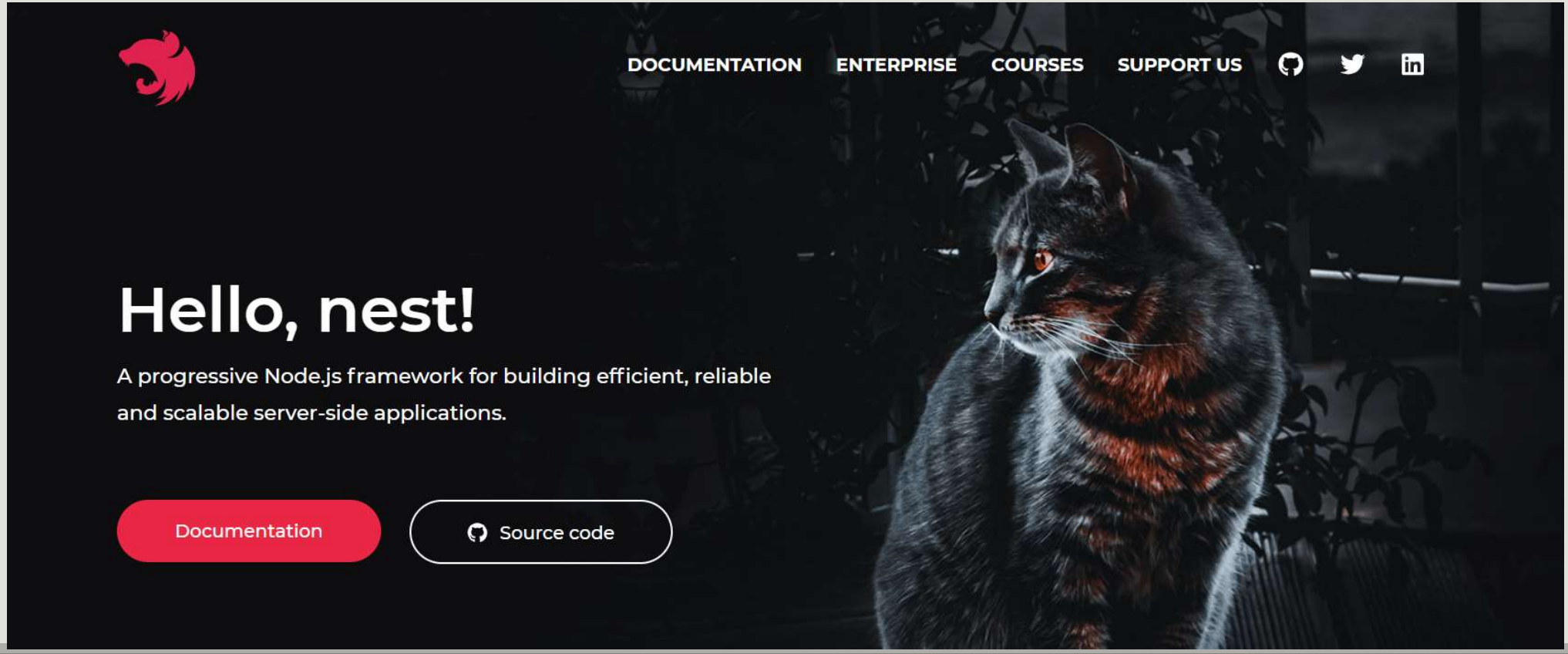
NestJs

AYMEN SELLAOUTI

Références

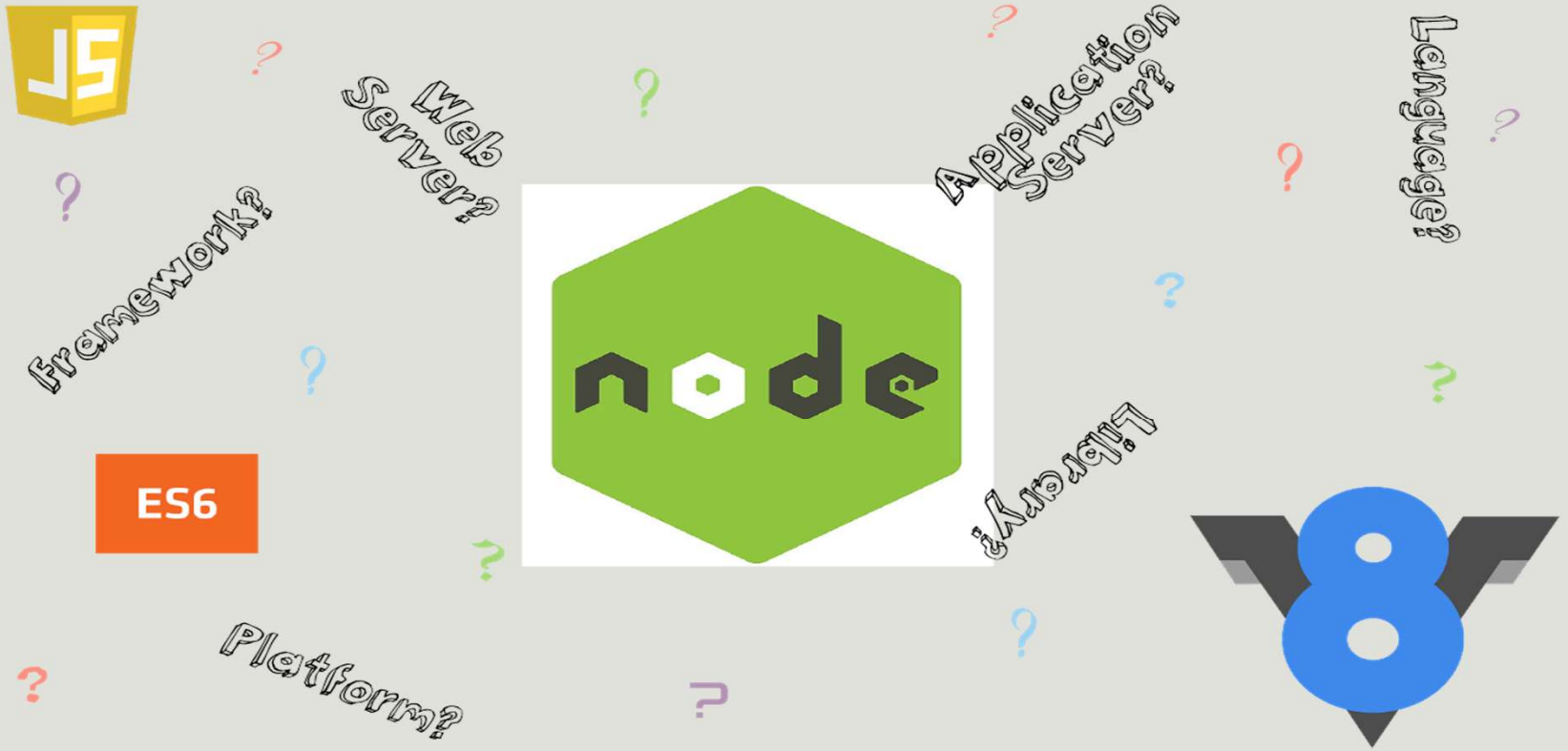


https://nestjs.com/

Plan du Cours

1) **Introduction**
2) **Les modules**
3) **Les Data transfer Object (DTO)**
4) **Les Middlewares**
5) **Les providers**
6) **Les pipes**
7) **Les filtres**
8) **Les intercepteurs**
9) **Les variables de configuration**
10) **Interaction avec une Base de données via TypeORM**
11) **Authentification et Autorisation**

C’est quoi NodeJs?



