Développement Backend

NestJS - Part 1

Installation - 2ème étape

Installation de @nestjs/cli

Une fois Node.js et Yarn installés sur votre machine, lors de la 1ère étape, vérifiez les versions en tapant dans votre terminal

```
$ yarn -v
=> 1.22.19

$ node -v
=> v18.3.0
```

Vous pouvez maintenant installer @nestjs/cli grâce à Yarn en tapant dans votre terminal

```
$ yarn global add @nestjs/cli
```

Déroulement de la formation

Github de la formation

https://github.com/simonh88/nwt-school-back

Déroulement de la formation

$ git clone https://github.com/simonh88/nwt-school-back

Installation des dépendances du projet

Une fois que vous avez cloné le projet dans votre répertoire de travail, récupérez les PDFs dans le dossier « documents » et copiez les à un autre endroit sur votre disque

Maintenant, vous devez passer sur la branche step-01 du projet en tapant dans votre terminal

```
$ git checkout -f step-01
```

Une fois sur cette branche, vous devez installer les dépendances du projet en tapant dans votre terminal

```
$ yarn install
```

Installation des dépendances du projet

Une fois que vous avez installé les dépendances du projet, vous pouvez vérifier toutes les versions en tapant dans votre terminal

$ nest info

```
[System Information]
OS Version     : macOS Monterey
NodeJS Version : v17.0.1
YARN Version    : 1.22.17

[Nest CLI]
Nest CLI Version : 8.1.4

[Nest Platform Information]
platform-fastify version : 8.2.1
schematics version       : 8.0.4
mongoose version         : 9.0.1
swagger version          : 5.1.4
testing version          : 8.2.1
common version           : 8.2.1
core version             : 8.2.1
cli version              : 8.1.4
```

Déroulement de la formation

Un concept clé NestJS

Un TP

- Une branche d’exercice : step-XX
- Une branche de solution : step-XX-solution

Quickstart

La stack « officielle »

- NestJS : https://nestjs.com/
- NestJS CLI : https://docs.nestjs.com/cli/overview
- Fichier de configuration : nest-cli.json

$ nest new my-awesome-app

- génère l'arborescence de l'application
- initialise un repo Git + 1er commit
- Demande de choisir le package manager: NPM ou YARN
- installe les deps NPM

La stack « officielle »

$ nest generate module user

- génère le répertoire et le fichier d'un module
  - src/user/user.module.ts
- met à jour le fichier src/app.module.ts avec la référence du nouveau module

La stack « officielle »

$ nest generate service user

- génère les fichiers d'un service
  - src/user/user.service.ts
  - src/user/user.service.spec.ts

$ nest generate service user shared

- génère les fichiers d'un service
  - src/shared/user/user.service.ts
  - src/shared/user/user.service.spec.ts

- met à jour le fichier src/app.module.ts avec la référence du nouveau service

La stack « officielle »

```
$ nest generate class
$ nest generate controller
$ nest generate decorator
$ nest generate filter
$ nest generate guard
$ nest generate interceptor
$ nest generate interface
$ nest generate module
$ nest generate pipe
$ nest generate service
…

$ nest start —watch
$ nest build
…
```

Langage utilisé

Typescript

- ES6+
- Types (optionnels)
- Annotations

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {
  constructor(private readonly catsService: CatsService) {}
}
```

TSC

- Bundle en JavaScript
- Hot reload
- Choix par défaut de NestJS

```
$ yarn run start:dev
```

Architecture Globale

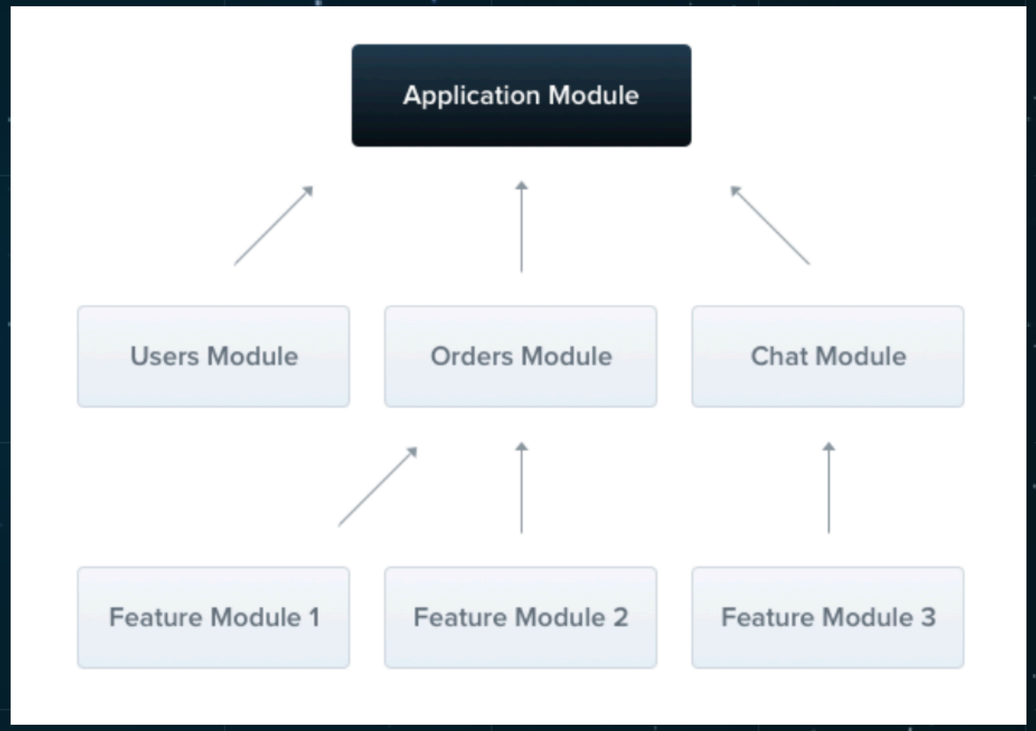
Framework

- NestJS est un framework Node.js
- Il reprend les concepts clés d'Angular
  - Pipes
  - Guards
  - Interceptors
  - Modules
  - Services
  - DI
  - Annotations
- Le core peut être étendu grâce à des extensions: HTTP, Websocket, MongoDB, Logger, …

Module

Un module

- Permet de regrouper des fonctionnalités
- Au moins un module par application
- Il est chargé de façon asynchrone
- Différents types de modules
  - Root (App) module
  - Feature Module
  - Shared module
  - Core module



app.module.ts

