

Développement Backend

NestJS - Part 2

Valider votre API REST

Validation

- ▶ Pour valider automatiquement les demandes entrantes, **Nest** fournit un **ValidationPipe** intégré, qui utilise le puissant package **class-validator** et ses **décorateurs** de validation déclaratifs.
- ▶ Les règles spécifiques sont déclarées avec de simples **annotations** dans des déclarations de classe / DTO locales dans chaque module.
- ▶ <https://github.com/typestack/class-validator#validation-decorators>

```
async function bootstrap() {  
  const app = await NestFactory.create(ApplicationModule);  
  app.useGlobalPipes(new ValidationPipe());  
  await app.listen(3000);  
}  
bootstrap();
```

Auto-validation

```
@Post()  
create(@Body() createUserDto: CreateUserDto) {  
    return 'This action adds a new user';  
}
```

Auto-validation

```
import { IsEmail, IsNotEmpty } from 'class-validator';

export class CreateUserDto {
    @IsEmail()
    email: string;

    @IsNotEmpty()
    password: string;
}
```

Auto-validation

```
{  
    "statusCode": 400,  
    "error": "Bad Request",  
    "message": [  
        {  
            "target": {},  
            "property": "email",  
            "children": [],  
            "constraints": {  
                "isEmail": "email must be an email"  
            }  
        }  
    ]  
}
```

Auto-validation

```
@Get(':id')
findOne(@Param() params: FindOneParams) {
    return 'This action returns a user';
}
```

Auto-validation

```
import { IsNumberString } from 'class-validator';

export class FindOneParams {
    @IsNumberString()
    id: number;
}
```

Auto-validation

```
app.useGlobalPipes(  
    new ValidationPipe({  
        disableErrorMessages: true,  
    })  
);
```

Désactivation du détail des erreurs

Validation - Whitelist

- **ValidationPipe** peut également filtrer les propriétés qui ne devraient pas être reçues par le **handler**.
- Dans ce cas, nous pouvons ajouter les propriétés acceptables à la **whitelist**, et toute propriété non incluse dans la whitelist est **automatiquement supprimée** de l'objet obtenu.
- Lorsqu'il est défini sur **true**, cela supprimera **automatiquement** les propriétés non inscrites sur la **whitelist** (celles sans décorateur dans la classe de validation).

```
app.useGlobalPipes(  
  new ValidationPipe({  
    whitelist: true,  
  })  
);
```

Validation - Whitelist

- Vous pouvez également **arrêter** le traitement de la requête lorsque des propriétés **non incluses** dans la **whitelist** sont présentes et renvoyer une **réponse d'erreur** à l'utilisateur.
- Pour l'activer, définissez la propriété **forbidNonWhitelisted** sur **true**, en combinaison avec **whitelist** sur **true**.

Validation - Payloads objects transformation

- ▶ Les **payloads** arrivant sur le réseau sont de **simples objets JavaScript**.
- ▶ **ValidationPipe** peut transformer **automatiquement** les données utiles en **objets typés** en fonction de leurs classes **DTO**.
- ▶ Pour activer la transformation automatique, définissez **transform** sur **true**.

```
app.useGlobalPipes(  
  new ValidationPipe({  
    transform: true,  
  })  
);
```

Exercice 12

`git checkout -f step-12`

Valider les entrées de votre API

Exercice 12 : Input validation

- ▶ Vous devez valider les inputs du CRUD de l'API people
- ▶ Aidez-vous de la documentation en ligne de [class-validator](#) si besoin est.
- ▶ Testez avec POSTMAN

Solution

```
git checkout -f step-12-solution
```

Sérialisation

- Processus qui se produit avant que les **objets** ne soient renvoyés dans une **réponse réseau**.
- C'est un endroit approprié pour fournir des **règles de transformation et de nettoyage** des données à renvoyer au client.
- Effectuer ces transformations manuellement peut s'avérer fastidieux et sujet aux erreurs, et peut vous inciter à savoir si tous les cas ont été couverts.