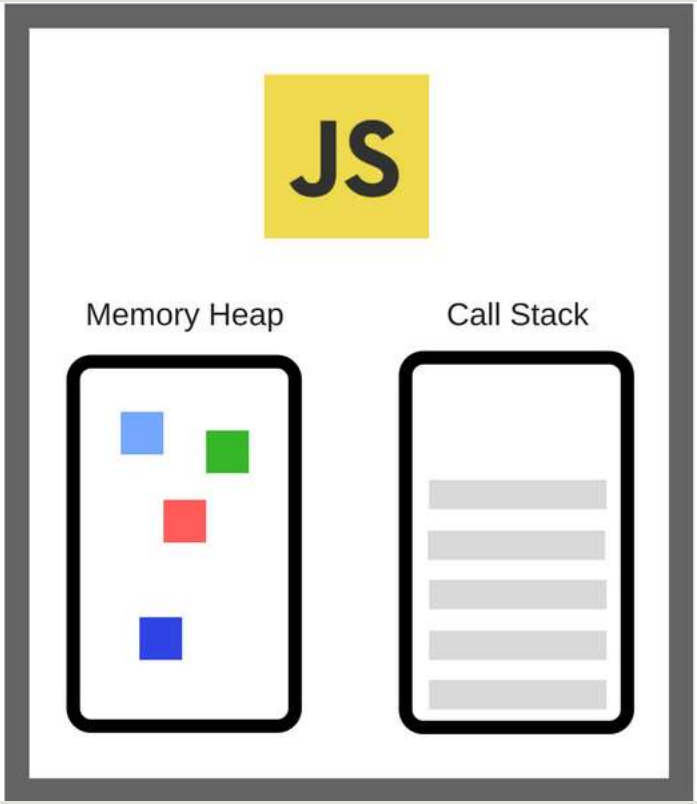
C’est quoi NodeJs?

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

- Environnement d’exécution JS
- Utilise V8 le moteur JS de Google.

Moteur Javascript : Javascript Engine

- Tout code JavaScript que vous écrivez a besoin d'un moteur Javascript pour l'exécuter.
- Le moteur JavaScript possède deux composants principaux:
  - **Memory Heap** : sert à allouer la mémoire utilisée dans le programme.
  - **Call Stack** (pile d'exécution) : contient les données des fonctions exécutées par le programme.



Moteur Javascript : Javascript Engine

V8 Javascript Engine

- V8 est un moteur JavaScript développé par Google et utilisé par chrome pour exécuter du code JavaScript.
- Il est *Open Source*
- Ecrit en C++

https://v8.dev/

Qu'est ce qu'un environnement d'exécution JS

- Un **environnement d'exécution** JS est l'endroit ou votre code JS va être exécuté. C'est **la ou vivra votre JavaScript Engine**.
- Il va, entre autre, **déterminer quels variables globales** sont accessibles pour vous **(window pour le *"runtime environment"* de votre browser)**
- **Ajax, DOM et d'autres API's ne font pas parti de JavaScript**. C'est l'environnement d'exécution Js (Ici fourni par votre Browser) qui les rend disponible.

https://v8.dev/

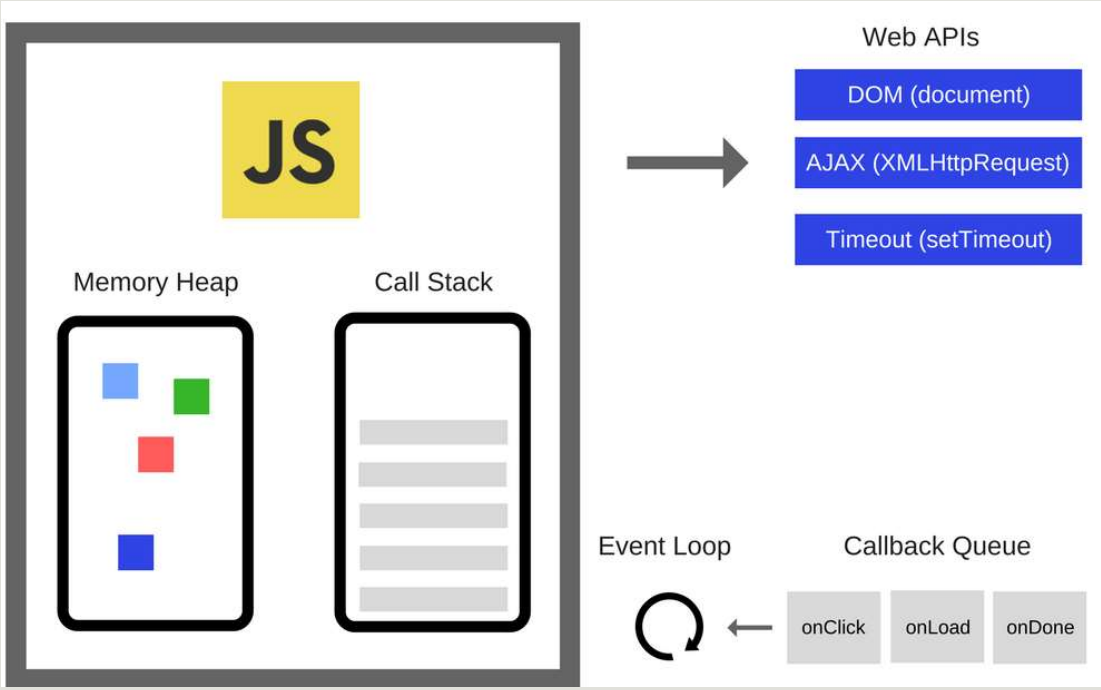
Est-ce que JavaScript est Synchrone ou Asynchrone ?

- Js est **SYNCHRONE**.
- Les fonctions asynchrones telles que **setTimout ne font pas partie de JS.**
- Elle font partie des API **de votre *runtime environment*.**
- Donc JS peut **agir d'une façon ASYNCHRONE mais ceci n'est pas innée**, ce n'est pas dans son Core mais plutôt du aux APIs offertes par le ***runtime***.
- Dans NodeJs, c'est **libuv qui permet cet aspet Asynchrone**.

https://nodejs.org/en/about/

Est-ce que JavaScript est Synchrone ou Asynchrone ?

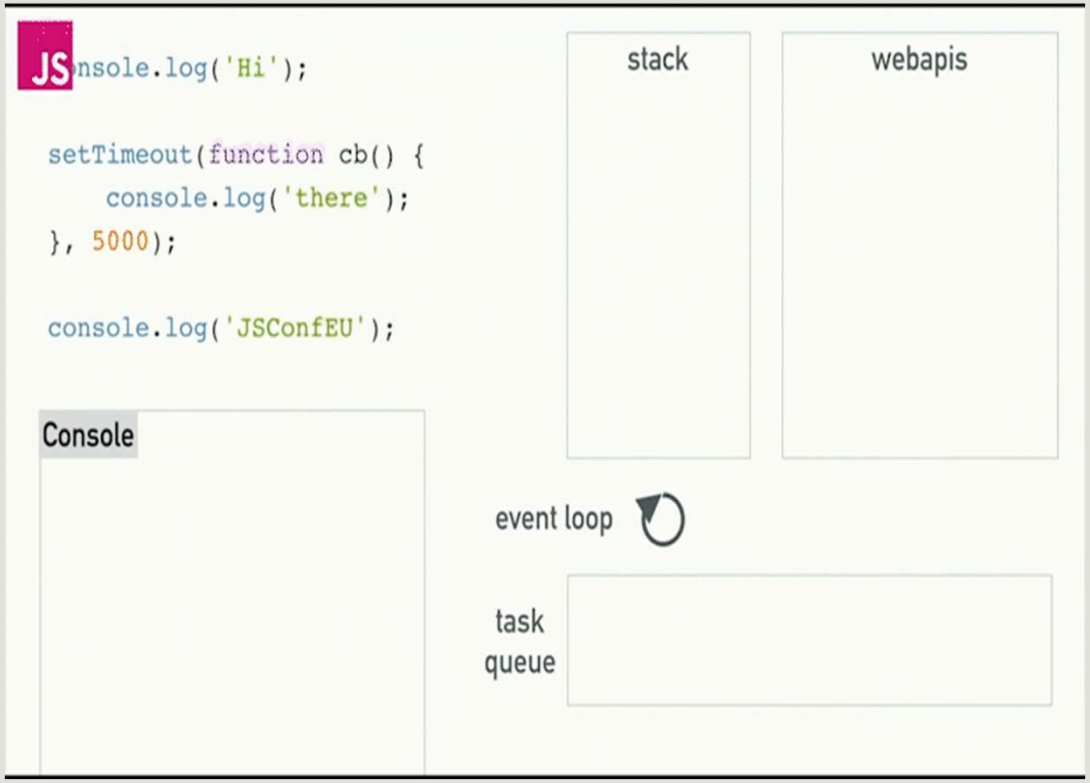
- Si on gardait uniquement l'aspect Synchrone de JS, ca sera un problème énorme avec le browser.
- La solution proposée par les API est l'ajout de la composante asynchrone via ***l'event loop*** et la ***callback queue***.
- Ceci va permettre d'effectuer des traitements sans bloquer la pile d'exécution du moteur JS.