```
import { Module } from '@nestjs/common';
import { CatsModule } from './cats/cats.module';

@Module({
  imports: [CatsModule],
})
export class AppModule {}
```

Exemple d’un module

```
import { NestFactory } from '@nestjs/core';
import {
  FastifyAdapter,
  NestFastifyApplication,
} from '@nestjs/platform-fastify';
import { ApplicationModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    ApplicationModule,
    new FastifyAdapter()
  );
  await app.listen(3000);
}
bootstrap();
```

Bootstrap

cats/cats.module.ts

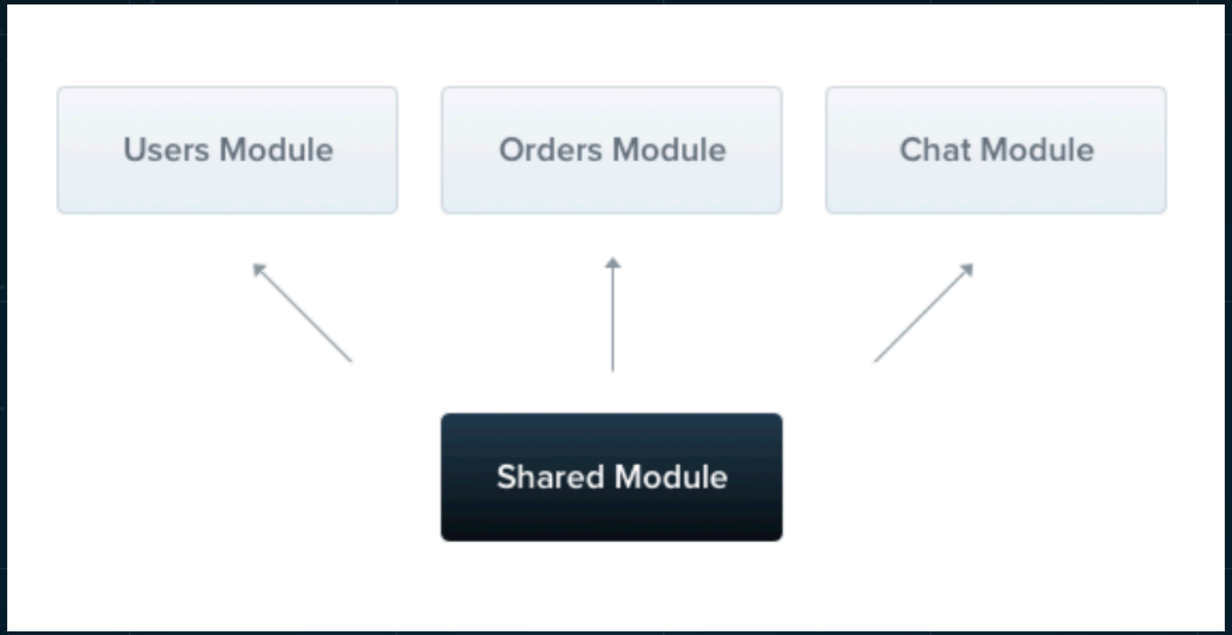
```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

Feature Module

```
src
├── cats
│   ├── dto
│   │   └── create-cat.dto.ts
│   ├── interfaces
│   │   └── cat.interface.ts
│   ├── cats.service.ts
│   ├── cats.controller.ts
│   └── cats.module.ts
├── app.module.ts
└── main.ts
```

Structure des applications



Shared Module

cats.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService]
})
export class CatsModule {}
```

Shared Module

```
import { Module, Global } from '@nestjs/common';
import { CatsController } from './cats.controller';
import { CatsService } from './cats.service';

@Global()
@Module({
  controllers: [CatsController],
  providers: [CatsService],
  exports: [CatsService],
})
export class CatsModule {}
```

Global Module

Exercice 1 : prise en main

git checkout -f step-01

Mise en place de votre première application

Exercice 1 : prise en main

- Créer un module AppModule
  - src/app.module.ts
- Configurer le bootstrap de l’application : src/main.ts

Solution

```
git checkout -f step-01-solution
```

Extensions

Etendre les fonctionnalités du core

- Permet d'ajouter des fonctionnalités au core
- Elles permettent d'initialiser et configurer le module ou les éléments associés une et une seule fois
- Différents types d'extensions sont déjà développées :
  - Websocket
  - MongoDB
  - RabbitMQ
  - gRPC
  - Logger
  - Redis
  - Validation
  - Caching
  - .....

app.module.ts

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

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

Application Module

Activer le logger de base

- Il est possible d'activer de manière très simple le logger de base afin d'avoir ensuite accès au service Logger n'importe où grâce à la DI.

- C'est ce service qui affiche les messages lorsque vous démarrez votre application.

```
new FastifyAdapter({ logger: true });
```

Activer le logger de base

- Lors du bootstrap, le service n'est pas encore accessible, il faut donc appeler l'objet de base fourni par NestJS.

```
import { NestFactory } from '@nestjs/core';
import { Logger } from '@nestjs/common';
import { FastifyAdapter, NestFastifyApplication } from '@nestjs/platform-fastify';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter( instanceOrOptions: { logger: true }),
  );
  await app.listen( port: 3000);
  Logger.log( message: `Application served at http://localhost:3000`,  context: 'bootstrap');
}

bootstrap();
```

Exercice 2 : logger extension

```
git checkout -f step-02
```

Activer l’extension logger

Exercice 2 : logger extension

- Activer l’extension logger dans votre application
- Afficher le message suivant au moment du bootstrap:
  - « Application served at http://localhost:3000 »

Solution

```
git checkout -f step-02-solution
```

Configuration

Configuration

- NestJS utilise les variables d'environnement par défaut
- Il possède un système de configuration basé sur ce processus: https://docs.nestjs.com/techniques/configuration

- Comme le montre la documentation, l'utilisation peut être assez complexe.

Configuration

- Nous allons utiliser la librairie node-config:
  - Plus facile d'utilisation
  - Très répandu dans le milieu Node.js
  - Permet d'utiliser les fichiers YAML

- Il faut uniquement créer un répertoire config à la racine et y placer un fichier default.yml
- Ce fichier, et donc la configuration, pourra être surchargé facilement en créant des fichiers suivant les environnements.
- Néanmoins, lors du build final, le répertoire config devra être copié à l'intérieur du projet dist final car la librairie cherche des données à la racine du projet exécuté.

