody`
- `@ApiImplicitHeader` is now `@ApiHeader`
- `@ApiOperation({{ '{' }} title: 'test' {{ '}' }})` is now `@ApiOperation({{ '{' }} summary: 'test' {{ '}' }})`
- `@ApiUseTags` is now `@ApiTags`

`DocumentBuilder` breaking changes (updated method signatures):

- `addTag`
- `addBearerAuth`
- `addOAuth2`
- `setContactEmail` is now `setContact`
- `setHost` has been removed
- `setSchemes` has been removed (use the `addServer` instead, e.g., `addServer('http://')`)

New methods

The following methods have been added:

- `addServer`
- `addApiKey`
- `addBasicAuth`
- `addSecurity`
- `addSecurityRequirements`