https://v8.dev/

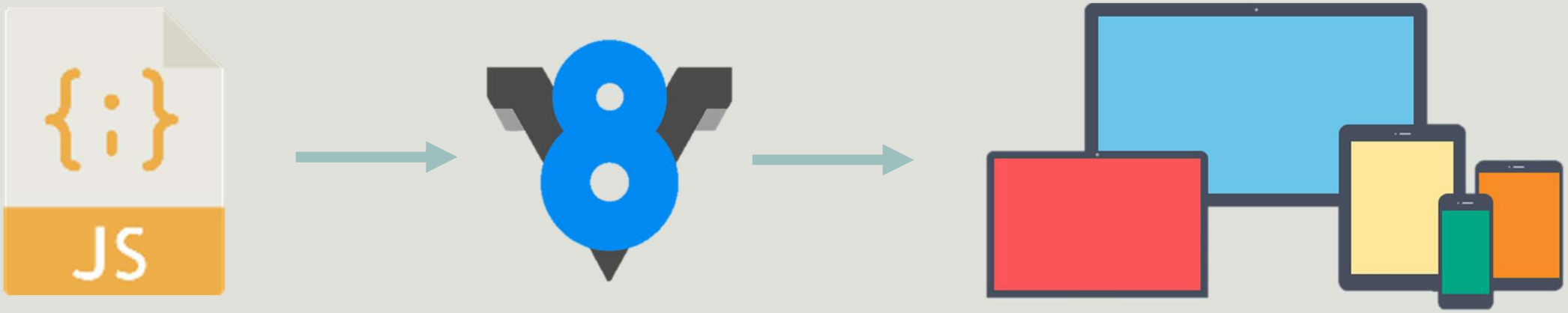
Comment ca marche dans le browser ???

- A part le moteur JS, votre environnement possède des **API's** qui permettent à votre Moteur JS **d'exécuter certaines fonctionnalités** comme le **timeout**, ou **fetch**.
- Pour le browser vous avez l'API DOM.



Comment est né Node ?

- Au départ on ne pouvait exécuter Js que dans un Browser. C'est lui qui fournissait son environnement d'exécution.



- **Une question a été donc posée. Pouvons nous exécuter du Js en dehors du browser ? Pouvons nous avoir du Js côté Serveur ?**

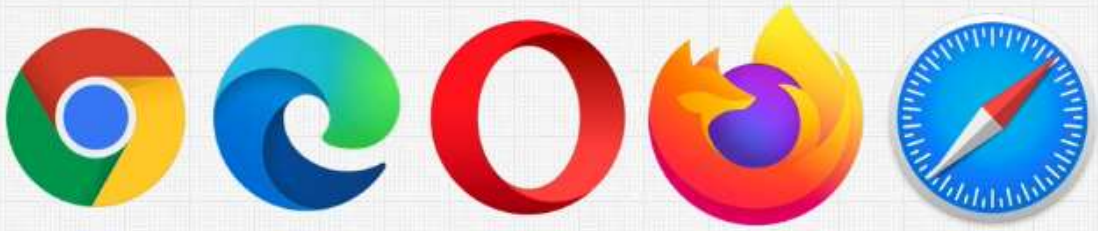
Comment est né Node ?

2009

2008

libuv

Comment est né Node ?



window
document
history
location

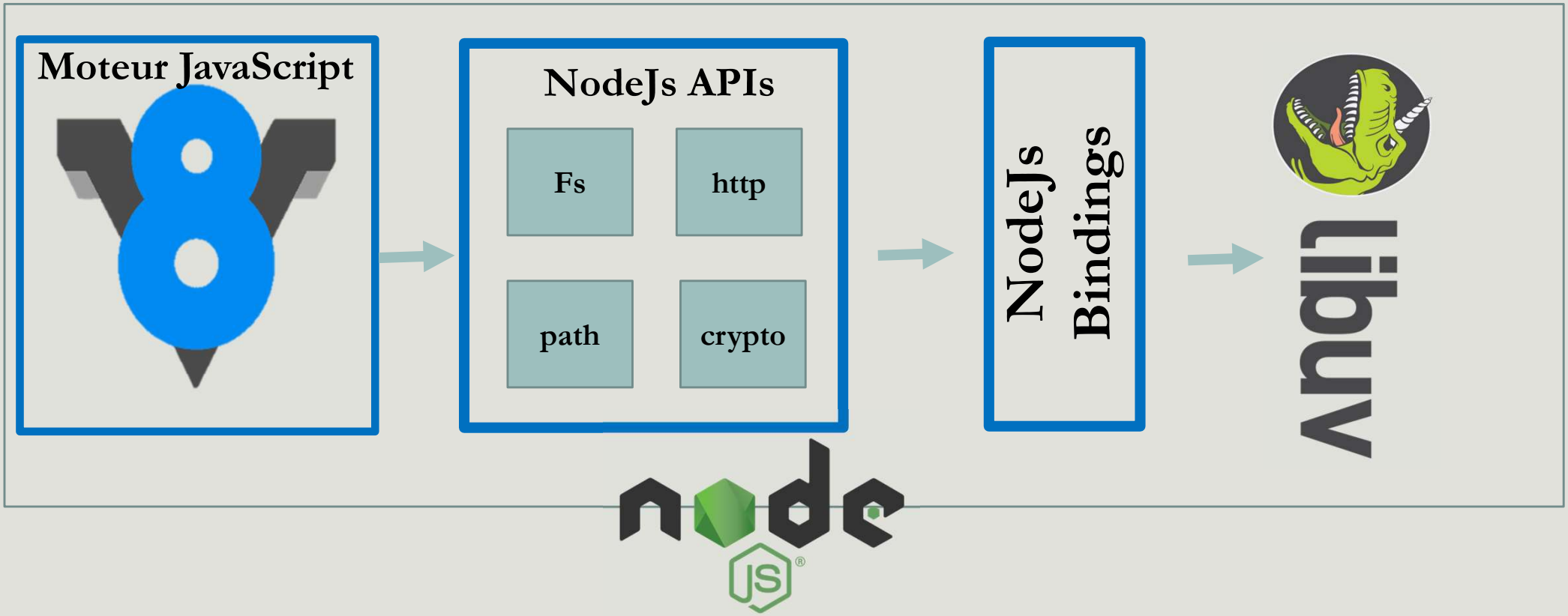
global
process
module
__filename

https://v8.dev/

NodeJs un environnement d'exécution

- Afin de s'exécuter, Javascript a besoin d'un environnement d'exécution.
- C'est comme un conteneur qui contient tout ce qu'il vous faut pour exécuter du code Js.
- Le noyau de l'environnement d'exécution est le moteur Js.
- Il y a aussi les APIs Node
- Libuv la bibliothèque qui permet de gérer les I/O Asynchrone
- Les nodes Js Binding

NodeJs un environnement d'exécution



Est-ce que NodeJs est Multi thread ?