```yaml
server:
  host: 0.0.0.0
  port: 4443
```

default.yml

```
import { NestFactory } from '@nestjs/core';
import { Logger } from '@nestjs/common';
import { FastifyAdapter, NestFastifyApplication } from '@nestjs/platform-fastify';
import { AppModule } from './app.module';
import * as Config from 'config';

async function bootstrap(config) {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    new FastifyAdapter( instanceOrOptions: { logger: true }),
  );
  await app.listen(config.port, config.host);
  Logger.log( message: `Application served at http://${config.host}:${config.port}`, context: 'bootstrap');
}

bootstrap(Config.get('server'));
```

Bootstrap

Exercice 3 : Configuration

```
git checkout -f step-03
```

Configuration dynamique

Exercice 3 : configuration dynamique

- Créer le fichier de configuration contenant les données relatives au server:
  - host: 0.0.0.0
  - port: 3000

- Utiliser cette configuration lors du bootstrap afin de ne plus avoir de données statiques

Solution

```
git checkout -f step-03-solution
```

Créer vos propres controllers

Définition d'un controller

- Les contrôleurs sont responsables du traitement des demandes entrantes et du renvoi des réponses au client.

- Le mécanisme de routage contrôle quel contrôleur reçoit quelles demandes.
- Fréquemment, chaque contrôleur dispose de plusieurs routes et différentes routes peuvent effectuer différentes actions.

- Pour créer un contrôleur de base, nous utilisons des classes et des décorateurs.
- Les décorateurs associent les classes aux métadonnées requises et permettent à Nest de créer une carte de routage (associer des demandes aux contrôleurs correspondants).

Définition d'un controller

- Une simple classe exportée
- Un décorateur @Controller()
- La méta-donnée optionnelle:
  - path
- Une méthode handler avec un décorateur permettant d'effectuer le traitement associé

cats.controller.ts

```
import { Controller, Get } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
```

Définition d’un controller

- Lorsqu'un contrôleur renvoie un objet ou un tableau JavaScript, il sera automatiquement sérialisé en JSON.

- Lorsqu'il renvoie un type primitif JavaScript (chaîne, nombre, booléen, par exemple), Nest envoie uniquement la valeur sans tenter de la sérialiser.

- Cela simplifie le traitement des réponses: il suffit de renvoyer la valeur et Nest s'occupe du reste.

Définition d’un controller

- Lorsque le contrôleur est entièrement défini, Nest ne sait toujours pas que CatsController existe et par conséquent, il ne créera pas d'instance de cette classe.

- Les contrôleurs appartiennent toujours à un module, c'est pourquoi nous incluons le tableau de contrôleurs dans le décorateur @Module ().

app.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';

@Module({
  controllers: [CatsController],
})
export class AppModule {}
```

Enregistrement d’un contrôleur

Request Object

- Les handlers ont souvent besoin d'accéder aux détails de la requête du client.
- Nest fournit un accès à l'objet de requête de la plateforme sous-jacente. (Fastify ou Express)
- Nous pouvons accéder à l'objet de requête en demandant à Nest de l'injecter en ajoutant le décorateur @Req() à la signature du handler.

```
import { Controller, Get, Req } from '@nestjs/common';
import { FastifyRequest } from 'fastify';

@Controller( prefix: 'hello')
export class HelloController {
  @Get()
  sayHello(@Req() request: FastifyRequest): string {
    return 'world';
  }
}
```

Request Object avec Fastify

Request Object

- Représente la requête HTTP et possède des propriétés :
  - Query string
  - Paramètres
  - Headers HTTP
  - Body

- Nous pouvons utiliser des décorateurs dédiés, tels que @Body() ou @Query()

```
@Param(key?: string)
```

```
@Body(key?: string)
```

```
@Query(key?: string)
```

```
@Headers(name?: string)
```

```
@Ip()
```

Response Object

- La deuxième façon de manipuler la réponse consiste à utiliser un objet de réponse spécifique à la plateforme sous-jacente. (Fastify ou Express).

- Nous pouvons accéder à l'objet de requête en demandant à Nest de l'injecter en ajoutant le décorateur @Res() à la signature du handler.

```
import { Controller, Get, Res } from '@nestjs/common';
import { FastifyReply } from 'fastify';
import { ServerResponse } from 'http';

@Controller( prefix: 'hello')
export class HelloController {
  @Get()
  sayHello(@Res() response: FastifyReply<ServerResponse>): void {
    response.send( payload: 'world');
  }
}
```

Response Object avec Fastify

Response Object

- Cette approche permet plus de flexibilité en offrant un contrôle total sur l'objet de réponse (manipulation des en-têtes, fonctionnalités spécifiques à la bibliothèque, etc.)

- MAIS elle doit être utilisée avec précaution.

- Moins claire et présente certains inconvénients:
  - Perte de compatibilité avec les fonctionnalités de Nest :
    - Intercepteurs
    - Décorateur @HttpCode().
  - Votre code peut devenir dépendant de la plate-forme
  - Plus difficile à tester

- L'approche standard Nest doit toujours être privilégiée lorsque cela est possible.

