nestJs

Table des matières

I - NestJs (présentation, installation)

1. Présentation de NestJs

NestJs est un **framework de haut niveau** (en **typescript**) permettant de **construire des applications "backends" modulaires** (avec partie "API REST" et partie "accès aux données") s'appuyant sur **nodeJs** .

NB : NestJs s'est fortement inspiré de la structure du framework frontEnd "angular" .
On peut évidemment utiliser n'importe quel framework du coté front ("vue" , "react" , "angular" , …") pour invoquer des api REST développées avec nestJs .

NestJS a été créé par **Kamil Myśliwiec** et la première version date de **décembre 2017**.

En 2024, la version 10 de NestJS nécessite au minimum la version 16 de NodeJs .

2. Mise en place d'un projet nestJS

2.1. @nestjs/cli

@nestjs/cli correspond à un **assistant** "nestjs" **en ligne de commande** (et donc utilisable sur de multiples plateformes : windows , linux, mac , …)

Installation de @nestjs/cli :

```
npm install -g @nestjs/cli
```

Création d'un début d'une nouvelle application avec @nestjs/cli :

```
nest new my-nestjs-app --strict
```

Eventuelle installation de compléments au sein de l'application :

cd my-nestjs-app

npm install --save @nestjs/swagger

npm install --save @nestjs/mongoose mongoose

3. Structure d'une application nestJS

Structure initiale construite via nest new :

my-nestjs-app	my-nestjs-app/src	my-nestjs-app/test
.git node_modules src test .eslintrc.js .gitignore .prettierrc nest-cli.json package.json package-lock.json README.md tsconfig.build.json tsconfig.json	app.controller.spec.ts app.controller.ts app.module.ts app.service.ts main.ts	app.e2e-spec.ts jest-e2e.json

principales dépendances (au sein de package.json) :

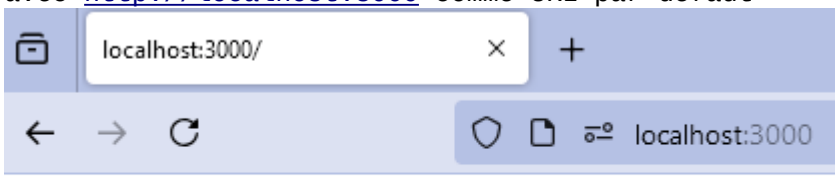
```
"dependencies": {
  "@nestjs/common": "^10.0.0",
  "@nestjs/core": "^10.0.0",
  "@nestjs/platform-express": "^10.0.0",
  "reflect-metadata": "^0.2.0",
  "rxjs": "^7.8.1"
 },
 "devDependencies": {
  "@nestjs/cli": "^10.0.0",
  "@types/express": "^5.0.0",
  "@types/jest": "^29.5.2",
  "@types/node": "^20.3.1",
  "eslint": "^8.0.0",
  "jest": "^29.5.0",
  "prettier": "^3.0.0",
  "typescript": "^5.1.3",
  ...
 },
```

NestJS a fait le choix de la technologie **jest** pour les tests (pas de jasmine , ni de mocha+chai) .

Démarrage de l'application nestJS (en phase de dev) :

npm run start:dev

avec http://localhost:3000 comme URL par défaut



Hello World!

Contenu initial des principaux fichiers :

main.ts	import { NestFactory } from '@nestjs/core';
	import { AppModule } from './app.module';
	
	async function bootstrap() {
	  const app = await NestFactory.create(AppModule);
	  await **app.listen(process.env.PORT ?? 3000);**
	}
	bootstrap();

app.module.ts	import { Module } from '@nestjs/common';
	import { AppController } from './app.controller';
	import { AppService } from './app.service';
	
	@Module({
	  imports: [],
	  controllers: [AppController],
	  providers: [AppService],
	})
	export class AppModule {}

app.service.ts	import { Injectable } from '@nestjs/common';
	
	@Injectable()
	export class **AppService** {
	  getApiDescription(): string { return 'xyz-api : ….' ; }
	}

app.controller.ts	import { Controller, Get } from '@nestjs/common';

```
import { AppService } from './app.service';

@Controller()
export class AppController {
  constructor(private readonly appService: AppService) {}

  @Get()
  getApiDescription(): string {    return this.appService.getApiDescription();  }
}
```

NB:

- Un **service** est un bloc de traitement invisible qui sert à gérer la logique métier (règles de gestion, transactions, …) et qui encapsule souvent un accès à une base de données.
- Un **controller** correspond à un point d'entrée pour le traitement des requêtes HTTP (souvent API REST)
- Un **module** est un regroupement de composants (service, controller , ...)

Les composants d'une application nestJs sont reliés entre eux via une injection de dépendance par constructeur

Principales commandes (nest …)

commandes	utilités
nest new appName --strict	créer nouvelle application
nest g service name	Générer un nouveau service NameService (data access)
nest g controller name	Générer un nouveau contrôleur NameController (api rest)

Organisation classique des répertoires

xxx/

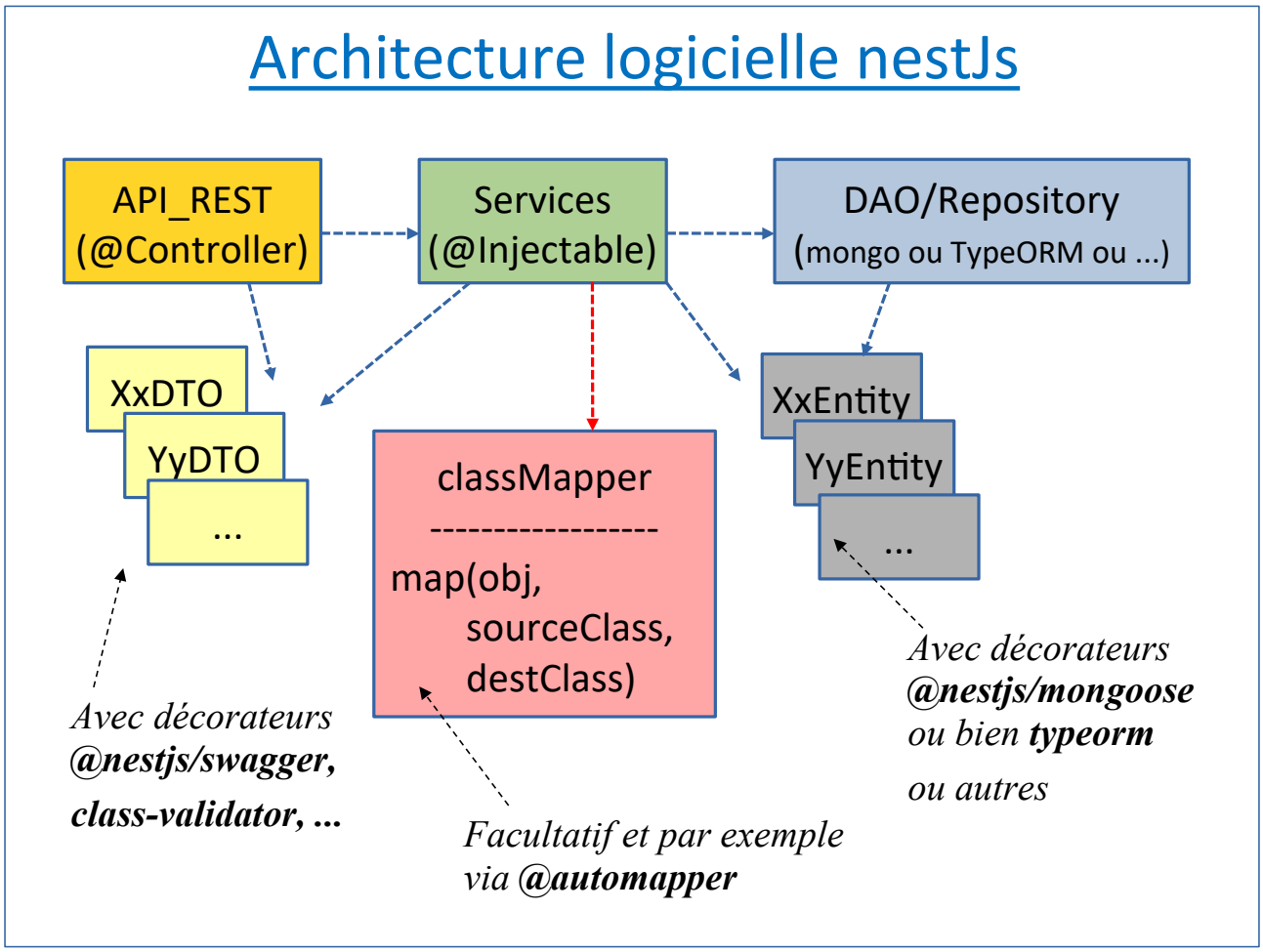
entities/xxx.schema.ts *//modèle mongoose ou bien entité typeorm*
xxx.entity.ts *(utilisé par code privé du service)*

dto/xxx.dto.ts *//dto(s) utilisés par le contrôleur*

xxx.controller.ts *//partie Web-service REST*
xxx.service.ts *//service interne encapsulant un accès aux données*

yyy/…

4. Architecture logicielle nestJs



Les classes de DTO (Data Transfer Object) comportent généralement :

- des décorateurs de @netjs/swagger (ex : @ApiProperty({default:'…' , required : true or false})
- des décorateurs de class-validator (ex : @IsNotEmpty() , IsEmail(), …) pour valider les données entrantes

5. Api REST et documentation swagger

```
npm install -s @nestjs/swagger
```

main.ts

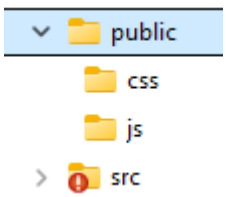
```
import { NestFactory } from '@nestjs/core';
import { SwaggerModule, DocumentBuilder } from '@nestjs/swagger';
import { AppModule } from './app.module';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.setGlobalPrefix('xyz-api');

  const swaggerConfig = new DocumentBuilder()
    .setTitle('news api')
    .setDescription('Simple news rest api')
    .setVersion('1.0')
    .addTag('news')
    .build();
  const document = SwaggerModule.createDocument(app, swaggerConfig);
  SwaggerModule.setup('xyz-api/api', app, document); //http://localhost:3000/xyz-api/api


  await app.listen(process.env.PORT ?? 3000);
  //http://localhost:3000/xyz-api avec AppController et AppService
  //http://localhost:3000 ou http://localhost:3000/index.html avec ServeStaticModule
}
bootstrap();
```

La technologie "**swagger/openApi**" constitue un ***complément incontournable*** pour **documenter** *(et éventuellement tester)* une **api rest**.