- NodeJs est monothread au départ puis le multithrading a été introduit.
- Vous avez encore V8 comme Moteur Js.
- La gestion de l'aspect Asynchrone est offerte par la bibliothèque libuv

https://blog.logrocket.com/node-js-multithreading-worker-threads-why-they-matter/

libuv

libuv

node JS

- libuv est une bibliothèque multiplatforme écrite en C
- Son rôle est de gérer les opération asynchrone d'I/O
- Elle permet à plusieurs langages de s'y connecter via les connecteurs de bindings.
- Elle offre la partie principale qui permet de gérer les traitements Asynchrone de NodeJs, l'EVENT LOOP.

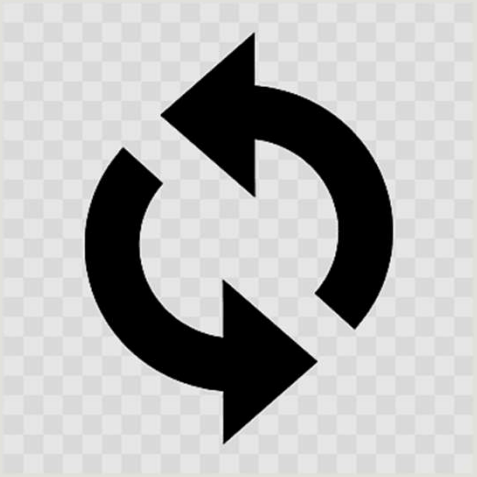
https://libuv.org/

libuv

libuv

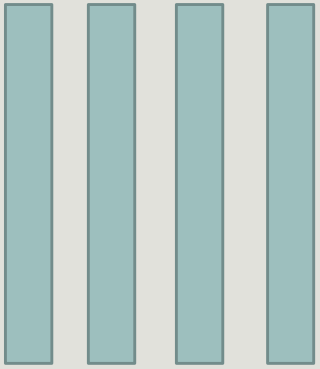
libuv

node JS



Event Loop

Thread Pool



Asynchrone I/O

Gestion de fichiers

Réseau

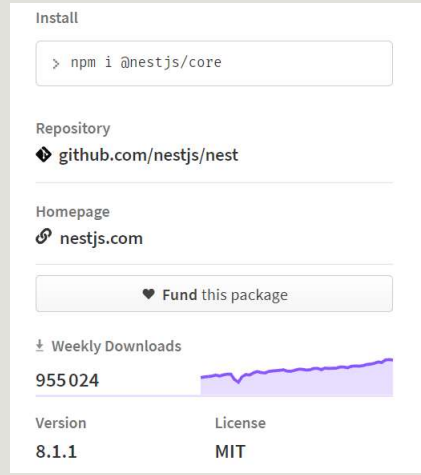
https://libuv.org/

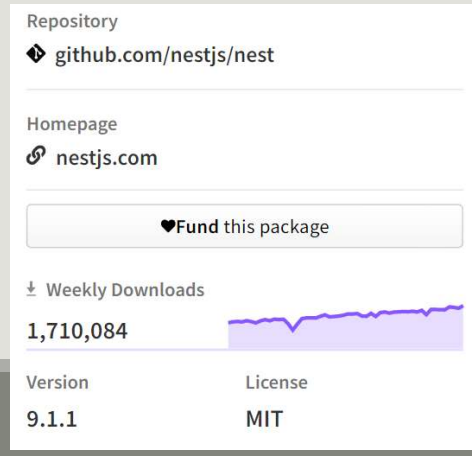
Introduction

- Nest (NestJs) est un **framework NodeJs.**
- Utilise **Typescript**
- NestJs est basé sur **ExpressJs** (par défaut) ou si vous le souhaitez sur Fastify.
- Nest ajoute une couche d'abstraction sur ses deux Framework et il permet aussi d'utiliser leur modules.

Introduction
Pourquoi NestJs

- C'est un FRAMEWORK
- Une communauté en croissance continue et très active
- La documentation
- Possibilités de construire des Rest API's, des applications MVC, des microservices, d'exposer du GraphQL, des Web Sockets ou des CLI's et des CRON jobs.
- Designs patterns implémentés et prêts à l'emploi (e.g. IOC)
- Une cli puissante qui facilite le développement
- Plusieurs modules prêts à l'emploi
- Intégration facile de plusieurs technologies





Installation

- Afin d'installer Nest vous devez avoir NodeJs (>= 10.13.0) pour la V8 et >=16.13 pour la v9 et la v10.

- Vous pouvez ensuite cloner un projet prêt à l'emploi ou utiliser le Nest Cli

- Pour installer le Nest Cli vous lancer la commande :

  npm i -g @nestjs/cli

- Une fois le nest CLI installé, vous avez accès à la commande **nest** qui vous permettra de créer votre projet et de scafolder du code.

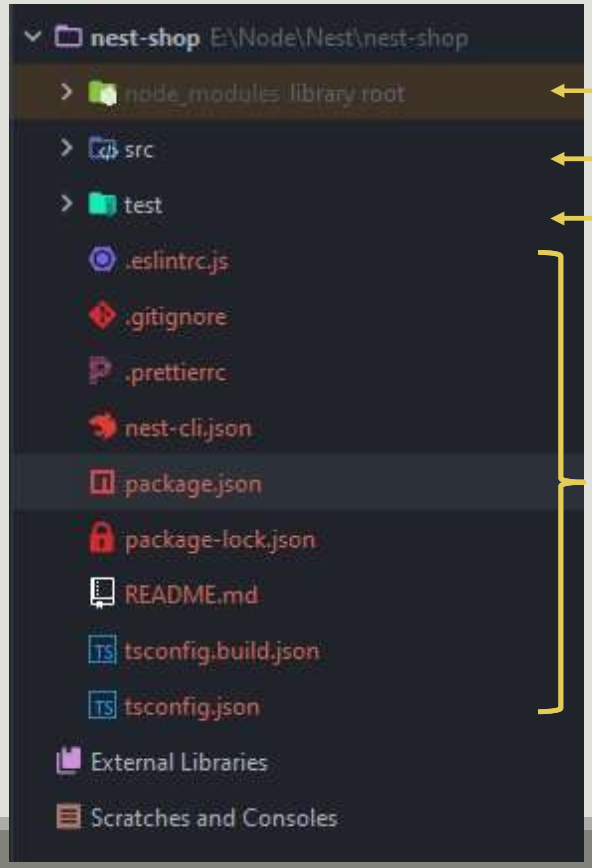
Installation

- Pour créer un nouveau projet, lancer la commande **nest new NomProjet.**

- Une fois fini, pour lancer votre projet taper la commande
  - **npm run start:dev**

  Ou
  - **nest start –watch**
- Votre application est maintenant accessible sur le port 3000 via l'url localhost:3000

