

# 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  
=> 1.22.15
```

```
$ node -v  
=> v16.10.0
```

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://www.github.com/akanass/nwt-school-back-nestjs>

# Déroulement de la formation

```
$ git clone https://www.github.com/akanass/nwt-school-back-nestjs
```

# 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 Big Sur
NodeJS Version  : v16.10.0
YARN Version    : 1.22.15

[Nest CLI]
Nest CLI Version : 8.1.2

[Nest Platform Information]
platform-fastify version : 8.0.10
schematics version       : 8.0.3
mongoose version         : 9.0.0
swagger version          : 5.1.0
testing version          : 8.0.10
common version           : 8.0.10
core version             : 8.0.10
cli version              : 8.1.2
```

# 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

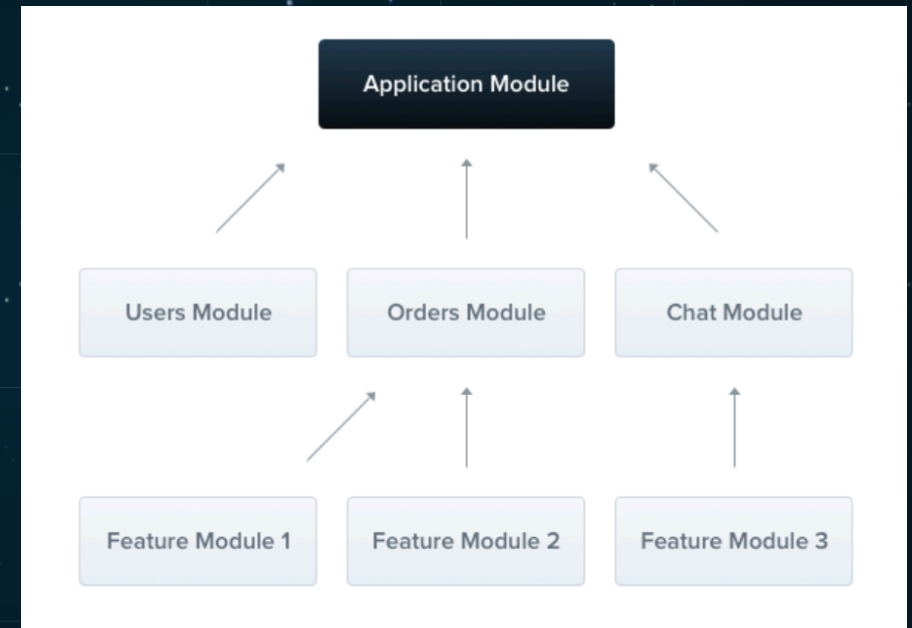
# Framework

- **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 { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create<NestFastifyApplication>(
    AppModule,
    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';

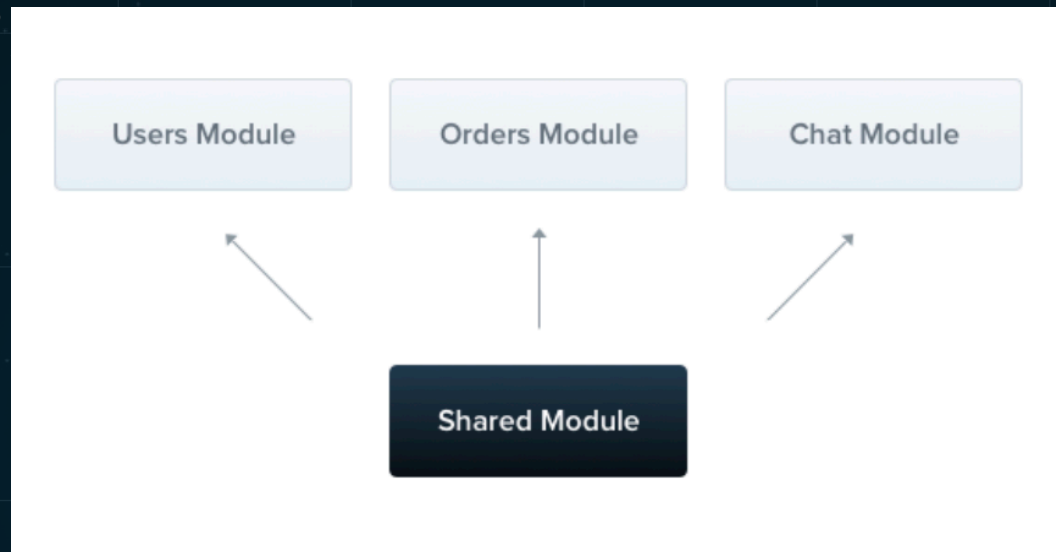
@Module({
  controllers: [CatsController],
  providers: [CatsService],
})
export class CatsModule {}
```

## 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.

```
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.
- Corrigions 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

Si vous avez apprécié la formation, envoyez un Tweet !

#NewWebTechnologies #nestjs @\_akanass\_ @TaDaweb @nestframework