Placer le nom de l'api (ex: *xyz-api*) en début d'URL est plutôt une bonne pratique et cela se paramètre via **app.setGlobalPrefix('xyz-api');**

6. Serving static files

```
npm install -s @nestjs/serve-static
```



Le nouveau répertoire **public** (à coté de src) pourra comporter **index.html** et d'éventuels compléments css et js (pour former une page de présentation ou bien un mini frontend de test)

app.module.ts

```
import { Module } from '@nestjs/common';
import { ServeStaticModule } from '@nestjs/serve-static';
import { join } from 'path';
...

@Module({
  imports: [
    ServeStaticModule.forRoot({
    rootPath: join(__dirname, '..', 'public') ,
    exclude: ['/xyz-api/(.*)'],
  }),
    ...],...
})
export class AppModule {}
```

Exemple de page **public/index.html** :

```
<html>
    <head>
      <title>xyz-index</title>
      <link rel="stylesheet" href="css/styles.css" />
    </head>
    <body>
      <h1>xyz api index</h1>
      <a href="xyz-api">xyz api description (from AppController & AppService)</a><br/>
      <a href="xyz-api/api" target="_blank" >xyz api (swagger) description</a><br/>
      <hr/>
      <a href="xyz-api/xyzs">all xyz</a><br/>
      <a href="xyz-api/xyzs/1">xyz 1</a><br/>
      <hr/>
      <a href="xyzAjax.html">xyzAjax</a><br/>
    </body>
</html>
```

Avec cette configuration (assez classique) :

- l'url menant aux fruit de AppController et AppService est **http://localhost:3000/xyz-api**
- l'url menant à la page statique public/index.html est **http://localhost:3000**
  ou bien **http://localhost:3000/index.html**

7. Code généré par nest resource

NB : La commande **nest resource** ***xyz*** permet de générer la structure type de la gestion d'une resource via nestJs.

Dans le cas fréquent d'une **api rest** avec *entités de données en base* (relationnelle ou mongoDB) , le code globalement généré par **nest resource** ***xyz*** est le suivant :

src/app.module.ts *(modif/ajout)*	@Module({ **imports:** [..., **XyzModule**],…}) export class AppModule {}
src/xyz/**xyz.module.ts**	**@Module({** **controllers: [XyzController],** **providers: [XyzService],** **})** export class **XyzModule** {}
src/xyz/**entities/xyz.entity.ts**	export class Xyz {}
src/xyz/**dto**/create-xyz.dto.ts	export class CreateXyzDto {}
src/xyz/**dto**/update-xyz.dto.ts	import { PartialType } from '@nestjs/swagger'; import { CreateXyzDto } from './create-xyz.dto'; export class UpdateXyzDto extends PartialType(CreateXyzDto) {}

src/xyz/**xyz.service.ts**	...
	@Injectable()
	export class ***XyzService*** {
	create(createXyzDto: CreateXyzDto) {
	return 'This action adds a new xyz';
	}
	findAll() { return `This action returns all xyz`;
	}
	findOne(id: number) {
	return `This action returns a #${id} xyz`;
	}
	update(id: number, updateXyzDto: UpdateXyzDto) {
	return `This action updates a #${id} xyz`;
	}
	remove(id: number) {
	return `This action removes a #${id} xyz`;
	}

	}

src/xyz/**xyz.controller.ts**	...
	@Controller('xyz')
	export class ***XyzController*** {
	constructor(private readonly ***xyzService: XyzService***) {}
	
	@Post()
	create(@Body() createXyzDto: CreateXyzDto) {
	return this.xyzService.create(createXyzDto);
	}
	
	@Get()
	findAll() {
	return this.xyzService.findAll();
	}
	
	@Get(':id')
	findOne(@Param('id') id: string) {
	return this.xyzService.findOne(+id);
	}
	
	@Patch(':id')
	update(@Param('id') id: string, @Body() updateXyzDto: UpdateXyzDto) {
	return this.xyzService.update(+id, updateXyzDto);
	}
	
	@Delete(':id')
	remove(@Param('id') id: string) {
	return this.xyzService.remove(+id);
	}
	}

src/xyz/xyz….**spec.ts**	Début de tests unitaires

évidemment ce code minimaliste reste à beaucoup amélioré/peaufiné mais il a le mérite de montrer une structuration assez classique du code avec nestJs .

Variations possibles :

- héritages complémentaires , ...

II - Services (nestJs)

1. Service nestJS (généralités)

nest g service xyz

→ génère xyz/xyz.service.ts et xyz/xyz.service.spec.ts
et ajoute XyzService dans partie providers : [] de app.module.ts ou xyz.module.ts

2. Service agnostique via DTO

2.1. Approche via DTO et éventuel @automapper:

Le service manipule des DTO (Data Transfert Object) dans sa partie publique (en paramètres et en valeur de retour). ces DTO seront (si besoin) transformés en interne en "document mongo" ou bien "entité typeORM" .

@automapper est une bonne technologie sur laquelle on peut éventuellement s'appuyer pour convertir si besoin des DTO en entités persistantes ou vice versa .

Utilisation de @automapper :

```
npm install --save --legacy-peer-deps @automapper/core @automapper/classes
@automapper/nestjs @automapper/types
```

NB : l'option --legacy-peer-deps n'est à utiliser qu'en cas de conflit entre versions de certaines dépendances indirectes

Dans **app.module.ts**

```
import { AutomapperModule } from '@automapper/nestjs';
import { classes } from '@automapper/classes';
...
@Module({
  imports: [
        AutomapperModule.forRoot( {   strategyInitializer: classes() } ),
      … ] , ...})
export class AppModule {}
```

Dans xyz/mapper/**xyz.mapper.profile.ts**

```
import { Mapper, createMap, forMember, mapFrom } from "@automapper/core";
import { AutomapperProfile, InjectMapper } from "@automapper/nestjs";
import { Injectable } from "@nestjs/common";
import { XxxDto, YyyDto } from "../dto/xyz.dto";
```

```
import { XyzEntity } from "../entities/xyz.entity";

@Injectable()
export class XyzMapperProfile extends AutomapperProfile {
  constructor(@InjectMapper() mapper: Mapper) {
    super(mapper);
  }

  override get profile() {
    return (mapper: Mapper) => {
      createMap(mapper, XxxDto,XyzEntity);
      createMap(mapper, YyyDto,XyzEntity);
      createMap(mapper, XyzEntity,XxxDto,
          forMember( d=> d.id, mapFrom(s => s.id))
        );
    };
  }
}
```

NB : On définit ici tous les besoins en conversions (entièrement automatique ou pas).
Si besoin d'une conversion spécifique , s=source , d=destination dans fonction fléchée/lambda .

Dans **xyz.module.ts**

```
import { NewsMapperProfile } from "./mapper/news.mapper.profile";
...
@Module({
  imports: [ …] ,
  providers: [  … ,    XyzMapperProfile ] , ...})
export class AppModule {}
```

Au sein des classes de DTO et d'entités :

```
import { AutoMap } from "@automapper/classes";
...
export class XxxDto  {
  @ApiProperty({default:'myFirstname'})
```

```
import { AutoMap } from "@automapper/classes";

@Entity("xyz")
export class XyzEntity {
```

```
  @IsNotEmpty()
  @AutoMap()
  public firstname: string;

  @ApiProperty({default:'myLastname'})
  @IsNotEmpty()
  @AutoMap()
  public lastname: string;
…
}
```

```
  @PrimaryGeneratedColumn()
  @AutoMap()
  id?: number;

  @Column()
  @AutoMap()
  firstname: string;

  @Column()
  @AutoMap()
  lastname: string;
}
```

Utilisation de @automapper au sein d'un service :

```
import { Mapper } from '@automapper/core';
import { InjectMapper } from '@automapper/nestjs';
...
@Injectable()
export class XyzService {
   constructor(
      ...,
      @InjectMapper() private readonly classMapper: Mapper,
   ){}

   async findAll():Promise<XxxDto[]>{
     const entityArray  = await this….;
     return this.classMapper.mapArrayAsync(entityArray,XyzEntity,XxxDto);
   }

   async findOne(id: string):Promise<XxxDto>{
    try{
      const entity = await this….. ;
      return this.classMapper.mapAsync(entity,XyzEntity,XxxDto);
    }catch(ex){
```

```
      throw new Error(`NOT_FOUND: xxx not found with id=${id}`);
    }
   }

  async create(xDto: XxxDto): Promise<XxxDto> {
      let xxxToCreate = this.classMapper.map(xDto,XxxDto,XyzEntity);
      const savedEntity = await this.xxxRepository.save(xxxToCreate);
      ...
    }
...
}
```

3. Service basé sur mongoose

```
npm install -s @nestjs/mongoose mongoose
```

app.module.ts

```
...
import { MongooseModule } from '@nestjs/mongoose';
import { NewsModule } from './news/news.module';

@Module({
  imports: [...,
    MongooseModule.forRoot('mongodb://localhost:27017/xyz'),
     NewsModule
      ],...
})
export class AppModule {}
```

news/news.module.ts

```
...
import { MongooseModule } from '@nestjs/mongoose';
import { NewsSchema } from './news/news.schema.doc';

@Module({
  imports: [...,
          MongooseModule.forFeature([ { name: "News", schema : NewsSchema}])
      ],
  controllers: [NewsController],
  providers: [NewsService , NewsMapperProfile],
})
export class AppModule {}
```

3.1. Schéma mongoose avec ou sans décorateur

On peut éventuellement définir la structure d'un schéma mongoose **sans décorateur** avec un code de de genre :

news.schema.ts *(v1 sans décorateur)*

```
import * as mongoose from 'mongoose';

export const NewsSchema = new mongoose.Schema({
  //with default mongo _id: { type : ObjectId , alias : "id" }
  title: String,
  text: String,
  timestamp: String,
});
```

```
import { Document } from 'mongoose';
import { News } from './news.itf';

export interface NewsDoc  extends News,  Document  {
  id?:string
}
```