Resources

- Nest fournit des décorateurs de requête HTTP standard :
  - @Get ()
  - @Post ()
  - @Put ()
  - @Delete ()
  - @Patch ()
  - @Options ()
  - @Head ()
  - @All ()

- Chacun représente sa méthode de requête HTTP respective.

cats.controller.ts

```
import { Controller, Get, Post } from '@nestjs/common';

@Controller('cats')
export class CatsController {
  @Post()
  create(): string {
    return 'This action adds a new cat';
  }

  @Get()
  findAll(): string {
    return 'This action returns all cats';
  }
}
```

Status Code

- Le status code de la réponse est toujours 200 par défaut, à l'exception des demandes POST qui sont 201.
- Nous pouvons facilement modifier ce comportement en ajoutant le décorateur @HttpCode (...) au niveau du handler.
- Le status code n'est pas toujours statique, mais dépend de divers facteurs.
- Utiliser le response object :
  - appeler directement res.status()
  - en cas d'erreur, émettre une exception

```
@Post()
@HttpCode(204)
create() {
  return 'This action adds a new cat';
}
```

Headers

- Pour spécifier un header personnalisé, vous pouvez utiliser un décorateur @Header()

- ou le response object :
  - appeler directement res.header().

```
@Post()
@Header('Cache-Control', 'none')
create() {
  return 'This action adds a new cat';
}
```

Redirect response avec Fastify

- Utiliser le response object :
  - appeler directement res.status(302).redirect(…).

```
@Get()
index(@Res() res) {
  res.status(302).redirect('/login');
}
```

Route parameters

- @Param() est utilisé pour décorer un paramètre de méthode.

- Nous pouvons accéder au paramètre id en référençant params.id.

```
@Get(':id')
findOne(@Param() params): string {
  console.log(params.id);
  return `This action returns a #${params.id} cat`;
}
```

Route parameters

- Vous pouvez également transmettre un token de paramètre particulier au décorateur, puis référencer le paramètre route directement par son nom dans le corps de la méthode.

```
@Get(':id')
findOne(@Param('id') id): string {
  return `This action returns a #${id} cat`;
}
```

Asynchronicité

- Nous aimons le JavaScript moderne et nous savons que l'extraction de données est principalement asynchrone.
- C'est pourquoi Nest prend en charge et fonctionne bien avec les fonctions asynchrones.
- Les handlers Nest sont encore plus puissants car ils peuvent renvoyer des flux observables RxJS.
- Nest s'abonnera automatiquement à la source et prendra la dernière valeur émise (une fois le flux terminé).

cats.controller.ts

```
@Get()
findAll(): Observable<any[]> {
  return of([]);
}
```

Asynchronicité

Request payloads

- Notre exemple précédent d'handler POST n'acceptait aucun paramètre client.
- Corrigeons cela en ajoutant le décorateur @Body().
- Nous devons d'abord déterminer le schéma DTO (Data Transfer Object):
  - Objet qui définit la manière dont les données seront envoyées sur le réseau.
  - Le schéma DTO peut-être une interface TypeScript ou une simple classe.

create-cat.dto.ts

```
export class CreateCatDto {
  readonly name: string;
  readonly age: number;
  readonly breed: string;
}
```

cats.controller.ts

```
@Post()
async create(@Body() createCatDto: CreateCatDto) {
  return 'This action adds a new cat';
}
```

Request payloads

cats.controller.ts

```
import { Controller, Get, Query, Post, Body, Put, Param, Delete } from '@nestjs/common';
import { CreateCatDto, UpdateCatDto, ListAllEntities } from './dto';

@Controller('cats')
export class CatsController {
  @Post()
  create(@Body() createCatDto: CreateCatDto) {
    return 'This action adds a new cat';
  }

  @Get()
  findAll(@Query() query: ListAllEntities) {
    return `This action returns all cats (limit: ${query.limit} items)`;
  }

  @Get(':id')
  findOne(@Param('id') id: string) {
    return `This action returns a #${id} cat`;
  }

  @Put(':id')
  update(@Param('id') id: string, @Body() updateCatDto: UpdateCatDto) {
    return `This action updates a #${id} cat`;
  }

  @Delete(':id')
  remove(@Param('id') id: string) {
    return `This action removes a #${id} cat`;
  }
}
```

Exercice 4

git checkout -f step-04

Créez votre propre controller

Exercice 4 : créez votre propre controller

- Utiliser le CLI pour créer un module hello qui contient un controller du même nom
  - nest g module hello
  - nest g controller hello —no-spec
- Ce controller doit avoir un handler GET qui permettra d'appeler une route accessible par le path '/hello'
- La méthode d'implémentation devra s'appeler sayHello()
- Lors de l'appel de la méthode, le mot world devra s'afficher

Solution

```
git checkout -f step-04-solution
```

Exercice 5

```
git checkout -f step-05
```

Retourner un observable

Exercice 5 : retourner un observable

- Retourner le résultat grâce à un observable
- Pensez à changer le type de la réponse de la méthode

Solution

```
git checkout -f step-05-solution
```

Créer votre API REST

Annexe au cours

Un annexe au cours vous a été fourni afin d’expliquer en détails comment créer une API REST respectant tous les standards du RESTful

Exercice 6

git checkout -f step-06

Retourner les people

Exercice 6 : retourner la liste des people

- Utiliser le CLI pour créer un module people qui contient un controller du même nom
- Ce controller doit avoir un handler GET qui permettra d'appeler une route accessible par le path '/people'
- La méthode d'implémentation devra s'appeler .findAll()
- Retourner le résultat du tableau se trouvant dans src/data/people.ts grâce à un observable
- Quels problèmes peuvent apparaître en procédant ainsi ?
- Comment y remédier ?