Structure d'un projet Nest



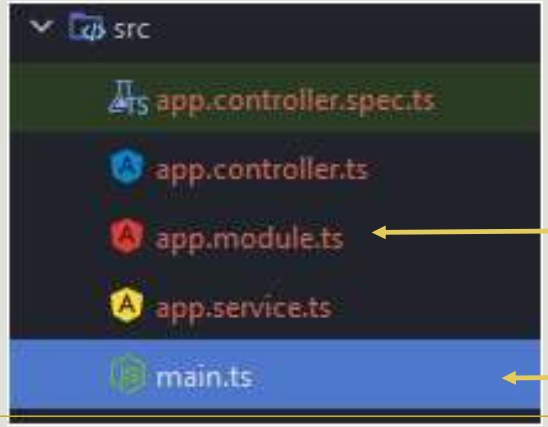
Dépendances de votre projet

Votre code source

Tests

Configuration

Structure d'un projet Nest



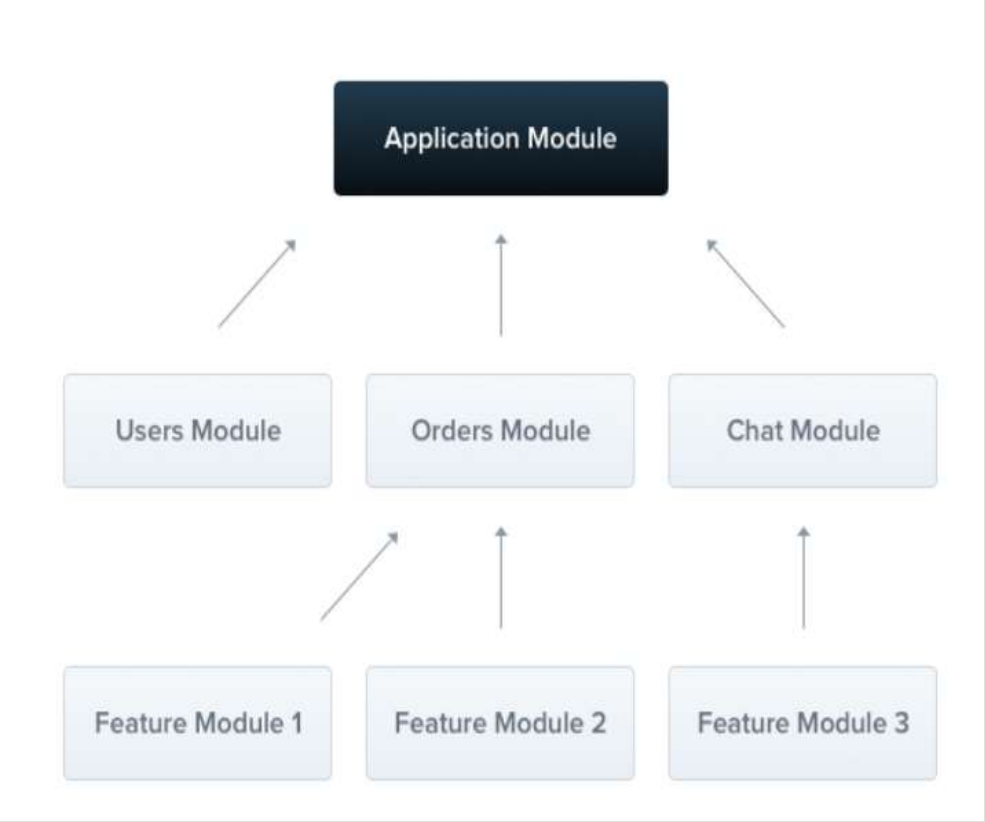
Le module principal de votre application

C'est le fichier principal de votre application qui utilise le NestFactory afin de créer une instance d'une application Nest.
main.ts contient une fonction asynchrone qui déclenchera votre application

```
1  import { NestFactory } from '@nestjs/core';
2  import { AppModule } from './app.module';
3
4  async function bootstrap() {
5    const app = await NestFactory.create(AppModule);
6    await app.listen( port: 3000);
7  }
8  bootstrap();
```

Les modules

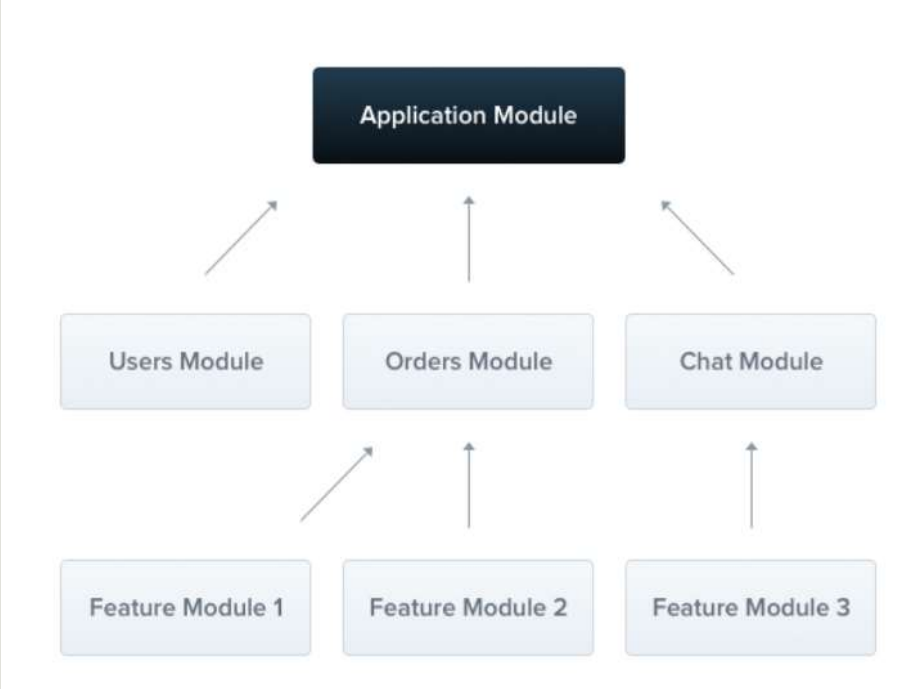
- NestJs a choisi de décomposer les projets en **Modules**.
- Un module est une **partie isolée de votre application**.
- Elle **encapsule** plusieurs fonctionnalités liées. Par exemple le module des utilisateurs qui se charge de gérer les users, les rôles ect.
- On peut dire qu'un module inclura les **fonctionnalités nécessaires pour un métier** de votre application.



https://docs.nestjs.com/modules

Les modules

- Un module est une **classe annotée** avec un décorateur **@Module().** Le @Module() fournit des métadonnées que Nest utilise pour organiser la structure de l'application.
- Chaque application possède au moins un module c'est le module racine.
- Le module racine est le point de départ utilisé par Nest pour créer le graphe de l'application.
- Nest **recommande fortement la décomposition en Modules**.



https://docs.nestjs.com/modules

Les modules

- Les paramètres de l'annotation @Module() sont :

providers	les providers qui seront instanciés par l'injecteur Nest et qui peuvent être partagés au moins sur ce module.
controllers	l'ensemble des contrôleurs définis dans ce module
imports	la liste des modules importés qui exportent les providers requis dans ce module.
exports	Les providers fournis par ce module et qui peuvent être utilisés par d'autres modules.

