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>
```