Solution

```
git checkout -f step-06-solution
```

Rappel : Définition d'un service

- Une simple classe exportée
- Un décorateur @Injectable()

cats.service.ts

```
import { Injectable } from '@nestjs/common';
import { Cat } from './interfaces/cat.interface';

@Injectable()
export class CatsService {
  private readonly cats: Cat[] = [];

  create(cat: Cat) {
    this.cats.push(cat);
  }

  findAll(): Cat[] {
    return this.cats;
  }
}
```

app.module.ts

```
import { Module } from '@nestjs/common';
import { CatsController } from './cats/cats.controller';
import { CatsService } from './cats/cats.service';

@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class AppModule {}
```

Enregistrement d’un service

Exercice 7

git checkout -f step-07

Utiliser un service

Exercice 7 : utiliser un service

- Utiliser le CLI pour créer un service PeopleService
- Stocker le tableau initial dans une variable de classe
- Créer la méthode .findAll() qui retournera l'observable de la liste de people
- Adapter le handler pour utiliser le service

Solution

```
git checkout -f step-07-solution
```

Exercice 8

git checkout -f step-08

Retourner une personne

Exercice 8 : retourner une personne

- Vous devez créer un handler permettant de retourner une personne
  - /people/:id
  - La valeur de :id est récupérée grâce au décorateur @Param() => @Param('id')
  - La méthode d'implémentation devra s'appeler .findOne()
- Dans le service, créer une méthode .findOne() permettant de trouver la personne dans le tableau suivant son id
- Si la personne n'est pas trouvée, vous devez renvoyer un observable d'erreur avec throwError()
- Utiliser la classe native de Nest NotFoundException pour afficher le message d'erreur
- Ceux qui ont fini avant, faites la même chose avec une personne random

Solution

```
git checkout -f step-08-solution
```

Exercice 9

git checkout -f step-09

Créer une personne

Exercice 9 : créer une personne

- Vous devez créer un handler permettant de créer une personne
  - /people
  - La valeur du payload est récupérée grâce au décorateur @Body()
  - La méthode d'implémentation devra s'appeler .create()
  - Créer le DTO associé en prenant en compte de la suite de l'énoncé
- Dans le service, créer une méthode .create() permettant d'ajouter la personne dans le tableau
- Si la personne existe déjà dans le tableau avec son nom et son prénom, vous devez renvoyer un observable d'erreur avec throwError()
- Utiliser la classe native de Nest ConflictException pour afficher le message d'erreur
- Sinon, générer un nouvel id, la photo (https://randomuser.me/api/portraits/lego/6.jpg) et la date de naissance pour cette personne et ajoutez la dans le tableau
- Retournez la personne ajoutée

Solution

```
git checkout -f step-09-solution
```

Exercice 10

```
git checkout -f step-10
```

Mettre à jour une personne

Exercice 10 : MAJ d’une personne

- Vous devez créer un handler permettant de mettre à jour une personne
  - /people/:id
  - La valeur du payload est récupérée grâce au décorateur @Body()
  - La valeur de :id est récupérée grâce au décorateur @Param() => @Param(‘id’)
  - La méthode d’implémentation devra s’appeler .update()
  - Créer le DTO associé
- Dans le service, créer une méthode .update() permettant de mettre à jour la personne dans le tableau
- Si la personne existe déjà dans le tableau avec son nom et son prénom, vous devez renvoyer un observable d’erreur avec throwError()
- Utiliser la classe native de Nest ConflictException pour afficher le message d’erreur
- Si la personne n’existe pas dans le tableau avec son id, vous devez renvoyer un observable d’erreur avec throwError()
- Utiliser la classe native de Nest NotFoundException pour afficher le message d’erreur
- Sinon, remplacer toute la personne dans le tableau
- Retournez la personne modifiée

Solution

```
git checkout -f step-10-solution
```

Exercice 11

```
git checkout -f step-11
```

Supprimer une personne

Exercice 11 : Supprimer une personne

- Vous devez créer un handler permettant de supprimer une personne
  - /people/:id
  - La valeur de :id est récupérée grâce au décorateur @Param() => @Param('id')
  - La méthode d'implémentation devra s'appeler .delete()
- Dans le service, créer une méthode .delete() permettant de supprimer la personne dans le tableau
- Si la personne n'existe pas dans le tableau avec son id, vous devez renvoyer un observable d'erreur avec throwError()
- Utiliser la classe native de Nest NotFoundException pour afficher le message d'erreur
- Retournez un status no content 204 sans réponse

Solution

```
git checkout -f step-11-solution
```

Quels problèmes peut on rencontrer ?

- Nous avons créer toutes nos routes mais nous n’avons aucune validation en I/O
- On ne peut pas garantir que les données envoyées à notre API sont correctes
- Et que les données que l’on renvoie le sont également
- Il faut donc mettre des validateurs I/O

Fin de la 4ème journée

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.