Sérialisation

- ▶ **Nest** fournit une fonctionnalité intégrée permettant de s'assurer que ces opérations peuvent être effectuées de manière simple.
- ▶ L'intercepteur **ClassSerializerInterceptor** utilise le puissant package **class-transformer** pour fournir un moyen déclaratif et extensible de transformer des objets.
- ▶ L'opération de base qu'il effectue consiste à prendre la valeur renvoyée par un **handler** et à appliquer la fonction **classToPlain()** de **class-transformer**.
- ▶ Ce faisant, il peut appliquer les règles exprimées par les **décorateurs de class-transformer** sur une classe d'entité / DTO.
- ▶ <https://github.com/typestack/class-transformer>

```
import { Exclude } from 'class-transformer';

export class UserEntity {
    id: number;
    firstName: string;
    lastName: string;

    @Exclude()
    password: string;

    constructor(partial: Partial<UserEntity>) {
        Object.assign(this, partial);
    }
}
```

Exclure des propriétés

```
@UseInterceptors(ClassSerializerInterceptor)
@Get()
findOne(): UserEntity {
    return new UserEntity({
        id: 1,
        firstName: 'Kamil',
        lastName: 'Mysliwiec',
        password: 'password',
    });
}
```

Exclure des propriétés

```
{  
    "id": 1,  
    "firstName": "Kamil",  
    "lastName": "Mysliwiec"  
}
```

Exclure des propriétés

```
@Expose()  
get fullName(): string {  
    return `${this.firstName} ${this.lastName}`;  
}
```

Exposer des propriétés

```
@Transform(role => role.name)  
role: RoleEntity;
```

Transformer des propriétés

```
@SerializeOptions({
    excludePrefixes: ['_'],
})
@Get()
findOne(): UserEntity {
    return {};
}
```

Options de sérialisation

Exercice 13

`git checkout -f step-13`

Valider les sorties de votre API

Exercice 13 : Output validation

- ▶ Vous devez valider les outputs du CRUD de l'API people
- ▶ Aidez-vous de la documentation en ligne de [class-transformer](#) si besoin est.
- ▶ Testez avec POSTMAN

Solution

```
git checkout -f step-13-solution
```

Documentation

- ▶ La spécification [OpenAPI](#) (Swagger) est un format de définition puissant pour décrire les API RESTful.
- ▶ **Nest** fournit un **module** dédié pour l'utiliser.
- ▶ [@nestjs/swagger](#) et [fastify-swagger](#)

Documentation: bootstrap

- Initialisation de **Swagger** grâce au module **SwaggerModule** dans **main.ts**
- **DocumentBuilder** permet de structurer un **document** de base pour **SwaggerModule**.
- Méthodes permettant de définir des propriétés :
 - titre
 - description
 - version
 - etc.

Documentation: bootstrap

- Afin de créer un document complet (avec des routes HTTP définies), nous utilisons la méthode `createDocument()` de la classe **SwaggerModule**. Deux arguments, respectivement :
 - l'instance d'application
 - les options de base Swagger
- La dernière étape consiste à appeler `setup()`. Il accepte séquentiellement:
 - (1) le path pour monter l'instance de Swagger
 - (2) l'instance de l'application
 - (3) le document décrivant l'application Nest.

```
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();
```