Dans un style de code typescript, on pourra préférer une approche alternative basée sur des décorateurs :

news/entities/news.entity.ts *(v2 avec décorateur)*

```
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { HydratedDocument } from 'mongoose'
import { AutoMap } from '@automapper/classes';

@Schema()
export class NewsEntity{

  //default @Prop  mongo _id: { type : ObjectId , alias : "id" }
  id:string;

  @Prop()  @AutoMap()
  title: string;

  @Prop()  @AutoMap()
  text: string;

  @Prop()  @AutoMap()
  timestamp?: string;
}

export type NewsDoc = HydratedDocument<NewsEntity>;

export const NewsSchema = SchemaFactory.createForClass(NewsEntity);
//with default mongo _id: { type : ObjectId , alias : "id" }
```

Quelques possibilités à ce niveau :

```
@Prop([String])
comments : string[];

@Prop({ required: true })
...

if (subobject , subSchema):
@Prop({ type: mongoose.Schema.Types.ObjectId, ref: 'SubType...' })

if array of subobjects:
@Prop({ type: [mongoose.Schema.Types.ObjectId, ref: 'SubType...' }]})
```

3.2. Service utilisant un modèle mongoose

```
import { Injectable } from '@nestjs/common';
...
import { Model } from 'mongoose';
import { InjectModel } from '@nestjs/mongoose';
import { NewsDoc, NewsEntity } from './entities/news.entity';

@Injectable()
export class NewsService {

 /*NB: @InjectModel('nameOfModel') doit correspondre au nom du modèle enregistré dans
  MongooseModule.forFeature([ { name: "nameOfModel", schema : SchemaQuiVaBien}])
 au sein de app.module.ts ou bien xxx.module.ts (en plus de MongooseModule.forRoot)
 */

  constructor(
     @InjectModel('News') private readonly newsModel: Model<NewsDoc>,
     @InjectMapper() private readonly classMapper: Mapper,
  ){}

 async findAll():Promise<NewsL1Dto[]>{
    const newsDocArray  : NewsDoc[] = await this.newsModel.find().exec();
    return this.classMapper.mapArrayAsync(newsDocArray,NewsEntity,NewsL1Dto);
  }

  async findOne(id: string):Promise<NewsL1Dto>{
   try{
     const newsDoc = await this.newsModel.findOne({ _id: id}).exec();
     return this.classMapper.mapAsync(newsDoc,NewsEntity,NewsL1Dto);
   }catch(ex){
     throw new Error(`NOT_FOUND: news not found with id=${id}`);
   }
  }

  async create(news: NewsL0Dto): Promise<NewsL1Dto> {
     const newsToCreate = news; // compatible , more simple
     //const newsToCreate = this.classMapper.map(news,NewsL0Dto,NewsEntity); //ok
     const persistentNewsEntity = new this.newsModel(newsToCreate);
     const savedNewsAsNewsEntity = <NewsEntity> <any> await persistentNewsEntity.save();
     return this.classMapper.mapAsync(savedNewsAsNewsEntity,NewsEntity,NewsL1Dto);
  }

   async remove(id: string): Promise<any>{
     const doesNewsExit = await this.newsModel.exists({ _id: id });
     if(!doesNewsExit)
       throw new Error(`NOT_FOUND: not existing news to delete with id==${id}`);

     try{
       return await this.newsModel.findByIdAndDelete(id); //deletedNews
```

```
      }catch(ex){
       throw new Error(`Exception in NewsService.remove() with id==${id}`);
      }
    }

    async update(id: string, newsDto: NewsL1Dto): Promise<NewsL1Dto> {
      const newsToUpdate = newsDto; // compatible , more simple
      //const newsToUpdate = this.classMapper.map(newsDto,NewsL1Dto,NewsEntity); //ok
      const updatedNewsAsNewsEntity = <NewsEntity> <any> await
            this.newsModel.findByIdAndUpdate( id, newsToUpdate, { newsToUpdate: true });
      if(updatedNewsAsNewsEntity==undefined)
        throw new Error(`NOT_FOUND: not existing news to update with id=${id}`);
      return this.classMapper.mapAsync(updatedNewsAsNewsEntity,NewsEntity,NewsL1Dto);
    }

}
```

Autres exemples :

```
async findByUsername(username: string){
  const userDoc = await this.userModel.findOne({ username: username}).exec();
…   }
```

4. Service basé sur TypeORM et mysql ou postgres

4.1. Configuration de TypeORM au sein de nestJS

```
npm install -s @nestjs/typeorm typeorm mysql2
```

app.module.ts (avec dépendance de type ***"mysql2": "^3.11.3"*** dans package.json)

```
...
import { TypeOrmModule } from '@nestjs/typeorm';
import { AccountEntity } from './account/entities/account.entity';
import { AccountModule } from './account/account.module';

@Module({
 imports: [...,
   TypeOrmModule.forRoot({
     type: 'mysql',
     host: 'localhost',
     port: 3306,
     username: 'root',
     password: 'root',
     database: 'nestJsBankDb',
     entities: [CustomerEntity,AccountEntity ,OperationEntity]
   }),
    CustomerModule , AccountModule
     ],...
})
export class AppModule {}
```

ou bien (avec dépendance de type ***"pg" : "^8.13.0"*** dans package.json)

```
TypeOrmModule.forRoot({
     type: 'postgres',
     host: 'localhost',
     port: 5432,
     username: 'postgres',
     password: 'root',
     database: 'nestJsBankDb',
     entities: [CustomerEntity,AccountEntity ,OperationEntity]
   }),
```

customer/customer.module.ts

```
...
import { TypeOrmModule } from "@nestjs/typeorm";

@Module({
 imports: [...,
         TypeOrmModule.forFeature([CustomerEntity])
     ],
 controllers: [CustomerController],
```

```
  providers: [CustomerService , CustomerMapperProfile],
})
export class AppModule {}
```

4.2. Entité simple basée sur TypeORM (et mysql ou postgresql)

customer/entities/**customer.entity.ts**

```
import { AutoMap } from '@automapper/classes';
import { Entity, Column, PrimaryGeneratedColumn } from 'typeorm';

@Entity("customer")
export class CustomerEntity {
  @PrimaryGeneratedColumn()
  @AutoMap()
  id?: number;

  @Column()
  @AutoMap()
  firstname: string;

  @Column()
  @AutoMap()
  lastname: string;

  @Column()
  @AutoMap()
  email?: string;
}
```

4.3. Service simple basé sur TypeORM

customer.service.ts

```
import { Injectable } from '@nestjs/common';
import { InjectRepository } from '@nestjs/typeorm';
import { Repository } from 'typeorm';
import { Mapper } from '@automapper/core';
import { InjectMapper } from '@automapper/nestjs';
import { CustomerEntity } from './entities/customer.entity';
import { CustomerL0Dto, CustomerL1Dto } from './dto/customer.dto';

@Injectable()
export class CustomerService {

  constructor(
     @InjectRepository(CustomerEntity) private customerRepository: Repository<CustomerEntity>,
     @InjectMapper() private readonly classMapper: Mapper,
    ) {}
```

```
  async findAll():Promise<CustomerL1Dto[]>{
    return this.classMapper.mapArrayAsync(await this.customerRepository.find(),
                                CustomerEntity,CustomerL1Dto);
  }


  async findOne(id: number):Promise<CustomerL1Dto>{
    try{
      const customerEntity = await this.customerRepository.findOneBy({ id });
      if(customerEntity==null)
         throw new Error(`NOT_FOUND: customer not found with id=${id}`);
      else
         return this.classMapper.mapAsync(customerEntity,CustomerEntity,CustomerL1Dto);
    }catch(ex){
      const subErrorPrefix = (ex instanceof Error)?`${ex.message}:`:"";
      throw new Error(`${subErrorPrefix}exception in CustomerService.findOne() with id=${id}`);
    }
   }

  async create(customerDto: CustomerL0Dto): Promise<CustomerL1Dto> {
    const customerToCreate = customerDto; // compatible in simple case , more simple/efficient
    //const customerToCreate = this.classMapper.map(customerDto,CustomerL0Dto,CustomerEntity);
    const savedCustomerAsEntity = await this.customerRepository.save(customerToCreate);
    return this.classMapper.mapAsync(savedCustomerAsEntity,CustomerEntity,CustomerL1Dto);
  }

  async _throwNotFoundErrorInNotExists(id:number,messagePart:string){
    const doesExit = await this.customerRepository.exists({ where: { id: id } });
      if(!doesExit)
         throw new Error(`NOT_FOUND: ${messagePart} with id=${id}`);
  }

  async remove(id: number): Promise<boolean>{
    await this._throwNotFoundErrorInNotExists(id,"no existing customer to delete");
    try{
      const deletedResult = await this.customerRepository.delete(id); //deletedResult
      //console.log("deletedResult=" + JSON.stringify(deletedResult));
      return (deletedResult.affected===1);
    }catch(ex){
      const subErrorPrefix = (ex instanceof Error)?`${ex.message}:`:"";
     throw new Error(`${subErrorPrefix}Exception in CustomerService.remove() with id==${id}`);
    }
  }



async update(id:number , dto : CustomerL1Dto): Promise<boolean> {
  await this._throwNotFoundErrorInNotExists(id,"no existing customer to update");
  try{
      const customerToUpdate = dto; // compatible in simple case , more simple/efficient
      //const customerToUpdate = this.classMapper.map(dto,CustomerL1Dto,CustomerEntity); //ok
      const updateResult =  await this.customerRepository.update(id, customerToUpdate);
      console.log("updateResult="+JSON.stringify(updateResult)); //no updatedEntity , just affected
      return (updateResult.affected===1);
  }catch(ex){
    const subErrorPrefix = (ex instanceof Error)?`${ex.message}:`:"";
```

```
      throw new Error(`${subErrorPrefix}Exception in CustomerService.update() with id==${id}`);
    }
  }
}
```