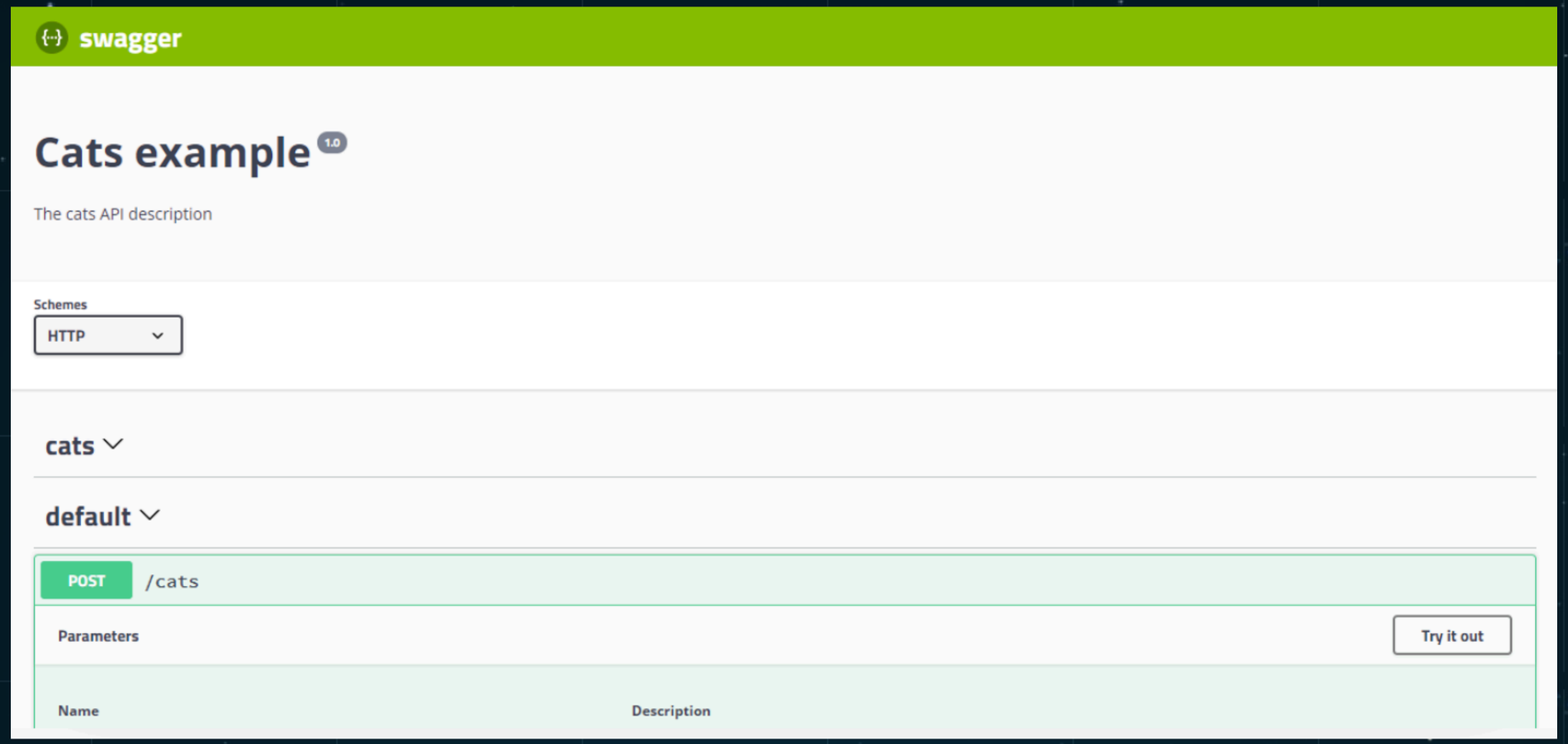
```
import { NestFactory } from '@nestjs/core';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { ApplicationModule } from './app.module';

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



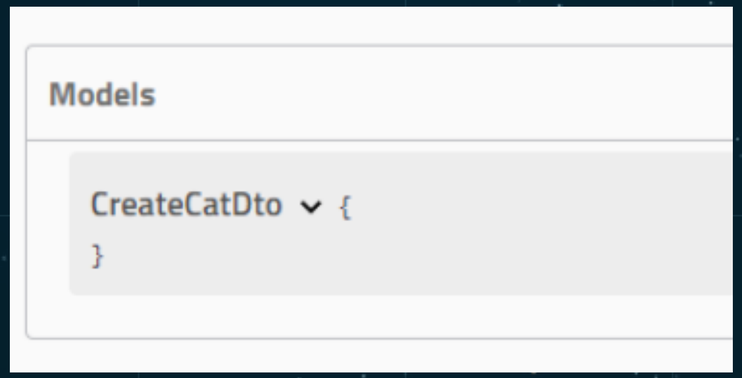
http://localhost:3000/api

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

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

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

  @ApiProperty()
  age: number;

  @ApiProperty()
  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

`@ApiOperation()`	Method
`@ApiResponse()`	Method / Controller
`@ApiProduces()`	Method / Controller
`@ApiConsumes()`	Method / Controller
`@ApiBearerAuth()`	Method / Controller
`@ApiOAuth2()`	Method / Controller
`@ApiBasicAuth()`	Method / Controller
`@ApiSecurity()`	Method / Controller
`@ApiExtraModels()`	Method / Controller

Documentations décorateurs

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

```yaml
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

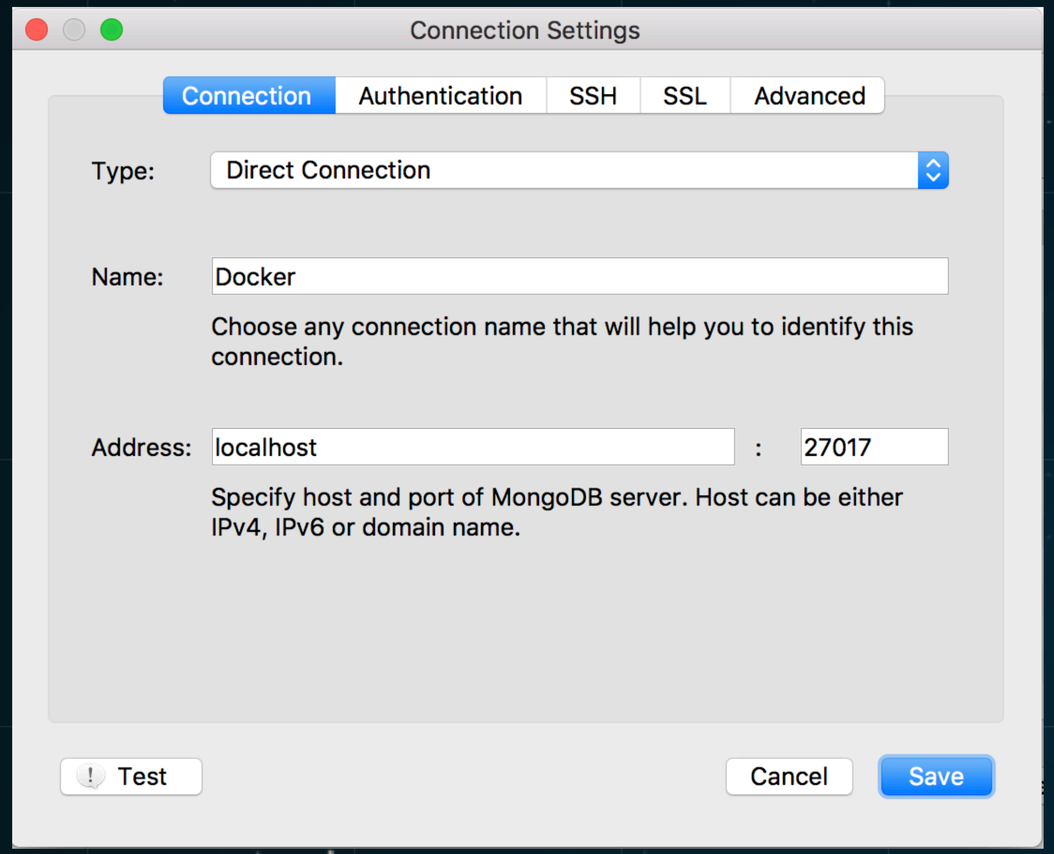
```
{
 "name": "MongoDB",
 "version": {
   "major": 3,
   "minor": 2
 },
 "load": 0.8
 "bytes": BinData(2, "JnkYp0MUwsIiFg=="),
 "startDate": ISODate("2016-02-10T01:00:00.338Z"),
 "started": true,
 "tags": ["storage","db","engine"],
 "nothing": null
}
```