bootstrap

swagger

Cats example 1.0

The cats API description

Schemes

HTTP ▾

cats ▾

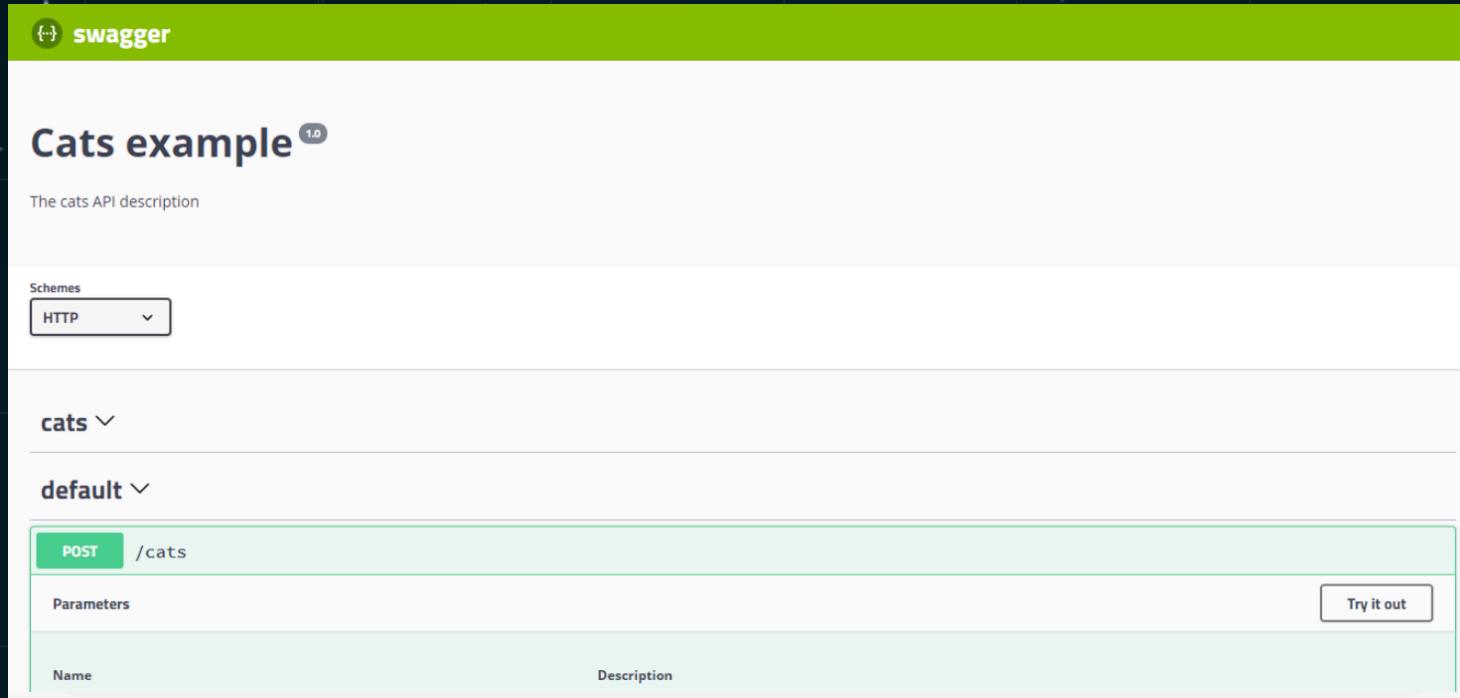
default ▾

POST /cats

Parameters

Try it out

Name	Description
------	-------------

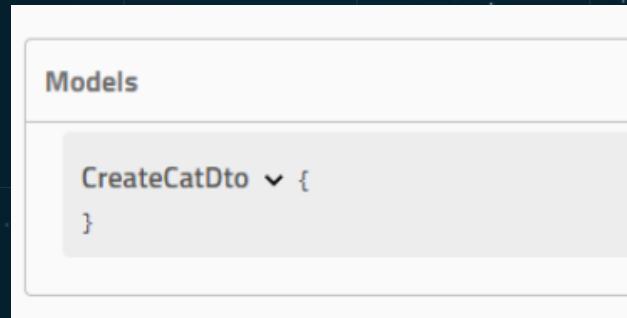


http://localhost:3000/api

Documentation: Body, query, path parameters

- Lors de l'examen des contrôleurs définis, le **SwaggerModule** recherche tous les décorateurs **@Body()**, **@Query()** et **@Param()** utilisés dans les **handlers**.
- Par conséquent, le **document valide** peut être créé.

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



@Body(): CreateCatDto

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

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

    @ApiProperty()
    age: number;

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

Models

```
CreateCatDto ▾ {  
  name      string  
  age       number  
  breed     string  
}
```

@ApiProperty()

```
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);
```

Multiple spécifications

Documentation: Tags

- Au début, nous avons créé un tag cats (en utilisant **DocumentBuilder**).
- Pour attacher le contrôleur au tag spécifié, nous devons utiliser le **décorateur** `@ApiTags(... tags)`.

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

Documentation: Réponse

- ▶ Pour définir une réponse HTTP personnalisée, nous utilisons le **décorateur @ApiResponse()**.

```
@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);  
}
```

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

Réponse : exemple de décorateurs

Documentation: Réponse

- En utilisant les **décorateurs** spécifiques

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

Documentation: Réponse

- Pour spécifier un modèle de retour pour les requêtes, il faut créer une classe et annoter toutes les propriétés avec le décorateur **@ApiProperty()**.

```
export class Cat {  
    @ApiProperty()  
    id: number;  
  
    @ApiProperty()  
    name: string;  
  
    @ApiProperty()  
    age: number;  
  
    @ApiProperty()  
    breed: string;  
}
```

```
@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);
  }
}
```



Réponse typée

<code>@ApiOperation()</code>	Method
<code>@ApiResponse()</code>	Method / Controller
<code>@ApiProduces()</code>	Method / Controller
<code>@ApiConsumes()</code>	Method / Controller
<code>@ApiBearerAuth()</code>	Method / Controller
<code>@ApiOAuth2()</code>	Method / Controller
<code>@ApiBasicAuth()</code>	Method / Controller
<code>@ApiSecurity()</code>	Method / Controller
<code>@ApiExtraModels()</code>	Method / Controller