Exemple de test du service :

customer.service.spect.ts

```
import { Test, TestingModule } from '@nestjs/testing';
import { CustomerService } from './customer.service';
import { TypeOrmModule } from '@nestjs/typeorm';
import { CustomerEntity } from './entities/customer.entity';
import { classes } from '@automapper/classes';
import { AutomapperModule } from '@automapper/nestjs';
import { CustomerMapperProfile } from './mapper/customer.mapper.profile';
import { CustomerL0Dto, CustomerL1Dto } from './dto/customer.dto';

//npm run test -t customer.service.spec

describe('CustomerService', () => {
  let service: CustomerService;
  let module: TestingModule;

  beforeEach(async () => {
    module = await Test.createTestingModule({
      providers: [CustomerService , CustomerMapperProfile],
      imports: [
        AutomapperModule.forRoot(  {   strategyInitializer: classes()  } ),
        TypeOrmModule.forRoot({
          type: 'mysql',         host: 'localhost',         port: 3306,
          username: 'root',        password: 'root',         database: 'nestJsBankDb',
          entities: [CustomerEntity]
        }),
        TypeOrmModule.forFeature([CustomerEntity])
      ]
    }).compile();

    service = module.get<CustomerService>(CustomerService);
  });

  it('should be defined', () => {
    expect(service).toBeDefined();
  });

  it('customer crud', async () => {
    //1. create and save a new customer
    let c= new CustomerL0Dto(); c.firstname="prenom1"; c.lastname="nom1";
    const createdCustomer : CustomerL1Dto = await service.create(c);
    const cId=createdCustomer.id??0;
    console.log("cId="+cId);

    //retreive/query it to check insertion
```

```
  let cRelu = await service.findOne(cId);
  console.log("cRelu="+JSON.stringify(cRelu))
  expect(cRelu.id).toBe(cId);
  expect(cRelu.firstname).toBe("prenom1");
  expect(cRelu.lastname).toBe("nom1");

  //2. update some customer values
  if(cRelu){
  cRelu.firstname="prenom_1";
  cRelu.lastname="Nom_1";
  await service.update(cId,cRelu);
  }

  //retreive/query it to check update
  let cRelu2 = await service.findOne(cId);
  console.log("cRelu2 after update="+JSON.stringify(cRelu2))
  expect(cRelu2.id).toBe(cId);
  expect(cRelu2.firstname).toBe("prenom_1");
  expect(cRelu2.lastname).toBe("Nom_1");

  //3. delete customer
  await service.remove(cId);

  //try to retreive it to check delete
  try{
    let cRelu3 = await service.findOne(cId);
    expect(true).toBe(false);//fail workaround
  }catch(expectedException){
      expect(expectedException).toBeInstanceOf(Error);
      expect((<Error>expectedException).message).toContain('NOT_FOUND');
      console.log("excepted exception if not found:" + expectedException);
  }
 });

 afterEach(async () => {
   module.close();
 });

});
```

résultats :

cId=12345574

cRelu={"firstname":"prenom1","lastname":"nom1","email":"aaa.bbb@xyz.com","id":12345574}

cRelu2 after update={"firstname":"prenom_1","lastname":"Nom_1","email":"aaa.bbb@xyz.com","id":12345574}

excepted exception if not found:Error: NOT_FOUND: customer not found with id=12345574:exception in CustomerService.findOne() with id=12345574

CustomerService

√ should be defined (194 ms)

√ customer crud (103 ms)

Test Suites: 1 passed, 1 total

```
Tests:       2 passed, 2 total
Snapshots:   0 total
Time:        7.471 s, estimated 8 s
```

4.4. Requêtes TypeORM (au sein de nestJs)

Exemples :

```
import { MoreThanOrEqual, Repository } from 'typeorm';
...
async findWithMinimumBalance(minimumBalance:number): Promise<AccountL1Dto[]> {
      const accountEntityArray = await this.accountRepository.find({
        where : {
          balance : MoreThanOrEqual(minimumBalance)
        }
       });
      return this.classMapper.mapArrayAsync(accountEntityArray,AccountEntity,AccountL1Dto);
    }
```

4.5. Mappings TypeORM et utilisations

4.5.a. n-1 (@ManyToOne)

Exemple :

```
import { Entity, Column, PrimaryGeneratedColumn, ManyToOne, JoinColumn} from 'typeorm';
...
@Entity("account")
export class AccountEntity  {
  @PrimaryGeneratedColumn()
  num?: number;

  @Column()
  label: string;

  @Column()
  balance: number;

 @ManyToOne(()=> CustomerEntity , owner => owner.id)
  @JoinColumn({name:'customer_id'}) //foreign key name
```

```
  owner : CustomerEntity;
}
```

Exemple :

```
findByOwnerId(ownerId:number):  {
      return this.accountRepository.find({
       // relations: {owner:true },
         where : {
            owner :{
              id : ownerId
            }
         }
      });
    }
```

NB: Par défaut , on retourne les comptes/accounts qui appartiennent au client/customer de numéro recherché qu'avec leurs données propres (num,label,balance) .

Avec le paramétrage ***relations: {owner:true }*** , chaque compte/account trouvé est retourné avec une liaison avec le client/customer associé : { num : … , label : … , solde : … **, owner : { … }** }

4.5.b. n-n (@ManyToMany)

Exemple :

```
import { Entity, Column, PrimaryGeneratedColumn, ManyToMany, JoinTable } from 'typeorm';
…
...
@Entity("account")
export class AccountEntity  {
  @PrimaryGeneratedColumn()
  num?: number;

  @Column()
  label: string;

  @Column()
  balance: number;

//joinColumn : this_side , inverseJoinColumn : collection side
@ManyToMany(()=> CustomerEntity )
   @JoinTable({ name: 'customer_account' ,
                joinColumn: { name: 'account_num' },
                inverseJoinColumn: {  name: 'customer_id' }
   })
   owners? : CustomerEntity[];
}
```

Exemple :

```
async findByOwnerId(ownerId:number): Promise<AccountL1Dto[]> {
      const accountEntityArray = await this.accountRepository.createQueryBuilder('account')
```

```
        .leftJoin('account.owners', 'owner')
        .where('owner.id = :ownerId', { ownerId: ownerId })
        .getMany();
    return this.classMapper.mapArrayAsync(accountEntityArray,AccountEntity,AccountL1Dto);
  }
```

III - Contrôleur d'api REST (nestjs)

1. Controller nestJS

```
nest g controller xyz
```

→ génère xyz/xyz.controller.ts et xyz/xyz.controller.spec.ts
et ajoute XyzController dans partie controllers : [] de app.module.ts ou xyz.module.ts

URL pour contrôleur "xyz"

```
//controller with name=xyz ---> localhost:3000/xyz ou bien  localhost:3000/my-api/xyz
@Controller('xyz')
export class XyzController {
…
}
```

xyz.module.ts

```
...
@Module({
  imports: [ ...  ],
  controllers: [XyzController],
  providers: [XyzService,XyzMapperProfile],
})
export class XyzModule {}
```

2. contrôleur simple avec DTO

customer.controller.ts

```
import { Body ,Controller,Delete, Get, HttpException, HttpStatus, Param, Post, Put} from
'@nestjs/common';
import { CustomerService } from './customer.service';
import { Message } from 'src/common/message';
import { ApiResponse } from '@nestjs/swagger';
import { CustomerL0Dto, CustomerL1Dto } from './dto/customer.dto';


@Controller('customers')
export class CustomerController {

   constructor(private readonly customerService: CustomerService) {
   }

   @Get()
   @ApiResponse({
      description : "collection of searched customers",
      type: [CustomerL1Dto],
```

```
  })
  async getCustomersByCriteria(): Promise<CustomerL1Dto[]> {
    return this.customerService.findAll();
  }

  @Get(':id')
  async getCustomerById(@Param('id') id:number): Promise<CustomerL1Dto> {
    return this.customerService.findOne(id);
  }

  //{ "firstName" : "prenom_x" , "lastname" : "nom_y" }
  @Post()
  async create(@Body() c: CustomerL0Dto): Promise<CustomerL1Dto> {
    const createadCustomer = await this.customerService.create(c);
    return createadCustomer;
   }

   @Delete(':id')
   //@HttpCode(204) if no return json message
   async remove(@Param('id') id:number) {
     const deletedOk = await this.customerService.remove(id);
     return new Message("customer with id="+id + " is now deleted");//with default 200/OK
   }

   //{"id": "1" ,  "firstName" : "prenom_x" , "lastname" : "nom_y" }
   @Put(':id') //or @Patch(':id')
   //@HttpCode(204) if no return json message
   async update(@Body() customerToUpdate: CustomerL1Dto,
                @Param('id') id:number): Promise<CustomerL1Dto> {
     customerToUpdate.id=id; //must be coherent
     const updatedOk =   await this.customerService.update(id, customerToUpdate);
     let updatedCustomer = await this.customerService.findOne(id);
     return updatedCustomer; //with default 200/ok
   }
}
```

3. Vérifications/Validations des données entrantes

3.1. Validation (via ValidationPipe)

Le framework nestJs est prévu pour intégré swagger et le petit framework de validation "**class-validator**" .

```
npm install -s @nestjs/swagger
npm install -s class-transformer
npm install -s class-validator
```

Il faut commencer par placer des décorateurs de @nestjs/swagger et class-validator au sein des classes de DTO :

```
import { ApiProperty } from "@nestjs/swagger";
import { IsEmail, IsNotEmpty } from "class-validator";
//NB: @ApiProperty() est nécessaire pour une bonne compréhension
//      du schema/DTO par swagger
export class CustomerDto  {
  @ApiProperty({default:'myFirstname'})
  @IsNotEmpty()
  public firstname: string;

  @ApiProperty({default:'myLastname'})
  @IsNotEmpty()
  public lastname: string;

  @ApiProperty({required:false,default:'aaa.bbb@xyz.com'})
  @IsEmail()
  public email?: string;
}
```

Pour demander au framework netstJs d'effectuer une validation des données entrantes , on peut :

- soit localement utiliser **@UsePipes(new ValidationPipe())** près de **@Post, @Put/@Patch** au niveau d'un @Controller
- soit globalement enregistrer le pipe de validation au sein de **main.ts** : **app.useGlobalPipes(new ValidationPipe());**

Exemple de comportement :

POST http://http://localhost:3000/bank-api/customers

entrées	*sorties*

{
"firstname": "jean",
"lastname": "Bon",
"email": "jean.bon@xyz.com"
}
	200/OK
	{
	"firstname": "jean",
	"lastname": "Bon",
	"email": "jean.bon@xyz.com",
	"id": 12345576
	}

{
"firstname": "",
"lastname": "Bon",
"email": "jean.bon@xyz.com"
}
	400/BAD_REQUEST
	{
	"**statusCode**": 400,
	"**message**": "Bad Request Exception"
	} *ou autre selon HttpExceptionFilter*

{
"firstname": "jean",
"lastname": "Bon",
"email": "jean.bonxyz.com"
}
	400/BAD_REQUEST
	{
	"**statusCode**": 400,
	"**error**": "Bad Request",
	"**message**": ["email must be an email"]
	}

3.2. Eventuelle transformation complémentaire :

Au niveau local (via @UsePipes()) ou bien au niveau global (via app.useGlobalPipes() dans main.ts) , new **ValidationPipe()** comporte un **paramètre facultatif { transform : true }**.