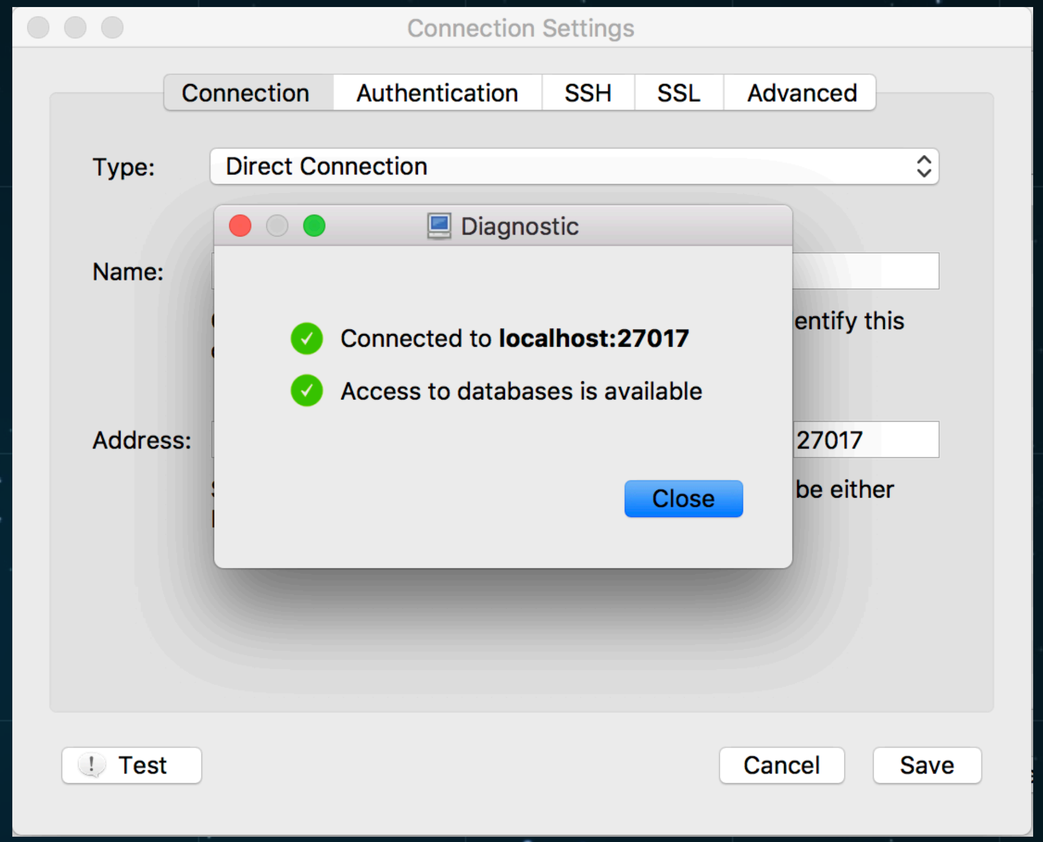
MongoDB : Concept de base

- 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.

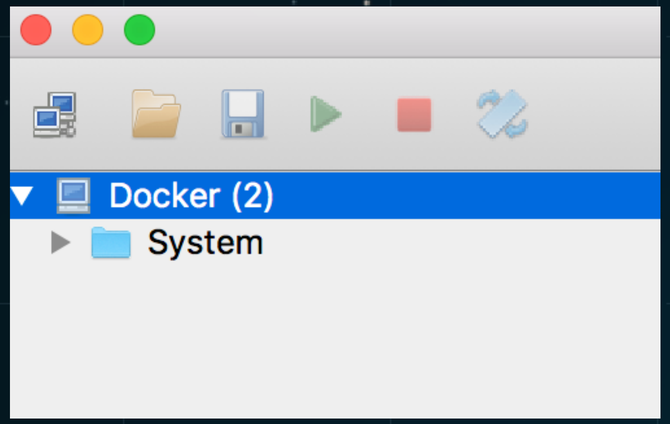
MongoDB : Documentation

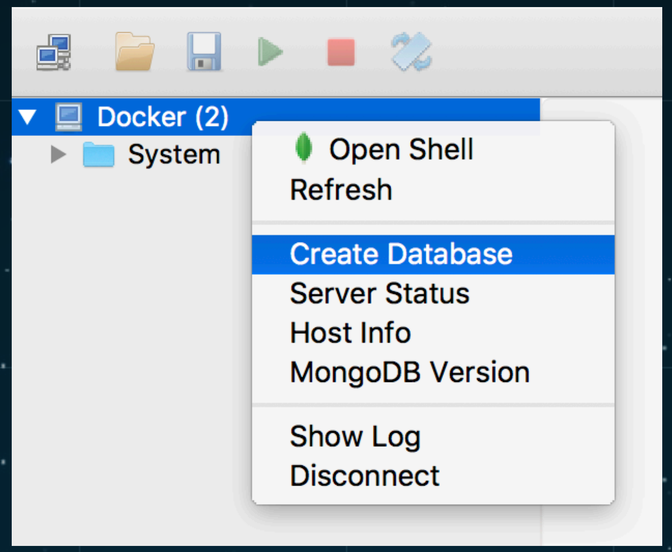
- Utilisation de mongo Shell
- Opération de CRUD
- Indexes
- Replication
- Sharding