Documentations décorateurs

<code>@ApiBody()</code>	Method
<code>@ApiParam()</code>	Method
<code>@ApiQuery()</code>	Method
<code>@ApiHeader()</code>	Method / Controller
<code>@ApiExcludeEndpoint()</code>	Method
<code>@ApiTags()</code>	Method / Controller
<code>@ApiProperty()</code>	Model
<code>@ApiPropertyOptional()</code>	Model
<code>@ApiHideProperty()</code>	Model
<code>@ApiExtension()</code>	Method

Documentations décorateurs

Exercice 14

`git checkout -f step-14`

Documenter votre API

Exercice 14 : Documentation

- Ajouter le module `Swagger`
- La route `helloWorld` ne doit pas être documentée
- Documenter l'API `people`
- Tester votre documentation par l'interface `Swagger` à l'adresse `/documentation`
- Documentation officielle du module: <https://docs.nestjs.com/openapi/introduction>

Solution

```
git checkout -f step-14-solution
```

Stockage des données

Installation

- MongoDB
- Robo 3T
- La façon la plus facile d'installer MongoDB est d'utiliser Docker

```
version: '2'
services:
  mongo:
    image: mongo
    container_name: mongo
    volumes:
      - /your/local/path/mongodb/data:/data/db
    ports:
      - 27017:27017
    restart: always
```

MongoDB : Introduction

- **MongoDB** est un système de stockage **NoSQL** orienté document.
- Il fonctionne en mode **client-serveur** via une communication TCP/IP (par défaut sur le port **27017**).
- **MongoDB** permet de sauvegarder, lire, modifier et supprimer des **documents** au format **BSON**. Il s'agit d'une extension du format JSON.

MongoDB : BSON

Le format **BSON**, propre à MongoDB, est une extension de JSON ajoutant :

- Un stockage des données au format **binaire** :
 - plus performant en lecture
 - gain de place au stockage
- Des types de données supplémentaires par rapport à JSON

MongoDB : BSON

- String
- Entier
- Double
- Date
- Données binaires (byte array)
- Booléen
- « Null »
- BSON Object
- BSON Array

