## 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).