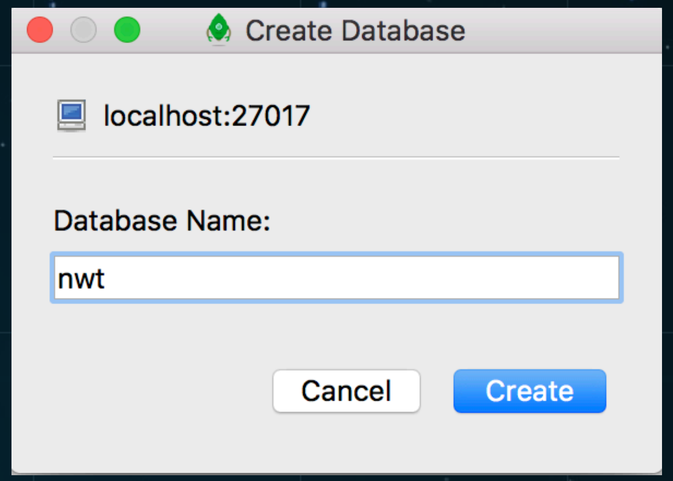




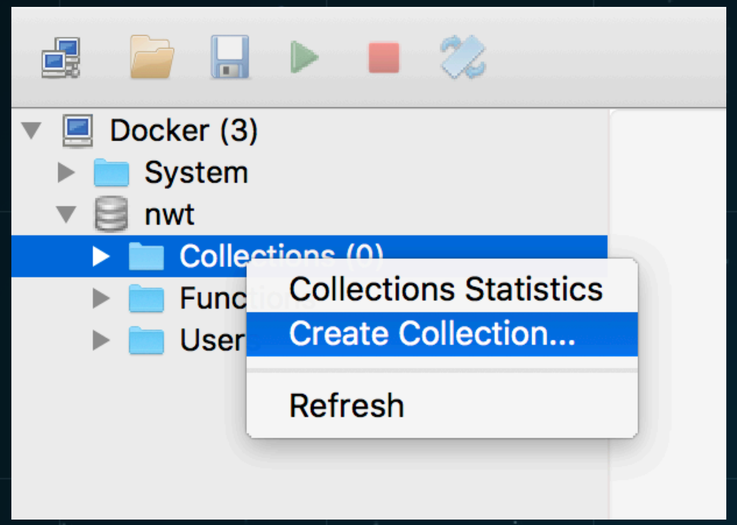
Robo 3T : Connexion

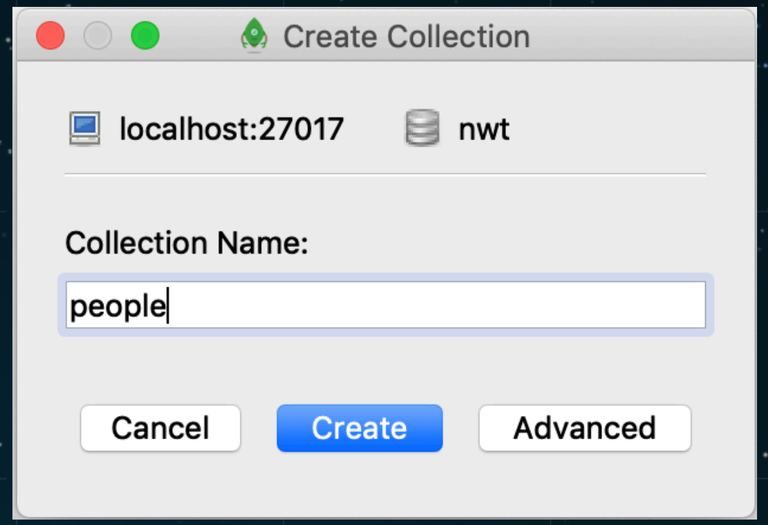


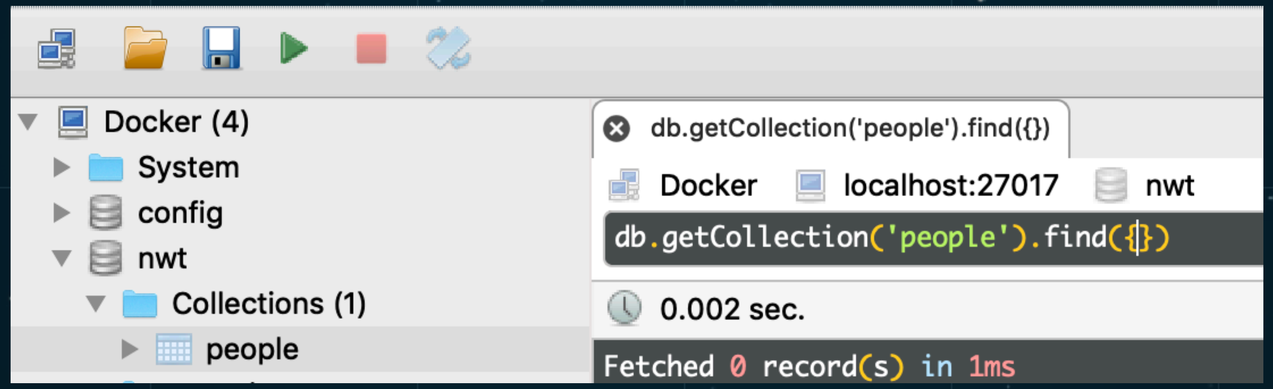




Robo 3T : Créer une DB







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

```typescript
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'adapteur
- 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 indexes 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

A vous de jouer maintenant et place à l’examen

Sujet de l’examen final

- Réaliser une application reprenant tous les concepts vus
- Vous devez avoir une application Front en Angular et une application Back en NestJS
- L’application Front doit avoir au minimum :
  - 2 vues
  - Un formulaire de création et modification
  - La possibilité de supprimer les éléments affichés
- L’application Back doit avoir :
  - Un CRUD complet
  - Une DB mongo
  - Les validations I/O

Sujet de l’examen final

- L’application doit être fonctionnelle et un minimum « belle »
- Inventer un concept d’application originale qui vous distinguera des autres
- Respecter tout ce qu’on a vu au niveau implémentation des concepts

- Effectuer une mini présentation pour expliquer votre application, à quoi elle sert, comment elle fonctionne, quelles sont les fonctionnalités, les problèmes rencontrés lors de la création, …
- Vous effectuerez un live démo de votre application devant tous les autres
- Vous me fournirez la présentation et le code source avec toutes les indications pour que je puisse l’exécuter sur ma machine (scripts pour les données initiales de la DB, …)

Bonne chance et merci pour votre implication