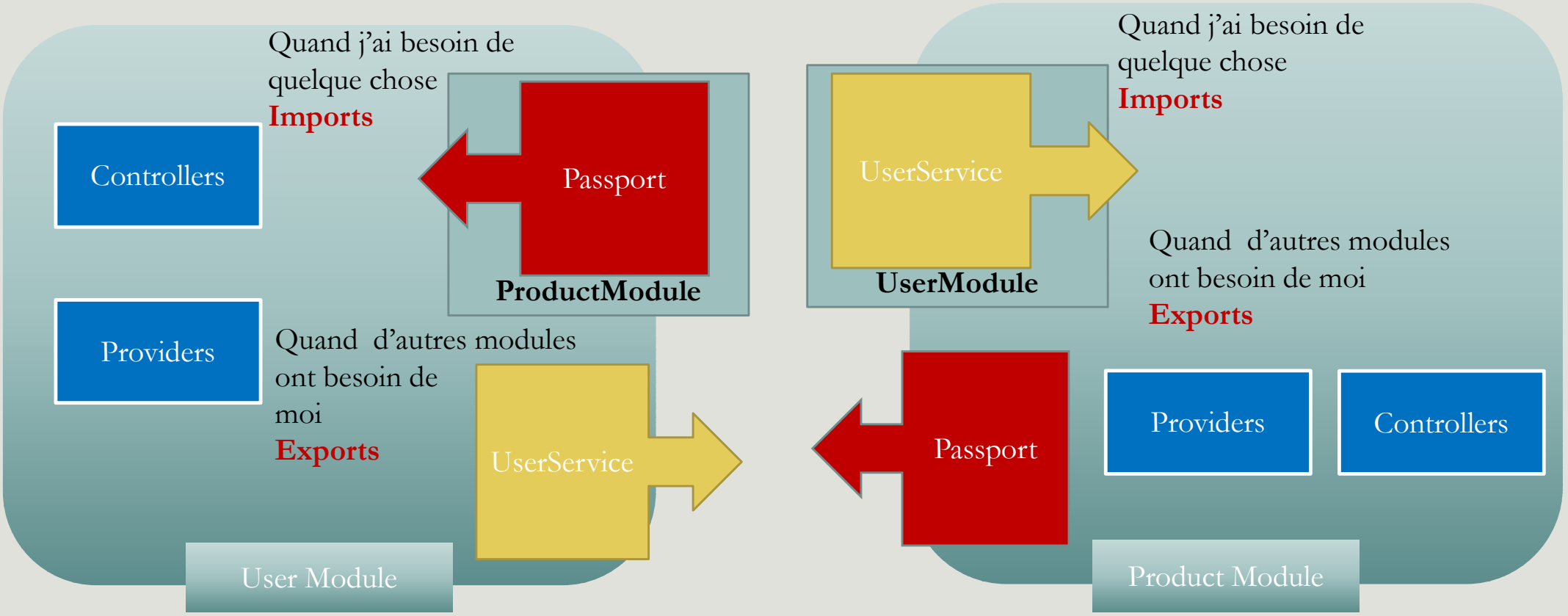
Les modules

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [],
  exports: [],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

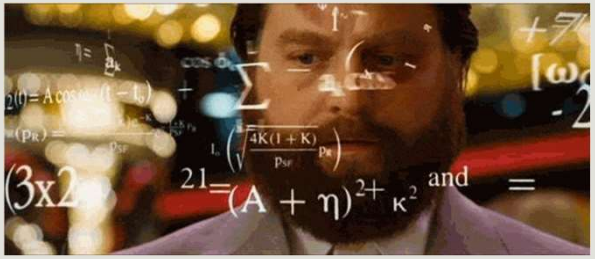
Les modules

- Par défaut, les modules encapsulent leurs **providers**
- Ceci implique que les providers sont uniquement accessible à l'intérieur de ce module par défaut.
- Nous pouvons conclure donc que dans un module, nous ne pouvons utiliser (réellement injecter) que **ses providers** ou **ceux exportés par les modules qu'il a importé**.
- NestJs décrit les providers exportés par un module comme son **Interface** ou son **API Publique**.

Les modules


Quand j'ai besoin de quelque chose
Imports
Controllers
Passport
ProductModule
Providers
Quand d'autres modules ont besoin de moi
Exports
UserService
User Module
Quand j'ai besoin de quelque chose
Imports
UserService
UserModule
Quand d'autres modules ont besoin de moi
Exports
Passport
Providers
Controllers
Product Module

Exercice



- Créer un module Premier.
- Intégrer le avec votre App.module.ts

Les modules

- Vous pouvez créer un module Nest via le Cli en utilisant la commande

nest generate **mo**dule nomDuModule

Ou via le raccourci :

nest g mo nomDuModule

Global Module
@Global

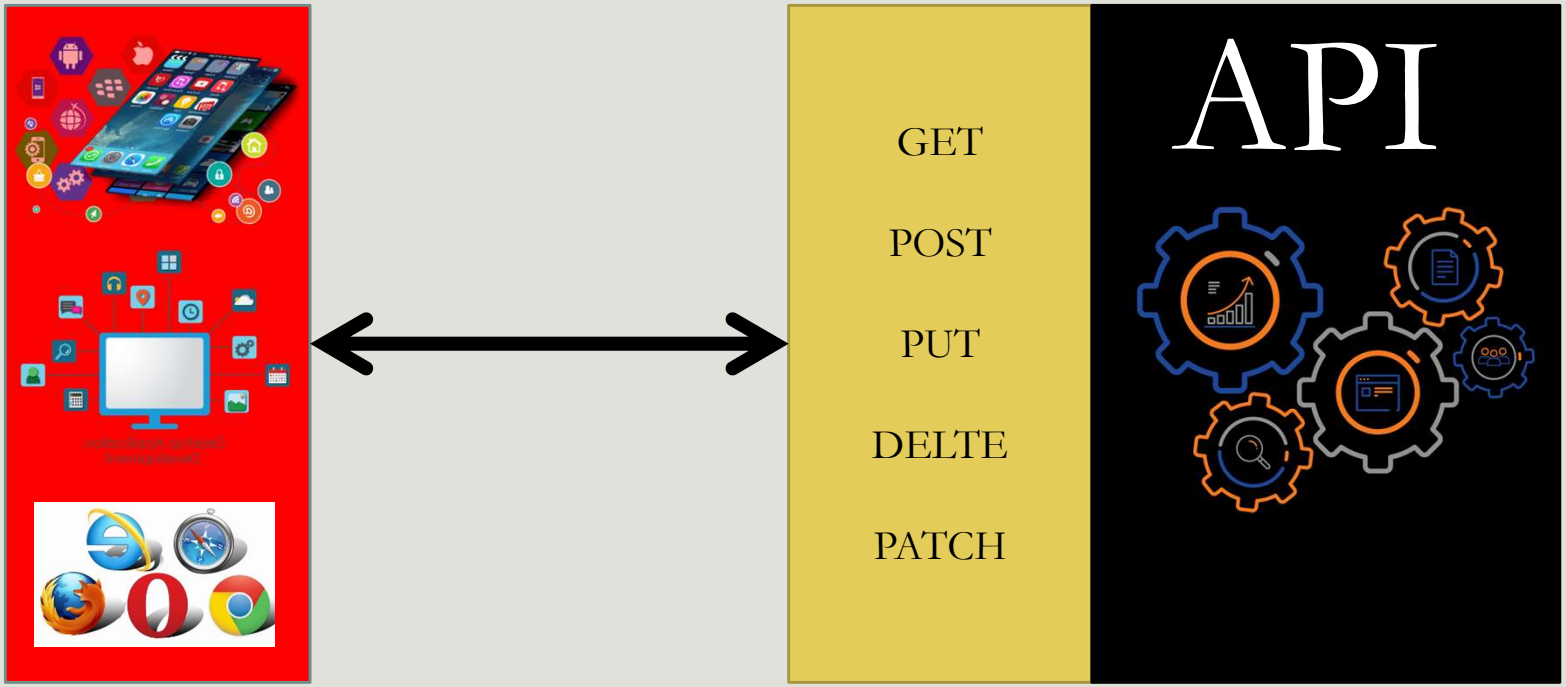
- Si vous devez importer un module partout, vous pouvez le déclarer Globale en utilisant le décorateur **@Global**.
- En mettant ce module en global, il **doit être enregistré une seule fois**.
- Généralement c'est le module principal qui s'en charge.
- Une fois définis le CatsModule est **visible partout**, vous n'avez pas besoin d'importer le module.