Par défaut , une requête Http entrante de type POST, PUT ou PATCH comportant des données JSON au sein du corps (@Body()) de la requête , mène à la suite de traitement suivante :

- conversion JSON → plain javascript object via JSON.parse()
- validation via ValidationPipe et décorations de class-validator avec retour automatique de 400/BAD_REQUEST en cas d'erreur(s)
- exécution de la méthode ad hoc du @Controller au sein de laquelle le paramètre d'entrée (associé à @Body()) est de type "plain object" mais pas une rélle instance de la classe du DTO (ce qui n'est pas forcément gênant tant que la structure reste bien compatible) .

Exemple :

```
@Post()
 @UsePipes(new ValidationPipe()) //ou équivalent global dans main.ts
 async create(@Body() c: CustomerL0Dto): Promise<CustomerL1Dto> {
    console.log( 'In CustomerController.create() typeof c = ' + typeof c);
    if(c instanceof CustomerL0Dto)
        console.log("c is an instance of CustomerL0Dto if ValidationPipe with transform:true");
    else
        console.log("In CustomerController.create() c is a plain object");
    const createadCustomer = await this.customerService.create(c);
    return createadCustomer;
  }
```

In CustomerController.create() typeof c = object

In CustomerController.create() c is a plain object

Pour demander à générer au runtime une véritable instance du DTO , il faut ajouter le paramètre **{ transform:true }** au sein du constructeur de **ValidationPipe()** :

```
@Post()
 @UsePipes(new ValidationPipe({ transform:true } )) //ou équivalent global dans main.ts
 async create(@Body() c: CustomerL0Dto): Promise<CustomerL1Dto> {
    console.log( 'In CustomerController.create() typeof c = ' + typeof c);
    if(c instanceof CustomerL0Dto)
        console.log("c is an instance of CustomerL0Dto if ValidationPipe with transform:true");
    else
        console.log("In CustomerController.create() c is a plain object");
    const createadCustomer = await this.customerService.create(c);
    return createadCustomer;
  }
```

In CustomerController.create() typeof c = object

c is an instance of CustomerL0Dto if ValidationPipe with transform:true

Au sens "typescript" (analysant de potentiels décorateurs) , la classe du DTO peut éventuellement comporter des décorateurs de l'api **class-transformer** telles que @Expose() ou @Exclude()

Exemple :

```
…
import { Exclude, Expose } from "class-transformer";

export class CustomerL0Dto  {
  ….
 @Expose({name:"prenom"}) //pour .prenom à la place de .firstname
 public firstname: string;
 ...

 @Exclude()  //pour ignorer (faire disparaître) la partie  .email
 public email?: string;

 @Expose({name:"mot_de_passe"})  //pour .mot_de_passe à la place de .password
 public password? : string = "pwd007";
}
```

Attention, attention :

- toute seule , l'api class-transformer semble intéressante et fonctionne bien
- couplée à d'autres api , l'api class-transformer est carrément incontrôlable (on s'y perd entre structure typescript , structure décorée , structure au runtime , …) .

→ Pour que le code d'une application nodeJs reste compréhensible et maîtrisable , il faut généralement se forcer à n'utiliser l'api class-transformer qu'au niveau des DTOs et des contrôleurs mais surtout pas au niveau des services (car cette api cohabite très mal avec mongoose ou typeorm par exemple) .

NB : Dans le sens inverse des entrées des modes POST/PUT/PATCH , il est éventuellement possible d'ajuster la sérialisation JSON via des décorateurs de l'API class-transformer et pour cela il faut placer l'annotation **@UseInterceptors(ClassSerializerInterceptor)** par exemple près d'un @Get.

Exemple:

```
import { ClassSerializerInterceptor,..., UseInterceptors } from '@nestjs/common';
...
@Get(':id')
@UseInterceptors(ClassSerializerInterceptor) //to interpret @Exlude , @Expose during json serialization
   async getCustomerById(@Param('id') id:number): Promise<CustomerL1Dto> {
      return this.customerService.findOne(id);
   }
```

4. Gestion explicite des statuts Http

La décoration **@HttpCode()** permet de spécifier un code de retour différent de 200/OK .

On peut par exemple choisir de traiter les requêtes en mode DELETE en retournant quand ça se passe bien le code **204/NO_CONTENT** sans aucune donnée JSON en accompagnement :

```
@Delete(':id')
@HttpCode(204) //if no return json message
    async remove(@Param('id') id:number) {
      const deletedOk = await this.customerService.remove(id);
      //Void method (no value returned)
    }
```

On peut également directement utiliser l'api de bas niveau express pour gérer par code une partie du couple (requête, réponse) Http :

```
import { Controller, Req, Res } from "@nestjs/common";
import { Request, Response } from "express";

@Controller("examples")
export class ExamplesController {
  @Post("rawInfos")
  exampleRequestObjectExpress(@Req() req: Request, @Res() res: Response) {
    const responseData = {
      approach: "express",
      routeParams: req.params,
      queryParams: req.query,
      body: req.body,
      headers: req.headers,
      ip: req.ip,
      ips: req.ips,
      hostname: req.hostname,
      subdomain: req.subdomains,
    };
    res.status(200).json(responseData);
  }
}
```

5. Gestion automatique des erreurs (ExceptionFilter)

Comportement par défaut sans aucun paramétrage :

- En cas d'**exception non rattrapée** (sans try/catch au niveau du contrôleur)
  → *INTERNAL_SERVER_ERROR/500*
  { "statusCode": 500, "message": "Internal server error"}

- Si le contrôleur remonte une exception de type HttpException('message' , statusCode)
  (exemple : **throw new HttpException**('Forbidden', **HttpStatus.FORBIDDEN**);)
  → *{ "statusCode": 403, "message": "Forbidden"}*

Filtre d'erreur personnalisé :

common/error.exception.filter.ts

```
import { ExceptionFilter, Catch, ArgumentsHost, HttpException, HttpStatus } from '@nestjs/common';
import {  Response } from 'express';

//pour traiter throw new HttpException('no yyy for id='+id, HttpStatus.NOT_FOUND);
//ou bien throw new HttpException('yyy', HttpStatus.YYYY);
@Catch(HttpException)
export class HttpExceptionFilter implements ExceptionFilter {
   catch(exception: HttpException, host: ArgumentsHost) {
      const ctx = host.switchToHttp();
      const response = ctx.getResponse<Response>();
      //const request = ctx.getRequest<Request>();
      const status = exception.getStatus();
     console.log("*** exception catched by HttpExceptionFilter:"+exception.message + " " + exception.stack);
      response
        .status(status)
        .json({
          statusCode: status,
          //path: request.url,
          message : exception.message
        });
    }
}

//pour traiter throw new Error(`NOT_FOUND: news not found with id=${id}`);
//ou bien  throw new Error(`XXX: message_xxx`);
@Catch(Error)
export class ErrorExceptionFilter implements ExceptionFilter {
  catch(error: Error, host: ArgumentsHost) {
    console.log("*** error catched by ErrorExceptionFilter:"+error.message + " " + error.stack);
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    //const request = ctx.getRequest<Request>();
    const fullMessage = error.message;
    const messageParts = fullMessage.split(':');
    const message = messageParts.length>1?messageParts[1].trim():fullMessage;
    const statusString = messageParts.length>1?messageParts[0]:null;
    let status = HttpStatus.INTERNAL_SERVER_ERROR; //by default
    switch(statusString){
       case "NOT_FOUND" : status = HttpStatus.NOT_FOUND; break;
       //...
```

```
    }
    response
      .status(status)
      .json({
        statusCode: status,
        timestamp: new Date().toISOString(),
        //path: request.url,
        message:message
      });
  }
}
```

NB : Les classes ***HttpExceptionFilter*** et ***ErrorExceptionFilter*** ci dessus permettent de contrôler la manière dont une exception sera transformée en réponse HTTP/JSON .

Au sein de l'exemple précédent, ***HttpExceptionFilter*** fait à peu près le traitement par défaut d'une exception de type **HttpException** et la classe ***ErrorExceptionFilter*** sert à traiter des instance de la classe **Error** . La méthode .catch() de ***ErrorExceptionFilter*** va essayer d'extraire de statut HTTP en analysant la valeur du message qui pourra par exemple commencer par "NOT_FOUND :" ou bien "CONFLICT :…." .

NB : L'enregistrement des filtres à utiliser peut s'effectuer via **@UseFilters()** au niveau d'un @Controller ou bien globalement via **app.useGlobalFilters()** au niveau de **main.ts**

Exemple :

customer.controller.ts

```
import { ...HttpException, HttpStatus,..., UseFilters } from '@nestjs/common';
...
import { ErrorExceptionFilter, HttpExceptionFilter } from 'src/common/error.exception.filter';


@Controller('customers')
@UseFilters(new ErrorExceptionFilter(),new HttpExceptionFilter())
export class CustomerController {
 ...

   @Get(':id')
   async getCustomerById(@Param('id') id:number): Promise<CustomerL1Dto> {
      return this.customerService.findOne(id);
       //ErrorExceptionFilter may return NOT_FOUND if necessary
   }

    @Delete(':id')
    //@HttpCode(204) if no return json message
    async remove(@Param('id') id:number) {
      const deletedOk = await this.customerService.remove(id);
      return new Message("customer with id="+id + " is now deleted");//with default 200/OK
      //ErrorExceptionFilter may return NOT_FOUND if necessary
    }

   //{"id": "1" ,  "firstName" : "prenom_x" , "lastname" : "nom_y" }
   @Put(':id') //or Patch(":id")
```

```
  //@HttpCode(204) if no return json message
  async update(@Body() customerToUpdate: CustomerL1Dto,
                  @Param('id') id:number): Promise<CustomerL1Dto> {
    customerToUpdate.id=id; //must be coherent
    const updatedOk =   await this.customerService.update(id, customerToUpdate);
    let updatedCustomer = await this.customerService.findOne(id);
    return updatedCustomer; //with default 200/ok
     //ErrorExceptionFilter may return NOT_FOUND if necessary
  }
}
```

Au sein de cet exemple, le contrôleur invoque sur le service des méthodes qui renvoient (ici par convention spécifique à un projet) des instance de 'Error' avec un message commençant par "NOT_FOUND: ..." ou "XYZ: ….'

```
  async _throwNotFoundErrorInNotExists(id:number,messagePart:string){
    const doesExit = await this.accountRepository.exists({ where: { num: id } });
      if(!doesExit)
        throw new Error(`NOT_FOUND: ${messagePart} with id=${id}`);
   }


   async remove(id: number): Promise<boolean>{
    await this._throwNotFoundErrorInNotExists(id,"no existing account to delete");
    try{
      const deletedResult = await this.accountRepository.delete(id); //deletedResult
      //console.log("deletedResult=" + JSON.stringify(deletedResult));
      return (deletedResult.affected===1);
    }catch(ex){
      const subErrorPrefix = (ex instanceof Error)?`${ex.message}:`:"";
     throw new Error(`${subErrorPrefix}Exception in AccountService.remove() with id==${id}`);
    }
   }
```

Ainsi , les erreurs/exceptions les plus fréquentes (de type Error(…)) remontées par les services se propagent automatiquement au niveau appelant (@Controller) et sans try/catch , la gestion automatique des exceptions (via des *ExceptionFilter* vus précédemment) fabrique automatiquement des bons retours d'erreurs HTTP/JSON (avec bons messages et bons statuts HTTP) .