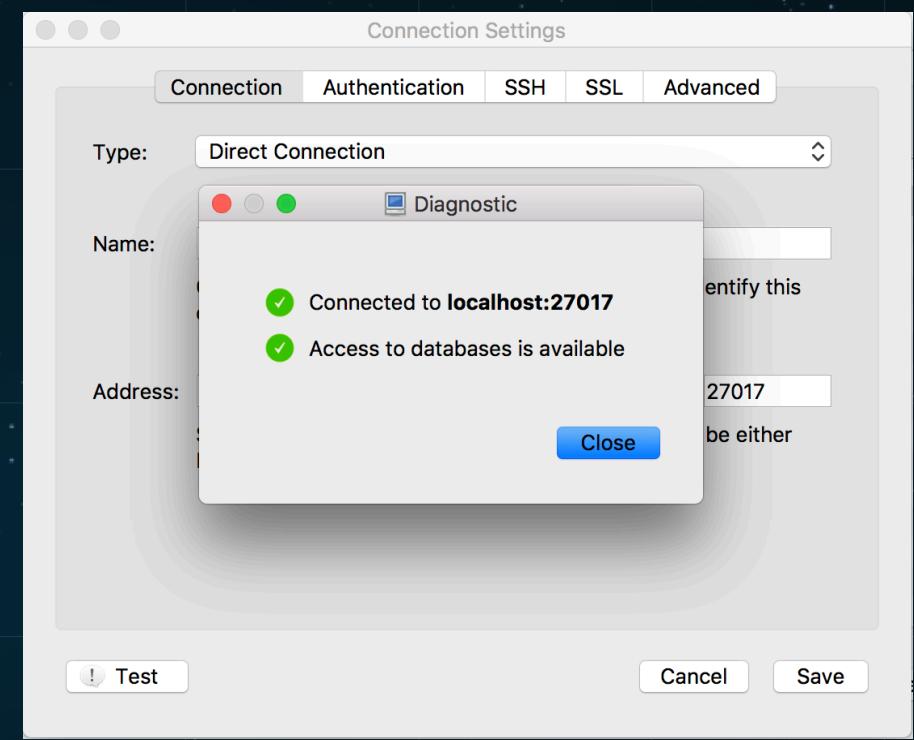
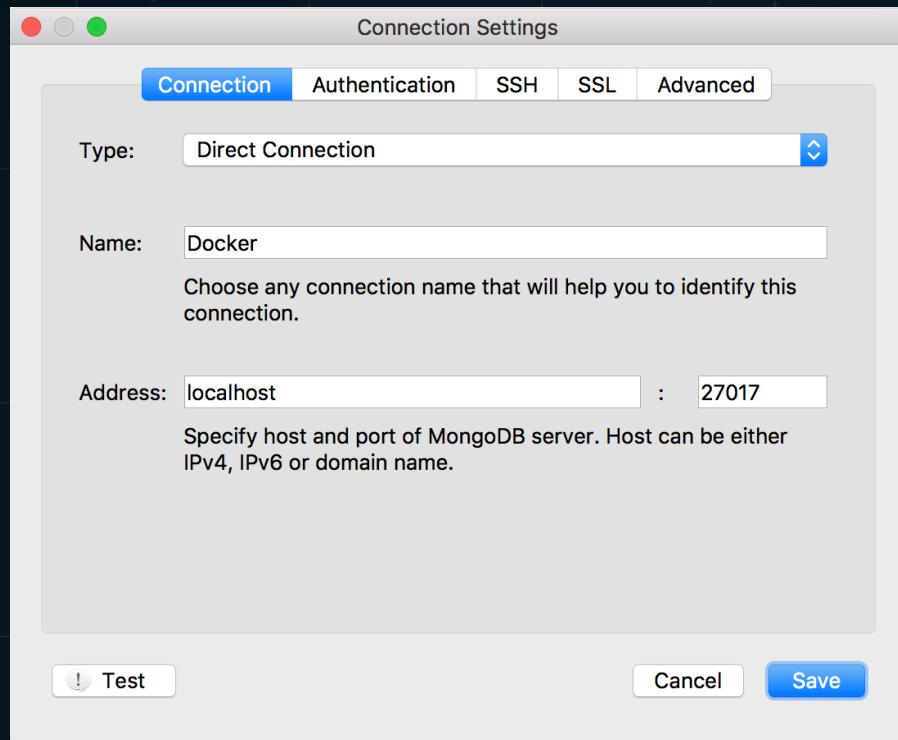
```
{  
  "name": "MongoDB",  
  "version": {  
    "major": 3,  
    "minor": 2  
  },  
  "load": 0.8  
  "bytes": BinData(2, "JnkYp0MUwsIiFg=="),  
  "startDate": ISODate("2016-02-10T01:00:00.338Z"),  
  "started": true,  
  "tags": ["storage", "db", "engine"],  
  "nothing": null  
}
```

MongoDB : Concept de base

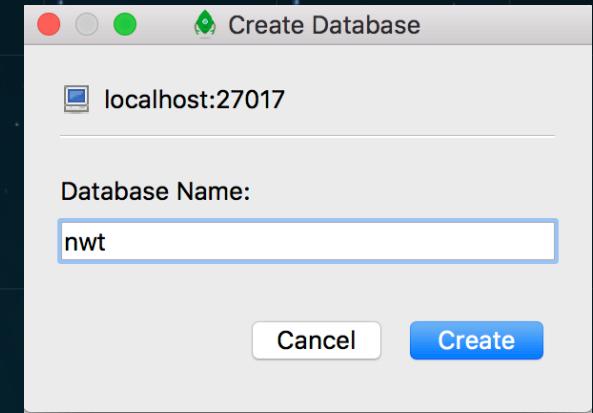
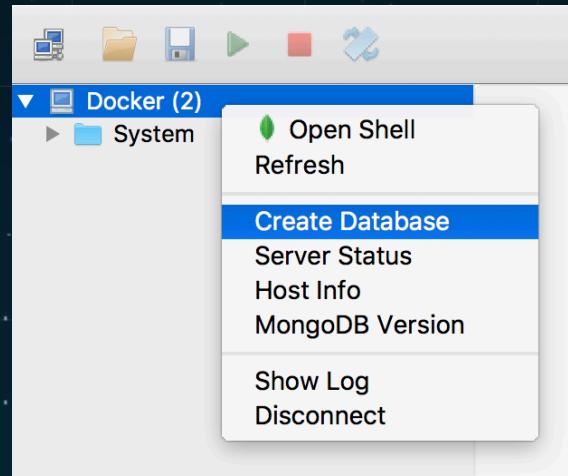
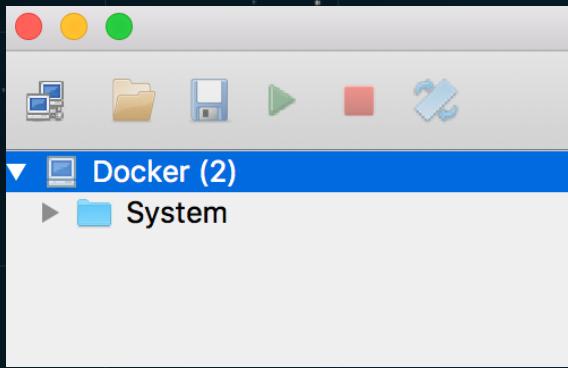
- Une instance de **MongoDB** est composée de base de données.
- Chaque base est composée de **collections**.
- Dans les collections se trouvent les **documents**.
- Chaque document possède un **identifiant unique** qui peut être auto-généré ou fourni par l'utilisateur.