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

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

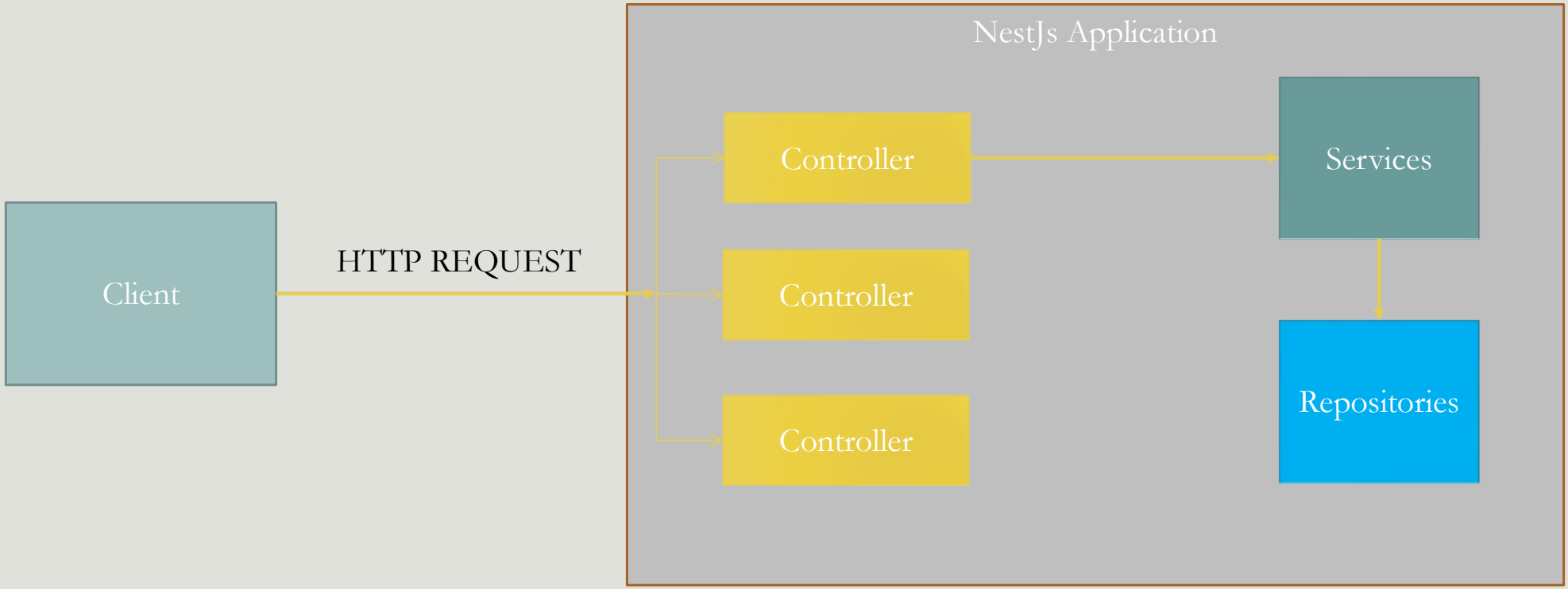
Les contrôleurs
Rest

- REST (**RE**presentational **S**tate **T**ransfer) est un style architectural, un design Pattern pour les API.
- Une API RestFull est une API qui utilises le protocole HTTP pour GET, PUT, POST et DELETE les données.

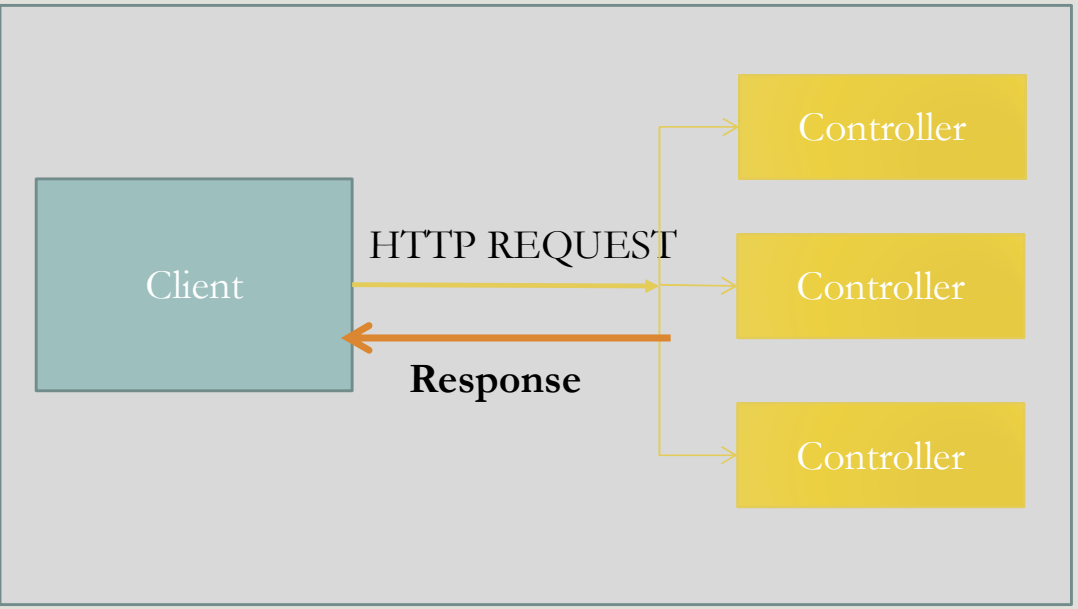
REST API



Architecture



Les contrôleurs



- Le rôle d'un Controller est de réceptionner les **requêtes** HTTP entrantes, de préparer une **réponse** et de la retourner.
- Le mécanisme de **routage** contrôle quel contrôleur reçoit quelle **requête**.
- Chaque contrôleur peut gérer **plusieurs routes**.
- Chaque route est responsable d'une action.
- Afin de créer un contrôleur, 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** permettant de lier les demandes aux contrôleurs correspondants.

Les contrôleurs

- Pour identifier une classe comme étant un contrôleur, vous devez l'annoter (la décorer) avec **@Controller et l'ajouter dans son module sous la clé controllers.**

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

@Controller()
export class AppController {

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

```
@Module({
 imports: [],
 controllers: [AppController],
 providers: [AppService],
})
export class AppModule {}
```

Les contrôleurs

- Pour résumer un contrôleur est une classe avec l'annotation @Controller contenant un groupement de méthodes (handlers) permettant de gérer les requêtes http envoyées par vos clients

- Pour créer le contrôleur, vous pouvez le faire manuellement en créant la classe, en la décorant avec @Controller et en l'ajoutant dans le module associé, ou via la commande

nest **g**enerate **co**ntroller controllerName

nest **g co** controllerName

Routing

- Une **route** va identifier **l'uri associé à une action (un handler)**.
- Nest propose des **décorateurs** (annotations) permettant de définir la route associée à **une action** de votre contrôleur.
- Pour **chaque méthode HTTP** vous avez un décorateur associé.
- Le décorateur prend en paramètre **l'uri** à gérer. Ceci nous permet d'avoir une combinaison **uri + méthode** identifiant exactement la requête HTTP à gérer par votre action.
- Les méthodes les plus utilisées sont :

@Get(") , @POST("), @Delete("), @Put(") et @Patch(")

Routing

- Dans cet exemple, si vous faite une requête Get sur la route localhost:3000/test, la méthode getHello sera exécutée.
- Elle vous renverra la réponse 'HELLO NEST'

```
@Get('test')
getHello(): string {
  return 'HELLO NEST';
}
```

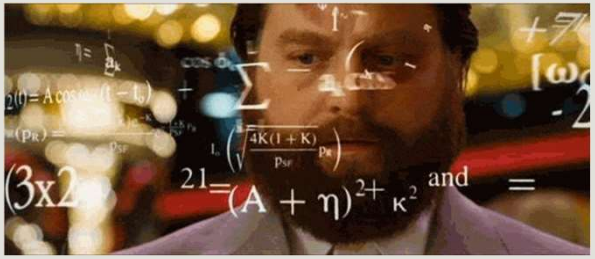
Routing
Préfixer les routes d'un contrôleur

- Vous pouvez préfixer les routes d'un contrôleur en indiquant le préfixe comme premier paramètre de l'annotation @Controller.

```
// Ici toutes les routes de ce contrôleur commenceront pas /tasks/
@Controller('tasks')
export class TasksController {
@Get('all')
getTasks(): string {
  // TODO
}
}
```

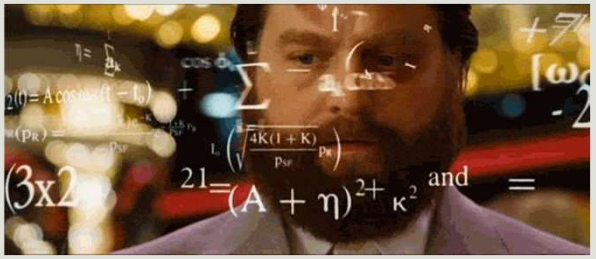
- Dans cet exemple, toutes les routes gérées par ce contrôleur seront préfixées par le segment 'tasks'. Donc pour accéder à la méthode getTasks, il faut l'uri /tasks/all

Exercice



- Créer un contrôleur premier dans le module premier.
- Faites le nécessaire pour gérer les méthodes GET, POST, DELETE, PUT et PATCH.
- Chaque appel devra uniquement loger le nom de la méthode appelée et la retourner.