IV - Sécurité (authentification , ...)

1. Eventuelles autorisations CORS

Dans **main.ts**

```
...
app.enableCors();
...
```

2. Authentification et autorisations avec NestJs

2.1. En mode "standalone – JWT"

```
npm install --save --legacy-peer-deps bcrypt
npm install --save-dev --legacy-peer-deps @types/bcrypt
npm install --save --legacy-peer-deps @nestjs/jwt
```

NB: l'option *--legacy-peer-deps* n'est utile qu'en cas d'éventuels conflits entre versions des dépendances indirectes.

2.1.a. Module "users" pour comptes utilisateurs (accès realm)

nest g module users

nest g service users

NB : Il y a plein de façons de stocker et relire des données d'authentification liées aux utilisateurs autorisés à se connecter à l'application.

L'exemple de code suivant sera basé sur :

- un **cryptage essentiel des passwords stockés en base** via l'algorithme ad hoc **bcrypt**

- une persistance des comptes utilisateurs dans une base "mongoDb" .

users/dto/**users.dto.ts**

```
import { AutoMap } from "@automapper/classes";
import { ApiProperty} from "@nestjs/swagger";

export class UserL0Dto{

   @ApiProperty({default:'myUsername'})
   @AutoMap()
   public username: string;

   @ApiProperty({default:'myFirstName'})
   @AutoMap()
   public firstName? : string;
```

```
  @ApiProperty({default:'myLastName'})
  @AutoMap()
  public lastName? : string;

  @ApiProperty({default:'aaa.bbb@xyz.com'})
  @AutoMap()
  public email? : string ; // | undefined or | null is possible but complex (automapper, ...)

  @ApiProperty({default:'pwd'})
  @AutoMap()
  public newPassword? : string;

  @ApiProperty({default:'user_of_sandboxrealm'})
  @AutoMap()
  public mainGroup? : string;

  constructor( username : string="myUsername", firstName : string="myFirstName",
     lastName : string="myLastName" ,   email : string="aaa.bbb@xyz.com",newPassword : string="pwd",
     mainGroup : string="user_of_sandboxrealm"){
        this.username=username; this.firstName=firstName; this.lastName=lastName; this.email=email;
        this.newPassword=newPassword; this.mainGroup=mainGroup;
  }
}


export class UserL1Dto  extends UserL0Dto  {
  @ApiProperty()
  @AutoMap()
  public id: string;

    constructor(id: string="?",
     username : string="myUsername", firstName : string="myFirstName", lastName : string="myLastName" ,
     email : string="aaa.bbb@xyz.com",newPassword : string="pwd",mainGroup : string="user_of_sandboxrealm"){
        super(username,firstName,lastName,email,newPassword,mainGroup);
        this.id=id;
  }

}
```

users/entities/**users.entity.ts**

```
import { AutoMap } from '@automapper/classes';
import { Prop, Schema, SchemaFactory } from '@nestjs/mongoose';
import { HydratedDocument } from 'mongoose'

@Schema()
export class UserEntity  {

 //default @Prop  mongo _id: { type : ObjectId , alias : "id" }
 id:string;

 @Prop()
 @AutoMap()
 username: string;

 @Prop()
 @AutoMap()
 firstName?: string;

 @Prop()
 @AutoMap()
 lastName?: string;
```

```
  @Prop()
  @AutoMap()
  email?: string ;

  @Prop()
  @AutoMap()
  newPassword?: string;

  @Prop()
  @AutoMap()
  mainGroup?: string;
}

export type UserDoc = HydratedDocument<UserEntity>;
export const UserSchema = SchemaFactory.createForClass(UserEntity);
```

users/mapper/**users.mapper.profiles.ts**

```
...
  override get profile() {
    return (mapper: Mapper) => {
      createMap(mapper, UserL0Dto,UserEntity);
      createMap(mapper, UserL1Dto,UserEntity);
      createMap(mapper, UserEntity,UserL1Dto,
          forMember( d=> d.id, mapFrom(s => s.id))
        );
    };
  }
```

users/**users.module.ts**

```
...
@Module({
  imports: [
    MongooseModule.forFeature([ { name: "Users", schema : UserSchema}]),
  ],
  exports:[UsersService],
  providers: [UsersService,UserMapperProfile],
  controllers: [UsersController]
})...
```

users/**users.service.ts**

```
...
import * as bcrypt from 'bcrypt';

@Injectable()
export class UsersService {

  constructor(
    @InjectModel('Users') private readonly userModel: Model<UserDoc>,
    @InjectMapper() private readonly classMapper: Mapper,
    ){}

   // …  findAll() , findOne() , …. , remove() , ...
```

```
  async findByUsername(username: string):Promise<UserL1Dto>{
     try{
        const userDoc = await this.userModel.findOne({ username: username}).exec();
        return this.classMapper.mapAsync(userDoc,UserEntity,UserL1Dto);
     }catch(ex){
        throw new Error(`NOT_FOUND: user not found with username=${username}`);
     }
  }

  async bcryptPassword(password:string){
     const saltOrRounds = 10;
     return  await bcrypt.hash(password, saltOrRounds);
  }

  async create(user: UserL0Dto): Promise<UserL1Dto> {
     const userToCreate = user; // compatible , more simple
     //const userToCreate = this.classMapper.map(user,UserL0Dto,UserEntity); //ok
     let persistentUserEntity = new this.userModel(userToCreate);
     persistentUserEntity.newPassword=await
                    this.bcryptPassword(persistentUserEntity.newPassword??"");
     const savedUserAsUserEntity = <UserEntity> <any> await persistentUserEntity.save();
     return this.classMapper.mapAsync(savedUserAsUserEntity,UserEntity,UserL1Dto);
  }


 async update(id: string, userDto: UserL1Dto): Promise<UserL1Dto> {
   const userToUpdate = userDto; // compatible , more simple
   //const userToUpdate = this.classMapper.map(userDto,UserL1Dto,UserEntity); //ok
   if(userToUpdate.newPassword!=null && userToUpdate.newPassword.charAt(0)!='')
     userToUpdate.newPassword=await this.bcryptPassword(userToUpdate.newPassword??"");

   const updatedUserAsUserEntity = <UserEntity> <any> await
            this.userModel.findByIdAndUpdate( id, userToUpdate, { userToUpdate: true });
   if(updatedUserAsUserEntity==undefined)
     throw new Error(`NOT_FOUND: not existing user to update with id=${id}`);
   return this.classMapper.mapAsync(updatedUserAsUserEntity,UserEntity,UserL1Dto);
 }

}
```

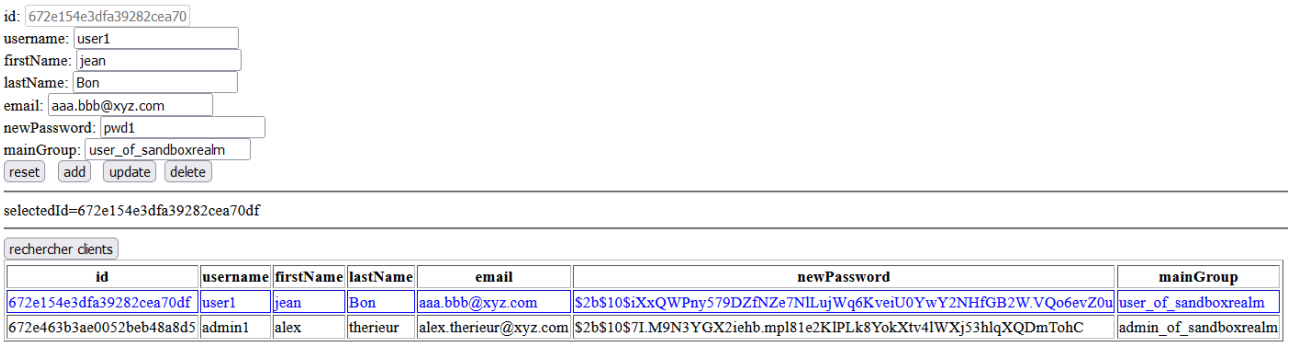
users/**users.controller.ts**

```
//code habituel qui va bien
```

Exemple de résultat de http://localhost:3000/news-api/users


```
[
{"username":"user1","firstName":"jean","lastName":"Bon",
"email":"aaa.bbb@xyz.com",
"newPassword":"$2b$10$iXxQWPny579DZfNZe7NlLujWq6KveiU0YwY2NHfGB2W.VQo6evZ0u",
"mainGroup":"user_of_sandboxrealm","id":"672e154e3dfa39282cea70df"},
{"username":"admin1","firstName":"alex","lastName":"therieur",
"email":"alex.therieur@xyz.com",
```

```
"newPassword":"$2b$10$7I.M9N3YGX2iehb.mpl81e2KlPLk8YokXtv4lWXj53hlqXQDmTohC",
"mainGroup":"admin_of_sandboxrealm","id":"672e463b3ae0052beb48a8d5"}
]
```

id	username	firstName	lastName	email	newPassword	mainGroup
672e154e3dfa39282cea70df	user1	jean	Bon	aaa.bbb@xyz.com	$2b$10$iXxQWPny579DZfNZe7NlLujWq6KveiU0YwY2NHfGB2W.VQo6evZ0u	user_of_sandboxrealm
672e463b3ae0052beb48a8d5	admin1	alex	therieur	alex.therieur@xyz.com	$2b$10$7I.M9N3YGX2iehb.mpl81e2KlPLk8YokXtv4lWXj53hlqXQDmTohC	admin_of_sandboxrealm

Les utilisateurs user1 et admin1 ont par hasard le même mot de passe "pwd1" . Les cryptages "bcrypt" de "pwd1" donnent des résultats différents $2...u et $2...C et ce comportement est normal.