MongoDB : Documentation

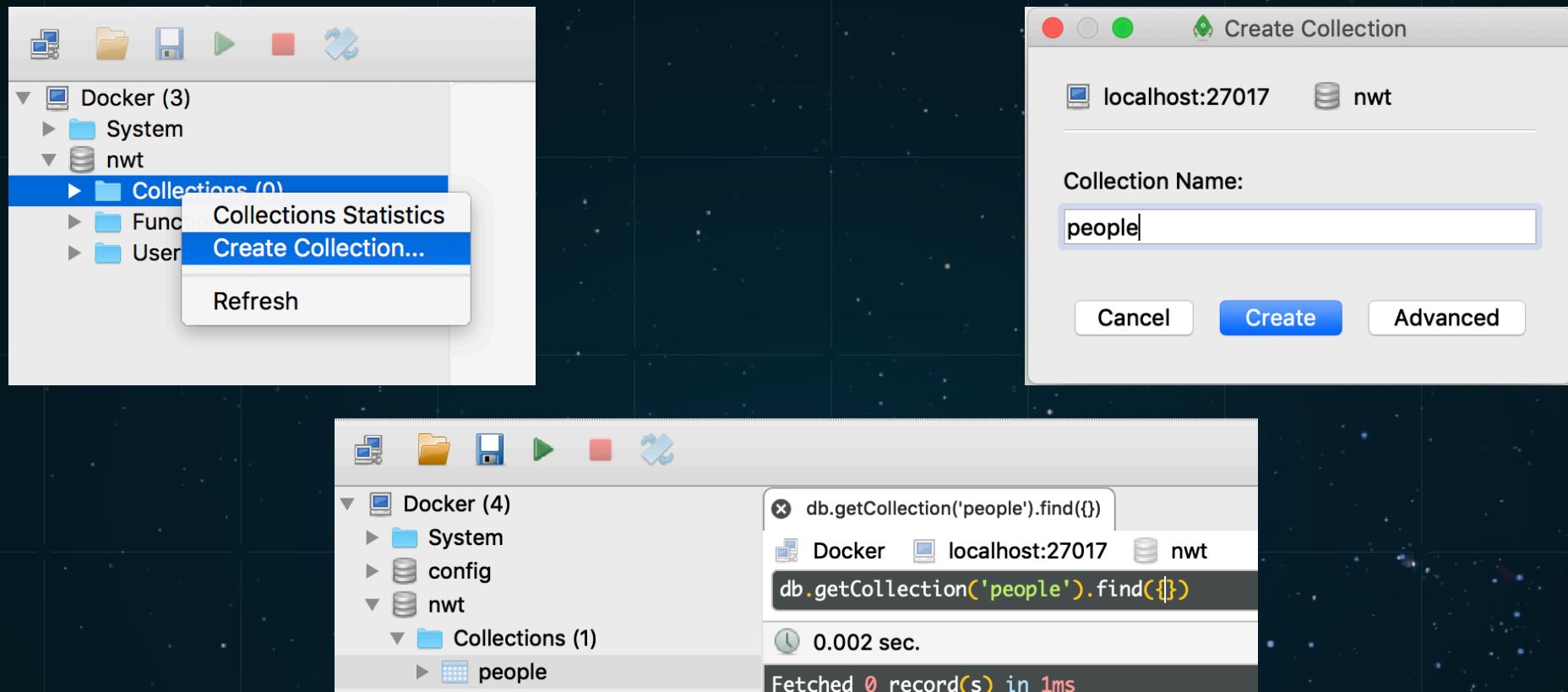
- Utilisation de [mongo Shell](#)
- Opération de [CRUD](#)
- [Indexes](#)
- [Replication](#)
- [Sharding](#)



Robo 3T : Connexion



Robo 3T : Créer une DB



Robo 3T : Créer une collection

Insertion des données initiales

- Vous devez insérer les données initiales dans la collection `people`
- Utiliser le script du projet fourni `./scripts/init.mongo.js`
- Analysez le et comparez le aux données statiques précédemment utilisées
- Utiliser le script d'index fourni `./scripts/index.mongo.js`

MongoDB Extension

MongoDB Extension

- **Nest** intègre **Mongoose** l'outil de modélisation d'objet **MongoDB** le plus populaire.
- `@nestjs/mongoose` et `mongoose`
- Une fois le processus d'installation terminé, nous pouvons importer le module **Mongoose** dans **AppModule**.
- La méthode `forRoot()` accepte le même objet de configuration que `mongoose.connect()` du paquet **Mongoose**, comme décrit [ici](#).

app.module.ts

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';

@Module({
  imports: [MongooseModule.forRoot('mongodb://localhost/nest')],
})
export class AppModule {}
```

MongoDB connexion

Exercice 15

`git checkout -f step-15`

Connexion à MongoDB

Exercice 15 : MongoDB connection

- ▶ La configuration MongoDB a été ajoutée
- ▶ Importer le module mongo

Solution

```
git checkout -f step-15-solution
```



Créer vos propres modèles

Définition d'un modèle

- ▶ Avec Mongoose, tout est dérivé d'un **schéma**
- ▶ Afin de récupérer les données en **JSON** et de pouvoir utiliser les alias et les méthodes virtuelles de **mongoose**, il faut passer une option à notre schéma:
 - ▶ `{toJSON: { virtuals: true, transform: (doc: any, ret: any) => { delete ret._id } }}`
- ▶ Dorénavant, vous pouvez appeler la méthode **toJSON()** sur le résultat de vos requêtes.
- ▶ Afin de ne pas stocker une donnée inutile sur les versions des documents, une option doit également être passée:
 - ▶ `{versionKey: false}`

schemas/cat.schema.ts

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { Document } from 'mongoose';

export type CatDocument = Cat & Document;

@Schema()
export class Cat {
    @Prop()
    name: string;

    @Prop()
    age: number;

    @Prop()
    breed: string;
}

export const CatSchema = SchemaFactory.createForClass(Cat);
```

Example de schéma

cats.module.ts

```
import { Module } from '@nestjs/common';
import { MongooseModule } from '@nestjs/mongoose';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';
import { Cat, CatSchema } from './schemas/cat.schema';

@Module({
  imports: [MongooseModule.forFeature([{ name: Cat.name, schema: CatSchema }])],
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

Enregistrement du schéma

Exercice 16

`git checkout -f step-16`

Créer le schéma Person

Exercice 16 : Person schema

- Créer le schéma PersonSchema en ajoutant l'option pour utiliser la méthode `toJSON()` et supprimer la version :
- Définissez le schema de données `mongoose` en se basant sur l'entity utilisée précédemment
- Déclarer le modèle dans le module
- Adapter les DTO pour que la validation de `managerId` soit considérée comme un `mongoid`.
- Faites de même pour la validation de l'`id` d'une personne.

Solution

```
git checkout -f step-16-solution
```

Gérer vos DAO

Définition d'un DAO

- Une simple classe exportée
- Un décorateur `@Injectable()`
- Une propriété **privée** permettant d'injecter le model

```
constructor(@InjectModel(Cat.name) private catModel: Model<CatDocument>) {}
```

- Pour créer un **DAO** avec le CLI il faut utiliser le type **provider** et non service
- Implémentation des méthodes de **CRUD** qui appellent les **fonctions natives** de l'adaptateur
- Dans le cas de **mongoose**, les fonctions retournent des **Promises**

Exercice 17

`git checkout -f step-17`

Retourner la liste de people

Exercice 17 : People list

- Créer le provider `PeopleDao`
- Implémenter la méthode `.find()` qui retournera la liste des people de la db
- Utiliser la méthode native de `mongoose` qui retourne une `promise` et dont les documents sont des `PersonDocument`
- Changer la méthode `.findAll()` du service `PeopleService` afin d'appeler la méthode du document
- Penser à utiliser la méthode `.toJSON()` sur chaque document pour les formater correctement avant de les retourner

Solution