2.1.b. Module "auth" pour authentification

```
nest g module auth
nest g controller auth
nest g service auth
```

`auth/dto/`**`auth.dto.ts`**

```
import { AutoMap } from "@automapper/classes";
import { ApiProperty, PartialType } from "@nestjs/swagger";


export class LoginRequest{
    @ApiProperty({default:'myUsername'})
    @AutoMap()
    public username: string;

    @ApiProperty({default:'pwd'})
    @AutoMap()
    public password : string;

    constructor( username : string="myUsername", password : string="pwd"){
        this.username=username; this.password=password;
    }
}

export class LoginResponse{
    @ApiProperty({default:'myUsername'})
    @AutoMap()
    public username: string;

    @ApiProperty({default:'successfull login OR login failed'})
    @AutoMap()
    public message : string;

    @ApiProperty({default:false})
```

```
    @AutoMap()
    public status : boolean;

    @ApiProperty({default:null})
    @AutoMap()
    public token : string | null ;

    @ApiProperty({default:null})
    @AutoMap()
    public scope: string | null ;

    constructor( username : string="myUsername"){
        this.username=username;
    }
}
```

auth/**auth.service.ts**

```
import { Injectable, UnauthorizedException } from '@nestjs/common';
import { UsersService } from 'src/users/users.service';
import * as bcrypt from 'bcrypt';
import { LoginRequest, LoginResponse } from './dto/auth.dto';
import { JwtService } from '@nestjs/jwt';

@Injectable()
export class AuthService {
    constructor(private usersService: UsersService,private jwtService: JwtService ) {
    }

  async login(loginRequest :LoginRequest): Promise<LoginResponse> {
    const user = await this.usersService.findByUsername(loginRequest.username);
    const cryptedPassword = user.newPassword;
    const isMatch = await bcrypt.compare(loginRequest.password, cryptedPassword??"");

    /*if (!isMatch) {
      throw new UnauthorizedException();
    }*/
   let loginResponse=new LoginResponse(loginRequest.username);
   if(isMatch){
     loginResponse.status=true; loginResponse.message="successful login";
     switch(user.mainGroup){
        case "admin_of_sandboxrealm":
            loginResponse.scope="resource.read resource.write resource.delete"; break;
        case "manager_of_sandboxrealm":
            loginResponse.scope="resource.read resource.write"; break;
        case "user_of_sandboxrealm":
        default:
            loginResponse.scope="resource.read";
    }
    const jwtPayload = { sub: user.id, username: user.username ,
                         scope: loginResponse.scope , name: `${user.firstName} ${user.lastName}` };
    loginResponse.token = await this.jwtService.signAsync(jwtPayload);
   }else{
```

```
    loginResponse.status=false; loginResponse.message="login failed";
    loginResponse.token=null;
   }
   return loginResponse;
  }

}
```

auth/**auth.controller.ts**

```
import { Body, Controller, Get, HttpCode, HttpStatus, Post} from '@nestjs/common';
import { AuthService } from './auth.service';
import { LoginRequest } from './dto/auth.dto';
import { Message } from 'src/common/message';

@Controller('auth')
export class AuthController {
    constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @Post()
  login(@Body() loginRequest:LoginRequest) {
    return this.authService.login(loginRequest);
    /*if (loginResponse.status==false) {
      throw new UnauthorizedException();
    }*/
   //loginResponse/200 may contains .status=false .message="login failed" and .token=null
  }
}
```

auth/**auth.module.ts**

```
...
@Module({
  imports:[UsersModule],
  controllers: [AuthController],
  providers: [AuthService]
}) export class AuthModule {}
```

Comportement du Web Service de Login (en tenant compte de la configuration ci-après) :

POST ***http://localhost:3000/news-api/auth***

{	{
"username": "user1",	"username": "user1",
"password": "pwd1"	"password": "wrong-pwd"
}	}

username: user1 password: pwd1 login logout authenticated successful login name: ? , may be seen in token after parsed oauth2/oidc scope: resource.read	username: user1 password: wrong-pwd login logout not authenticated login failed name: ? , may be seen in token after parsed oauth2/oidc scope: undefined

200/OK
{
"username": "user1",
"status": true,
"message": "successful login",
"scope": "resource.read",
"token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiI2NzJlMTU0ZTNkZmEzOTI4MmNlYTcwZGYiLCJ1c2VybmFtZSI6InVzZXIxIiwic2NvcGUiOiJyZXNvdXJjZS5yZWFkIiwibmFtZSI6ImplYW4gQm9uIiwiaWF0IjoxNzMxNTAzNjY2LCJleHAiOjE3MzE1MDQyNjZ9.xiWWGPfjlBiF-X-_IsoBdhi2TaYcNFQDkQuVJFp_O3c"
}
	200/OK
	{
	"username": "user1",
	"status": false,
	"message": "login failed",
	"token": null
	}

2.1.c. Configuration pour @nestjs/jwt :

auth/**constants.ts**

```
export const jwtConstants = {
    secret: 'MySecret_007_James_Bond',
  };
```

app.module.ts

```
...
import { JwtModule } from '@nestjs/jwt';
import { jwtConstants } from './auth/constants';
import { APP_GUARD } from '@nestjs/core';
import { AuthGuard } from './auth/auth.gard';

@Module({
  imports: [
    JwtModule.register({
      global: true,
      secret: jwtConstants.secret,
      signOptions: { expiresIn: '600s' },
    }),
  ...,
```

```
  MongooseModule.forRoot('mongodb://localhost:27017/news'),
  NewsModule,  AuthModule,  UsersModule
  ],
  controllers: [AppController],
  providers: [AppService,
    {
      provide: APP_GUARD,
      useClass: AuthGuard,
    }
  ],
})
export class AppModule {}

//NB: authentification JWT selon https://docs.nestjs.com/security/authentication
```

NB : Pour enregistrer le gardien qui va vérifier la présence de jeton valide dans la partie ***Authorization : Bearer ….*** de l'entête HTTP de la requête entrante , on peut soit :

- utiliser localement **@UseGuards(AuthGuard)** près de @Get ou @Post ou autre sur un contrôleur
- **ou bien** globalement ajouter **{ provide: APP_GUARD, useClass: AuthGuard }**dans la partie providers:[…] de **app.module.ts**

auth/**auth.guard.ts**

```
import {  CanActivate,  ExecutionContext,  Injectable,  UnauthorizedException,
  } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { jwtConstants } from './constants';
import { Request } from 'express';

  @Injectable()
  export class AuthGuard implements CanActivate {

    constructor(private jwtService: JwtService) {
    }

   async canActivate(context: ExecutionContext): Promise<boolean> {

    const request = context.switchToHttp().getRequest();
    const token = this.extractTokenFromHeader(request);
    if (!token) {
      throw new UnauthorizedException();
    }
    try {
      const payload = await this.jwtService.verifyAsync(
        token,
        {
          secret: jwtConstants.secret
        }
      );
      // 💡 We're assigning the payload to the request object here
      // so that we can access it in our route handlers
      request['user'] = payload;
```

```
    } catch {
      throw new UnauthorizedException(); //{"statusCode":401,"message":"Unauthorized"}
      //if return false ---> 403/Forbidden by default
    }
    return true;
  }

  private extractTokenFromHeader(request: Request): string | undefined {
    const [type, token] = request.headers.authorization?.split(' ') ?? [];
    return type === 'Bearer' ? token : undefined;
  }
}
```

@Public() pour ne pas tout bloquer

auth/public.decorator.ts

```
import { SetMetadata } from '@nestjs/common';


export const IS_PUBLIC_KEY = 'isPublic';
export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
```

Ce nouveau décorateur **@Public()** peut servir à **spécifier quelles sont les méthodes** (des contrôleurs) **que l'on peut déclencher sans authentification préalable** .

Tout le monde doit au moins pouvoir tenter le login() et l'on peut donc placer @Public() au dessus de la méthode login() de AuthController :

auth/**auth.controller.ts**

```
…
import { Public } from './public.decorator';

@Controller('auth')
export class AuthController {
   constructor(private authService: AuthService) {}

  @HttpCode(HttpStatus.OK)
  @Post()
  @Public()
  login(@Body() loginRequest:LoginRequest) {
    return this.authService.login(loginRequest);
   //loginResponse/200 may contains .status=false .message="login failed" and .token=null
  }
}
```

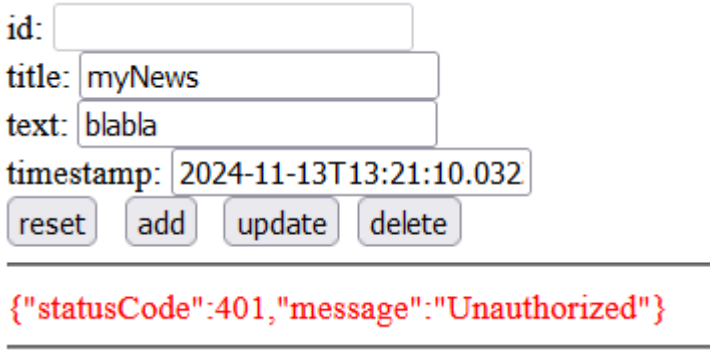
Il faut penser à améliorer le code du gardien **AuthGuard** de manière à tenir compte de l'éventuelle présence de la décoration @Public() sur la route associée à la requête entrante :

Amélioration de auth/**auth.guard.ts**

```
import { IS_PUBLIC_KEY } from './public.decorator';
import { Reflector } from '@nestjs/core';
…
@Injectable()
  export class AuthGuard implements CanActivate {
     constructor(private jwtService: JwtService,
     private reflector: Reflector
    ) {}

    async canActivate(context: ExecutionContext): Promise<boolean> {

      const isPublic = this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
        context.getHandler(),
        context.getClass(),
      ]);
      if (isPublic) {
        // with @Public in a @Controller code
        return true;
      }
     …
    }
}
```