```
git checkout -f step-17-solution
```

Exercice 18

`git checkout -f step-18`

Retourner une personne

Exercice 18 : One person

- Implémenter la méthode `.findById()` qui retournera une personne de la db
- Utiliser la méthode native de `mongoose` qui retourne une promise et dont le document est un `PersonDocument`
- Changer la méthode `.findOne()` du service `PeopleService` afin d'appeler la méthode du DAO
- La partie `NotFound` doit rester dans `PeopleService`
- Penser à utiliser la méthode `.toJSON()` sur le document pour le formater correctement avant de le retourner
- Si la personne n'est pas trouvée, le document doit renvoyer un observable `undefined`
- Si une `erreur` se produit lors de la requête, vous devez la `catcher` et le service doit renvoyer un observable d'erreur avec la méthode `throwError()`
- Utiliser la classe native de `Nest UnprocessableEntityException` pour afficher le message d'erreur

Solution

```
git checkout -f step-18-solution
```

Exercice 19

`git checkout -f step-19`

Créer une personne

Exercice 19 : Create person

- Implémenter la méthode `.save()` qui retournera une personne créée dans la db
- Utiliser la méthode native de `mongoose` qui retourne une `promise` et dont le document est un `PersonDocument`
- Changer la méthode `.create()` du service `PeopleService` afin d'appeler la méthode du document
- La partie `response` doit rester dans `PeopleService`
- Penser à utiliser la méthode `.toJSON()` sur le document pour le formater correctement avant de le retourner
- La vérification si la personne existe déjà se fait grâce à la méthode native des indexées de `mongoose` qui renvoie un code `11000`
- Si la personne existe la méthode doit renvoyer un observable d'erreur avec la méthode `throwError()`
- Utiliser la classe native de `Nest ConflictException` pour afficher le message d'erreur et en cas d'une autre erreur, la classe native de `Nest UnprocessableEntityException` pour afficher le message

Solution

```
git checkout -f step-19-solution
```

Exercice 20

`git checkout -f step-20`

Mettre à jour une personne

Exercice 20 : Update person

- Implémenter la méthode `.findByIdAndUpdate()` qui retournera la `personne modifiée` dans la db
- Utiliser la méthode native de `mongoose` qui retourne une `promise` et dont le document est un `PersonDocument`
- Changer la méthode `.update()` du service `PeopleService` afin d'appeler la méthode du document
- La partie `NotFound` doit rester dans `PeopleService`
- Penser à utiliser la méthode `.toJSON()` sur le document pour le formater correctement avant de le retourner
- Si la personne n'est pas trouvée, le document doit renvoyer un observable `undefined`
- La vérification si la personne existe déjà se fait grâce à la méthode native des indexes de `mongoose` qui renvoie un code `11000`
- Si une `erreur` se produit lors de la requête, vous devez la `catcher` et le service doit renvoyer un observable d'erreur avec la méthode `throwError()`
- Utiliser la classe native de `Nest ConflictException` pour afficher le message d'erreur et en cas d'une autre erreur, la classe native de `Nest UnprocessableEntityException` pour afficher le message

Solution

```
git checkout -f step-20-solution
```

Exercice 21

`git checkout -f step-21`

Supprimer une personne

Exercice 21 : Delete person

- Implémenter la méthode `.findByIdAndRemove()` qui retournera la **personne supprimée** dans la db
- Utiliser la méthode native de `mongoose` qui retourne une **promise** et dont le document est un `PersonDocument`
- Changer la méthode `.delete()` du service `PeopleService` afin d'appeler la méthode du document
- La partie `NotFound` doit rester dans `PeopleService`
- Penser à utiliser la méthode `.toJSON()` sur `le document` pour le formater correctement avant de le retourner
- Si la personne n'est pas trouvée, `le document` doit renvoyer un `observable undefined`
- Si une **erreur** se produit lors de la requête, vous devez la `catcher` et le service doit renvoyer un observable d'erreur avec la méthode `throwError()`
- Utiliser la classe native de `Nest UnprocessableEntityException` pour afficher le message d'erreur

Solution

```
git checkout -f step-21-solution
```



Fin de la 5ème journée et du développement backend
Si vous avez apprécié la formation, envoyez un Tweet !

#NewWebTechnologies #nestjs @_akanass_ @TaDaweb @nestframework