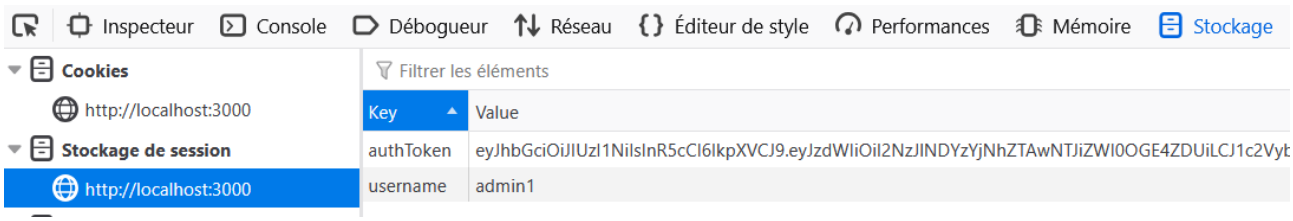
Comportement de l'application (avec authentification en place) :

Sans login préalable :



Avec login correct préalable :

jeton stocké dans zone "sessionStorage" du navigateur :



et retransmis selon le protocole HTTP en mode "Bearer" :

Ce jeton valide est testé et accepté par le gardien AuthGuard:

id: 6734a858b925f39aff49fe0
title: myNews
text: blabla
timestamp: 2024-11-13T13:23:20.705
reset add update delete

selectedId=6734a858b925f39aff49fe04

rechercher news

id	title	text	timestamp
672c938dbaf41fced60342eb	newsTitle	text of news	2024-11-07T10:20:08.525Z
6734a858b925f39aff49fe04	myNews	blabla	2024-11-13T13:23:20.705Z

2.2. Autorisations nestJs (roles ou scopes , 403/Forbidden, ...)

NB: un jeton JWT peut comporter (dans sa partie "claim") , une information de type "roles" ou "scopes"
Exemples: "roles" : "admin manager" ou "scopes" : "resource.read resource.write resource.delete"

Sigle **RBAC** = **Role-based access Control** : on vérifie si l'utilisateur connecté a ou pas le ou les rôle(s) requis .

Variante avec scopes (liste de droits élémentaires complémentaires) : : on vérifie si l'utilisateur connecté a ou pas l'ensemble des scopes (droits élémentaires) requis .

Décorateur @HasRoles ou @HasScope() :

auth/**rolesOrScope.decorator.ts**

```
import { SetMetadata } from '@nestjs/common';

/*
export enum Role {
  User = 'user',
  Admin = 'admin',
  Manager = 'manager',
}

export const HAS_ROLES_KEY = 'hasRoles';
export const HasRoles = (...roles: Role[]) => SetMetadata(HAS_ROLES_KEY, roles);
*/
//------------------

export const HAS_SCOPE_KEY = 'hasScopes';
export const HasScopes = (...scopes:string[]) => SetMetadata(HAS_SCOPE_KEY, scopes);
```

Gardien associé :

auth/**scopes.guard.ts**

```
import { Injectable, CanActivate, ExecutionContext } from '@nestjs/common';
import { Reflector } from '@nestjs/core';
import { HAS_SCOPE_KEY } from './rolesOrScope.decorator';

@Injectable()
export class ScopesGuard implements CanActivate {
  constructor(private reflector: Reflector) {}

  canActivate(context: ExecutionContext): boolean {
    const requiredScopes = this.reflector.getAllAndOverride<string[]>(HAS_SCOPE_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    console.log("in ScopesGuard, requiredScopes="+requiredScopes);
    if (!requiredScopes) {
      return true;
    }
```

```
    const request = context.switchToHttp().getRequest();
    //Hypothèse importante:
    //ce gardien est enregistré en seconde position , après AuthGuard
    //qui a déjà extrait le jeton JWT de l'entête HTTP
    //et qui a déjà extrait et stocker la partie "claim" au sein de request['user']
    const userClaim = request['user'];

    console.log("in ScopesGuard, userClaim="+JSON.stringify(userClaim));
    return requiredScopes.some((s) => userClaim.scope?.includes(s));
    //NB: array.some(predicate) return true if predicate is ok for each item of array
    //if returning false --> {"statusCode":403,"message":"Forbidden resource"}
  }

}
```

Amélioration de **app.module.ts** pour **enregistrer un deuxième gardien :**

```
...
providers: [AppService,
    {  provide: APP_GUARD,
       useClass: AuthGuard,
    },
    {   provide: APP_GUARD,
        useClass: ScopesGuard,
    }  ],
…
```

Exemple d'utilisation de @HasRole() ou @HasScope() :

```
export class NewsController {
…
    @Post()
    @HasScopes("resource.write") //ou bien @HasScopes("resource.read" , "resource.write")
     //ou bien @HasRoles(Role.Admin) ou bien @HasRoles("admin") ou ...
    async create(@Body() news: NewsL0Dto): Promise<NewsL1Dto> {
       return this.newsService.create(news);//returning news with generated id
     }
```

Comportement :

En cas de droit insuffisant (ex : manque le scope "resource.write" pour déclencher un POST) , le framework nestJs retourne automatiquement ce genre de message de réponse :

{ **"statusCode": 403,** "message": "Forbidden resource", "error": "**Forbidden**"}

Exemples :

username,password	mainGroup	scopes	POST news
user1,pwd1	user_of_sandboxrealm	resource.read	403/Forbidden
admin1,pwd1	admin_of_sandboxrealm	resource.read resource.write resource.delete	200/OK

2.3. En mode OAuth2 avec @nestjs/passport

NB : au sein de l'exemple suivant , la vérification des jetons JWT sera délégué à un serveur spécialisé (ici de type exact "keycloak") support le protocole OAuth2/OIDC .

En tant que couche haute, le framework nestJs, peut s'appuyer sur le framework nodeJs spécialisé dans l'authentification : "passport" comportant lui même plein de variantes/extensions appelées "stratégies" .

Documentation générale de référence : https://docs.nestjs.com/recipes/passport

```
npm install --save --legacy-peer-deps  @nestjs/passport passport passport-keycloak-bearer
```

common/**oauth.strategy.ts**

```
import * as KeycloakBearerStrategy from 'passport-keycloak-bearer';
import { PassportStrategy } from '@nestjs/passport';
import { Injectable } from '@nestjs/common';


@Injectable()
export class OAuthStrategy extends
            PassportStrategy(KeycloakBearerStrategy,"myOauthKeycloakStrategy") {
  constructor() {
    super({
       "realm": "sandboxrealm",
       "url": "https://www.d-defrance.fr/keycloak"
     }); //paramétrages selon stratégie utilisée (ici  passport-keycloak-bearer)
  }

  extractOidcUserInfosFromJwtPayload(jwtPayload:any){
    return {
      username : jwtPayload.preferred_username,
      name : jwtPayload.name,
      email : jwtPayload.email,
      scope : jwtPayload.scope
    }
  }

  async validate(jwtPayload: any) {
    //console.log("OAuthStrategy.validate() with jwtPayload="+ JSON.stringify(jwtPayload));
    const user = this.extractOidcUserInfosFromJwtPayload(jwtPayload);
    //console.log("OAuthStrategy.validate() with user="+ JSON.stringify(user));
    return user; //stored in request and can be analysed by other guard (ex: ScopeGuard)
  }
}
```

Rappels sur décorateurs habituels :

public.decorator.ts vu comme **@Public()**	import { SetMetadata } from '@nestjs/common'; export const IS_PUBLIC_KEY = 'isPublic'; export const Public = () => SetMetadata(IS_PUBLIC_KEY, true);
rolesOrScope.decorator.ts vu comme **@HasScopes("….")**	import { SetMetadata } from '@nestjs/common'; export const HAS_SCOPE_KEY = 'hasScopes'; export const HasScopes = (...scopes:string[]) => SetMetadata(HAS_SCOPE_KEY, scopes);

Gardien héritant de @nestJs.AuthGuard et tenant compte de @Public :

common/**my-auth.guard.ts**

```
import { Injectable, ExecutionContext } from "@nestjs/common";
import { Reflector } from "@nestjs/core";
import { AuthGuard } from "@nestjs/passport";
import { IS_PUBLIC_KEY } from "./public.decorator";

@Injectable()
export class MyPublicPrivateAuthGuard extends AuthGuard('myOauthKeycloakStrategy') {
  constructor(private reflector: Reflector) {
    super();
  }

  canActivate(context: ExecutionContext) {

    const isPublicViaDecorator = this.reflector.getAllAndOverride<boolean>(IS_PUBLIC_KEY, [
      context.getHandler(),
      context.getClass(),
    ]);
    if (isPublicViaDecorator) {
      return true; //because @Public() is present
    }
    /*
    const request = context.switchToHttp().getRequest();
    if( request.path.includes("/public/")){
      return true; //because request uri contains "/public/"
    }
    */
    //all others request are "private" by default and handle by inherited passport decorator:
    return super.canActivate(context);
  }
}
```

Code de **ScopesGuard** = identique que celui de la version "standalone jwt" (sans délégation

oauth2) → voir paragraphe ou chapitre précédent .

Configuration de l'authentification **@nestjs/passport** au sein de **app.module.ts :**

```
...
import { APP_GUARD } from '@nestjs/core';
import { AuthGuard, PassportModule } from '@nestjs/passport';
import { OAuthStrategy } from './common/oauth.stategy';
import { ScopesGuard } from './common/scopes.guard';
import { MyPublicPrivateAuthGuard } from './common/my-auth.guard';

@Module({
 imports: [   ...,
   PassportModule,
   NewsModule , XyzModule
 ],
 controllers: [AppController],
 providers: [AppService , OAuthStrategy,
   {     provide: APP_GUARD,
       /*useClass: AuthGuard('myOauthKeycloakStrategy'),*/
        useClass:  MyPublicPrivateAuthGuard //401 if no valid token , ...
   },
   {     provide: APP_GUARD,
          useClass: ScopesGuard,  //403 if no valid scope
   }   ]
})
export class AppModule {}
```

Utilisation (identique à celle du mode "standalone_JWT") :

```
  @Get()   @Public()
  async findByCriteria(): Promise<NewsL1Dto[]> {
     return  this.newsService.findAll();
  }

  @Post()   @HasScopes("resource.write")
  async create(@Body() news: NewsL0Dto): Promise<NewsL1Dto> {
     return this.newsService.create(news);//returning news with generated id
   }
```

V - Communication entre microservices

…. à rédiger ...

ANNEXES

VI - Annexe – Bibliographie, Liens WEB + TP

1. Bibliographie et liens vers sites "internet"

2. TP