Exercice



- Créer un module TodoModule.
- Créer un contrôleur TodoController et ajouter le au module TodoModule.
- Créer une Class TodoModel Représentant un Todo qui est caractérisé par son id, son name, sa description, sa date de création et son statut.
- Le statut doit être l'un de ces trois : En attente, En cours, Finalisé. Utiliser un **enum** pour définir ce type.
- Ajouter une méthode Get qui retourne la liste des todos qui est une propriété de votre TodoController.

```
import { Controller, Delete, Get, Patch, Post, Put }
from '@nestjs/common';
@Controller('todo')
export class TodoController {
  private todos = [];
  @Get()
  getTodos() {
   //  Todo 2 : Get the todo liste
  }
}
```

```
export enum TodoStatusEnum {
  'actif' = "En cours",
  'waiting' = "En attente",
  'done' = "Finalisé"
}
```

L'objet Request

- Si vous voulez récupérer l'objet Request (offerte par le framework que vous utilisez et qui est express par défaut), Nest vous offre une annotation vous permettant de le récupérer. C'est l'annotation **@Req.**

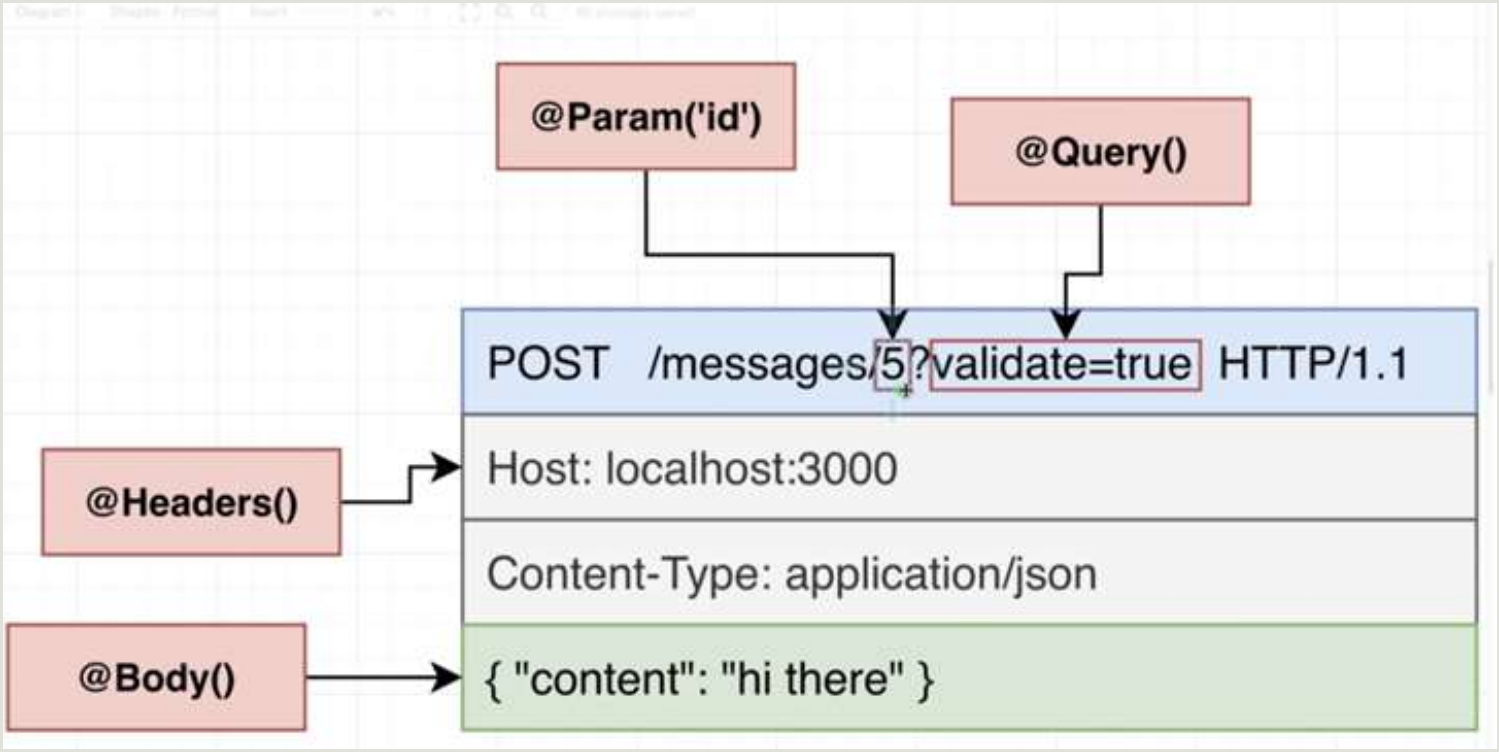
```
import { Controller, Get, Req } from '@nestjs/common';
import { Request } from 'express';

@Controller()
export class AppController {

  @Get()
  getHello(@Req() req: Request): string {
    console.log(req);
    return 'HELLO NEST';
  }
}
```

L'objet Request
Récupérer les différents éléments de la requête

- Au lieu de passer par l'objet **request** pour récupérer ce que vous voulez, il suffit d'utiliser les **decorateurs** offertes par Nest.

@Req ()	req	Récupérer l'objet Request
@Param(key?: string)	req.params / req.params[key]	Récupérer les paramètres du Body de votre requête
@Body(key?: string)	req.body / req.body[key]	Récupérer le body de votre requête
@Query(key?: string)	req.query / req.query[key]	Récupérer les queryParams envoyé en GET
@Headers(name?: string)	req.headers / req.headers[name]	Récupérer les Headers
@Ip()	req.ip	Contient l'adresse IP de la requête

Routing
Récupérer les différents éléments de la requête



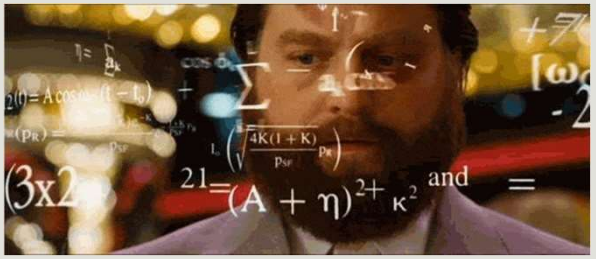
Contrôleur
Request Body : Récupérer le body d'une requête POST

- Pour récupérer le body d'une requête POST vous devez utiliser le décorateur @Body() dans les paramètres de votre action.

```
@Post('/test')
testPost(
  @Body() body
) {
  console.log(body);
}
```

- Pour récupérer un champ particulier du body, ajouter comme paramètre de votre annotation la clé de champ : @Body('name') vous permettra de récupérer le champ name de votre body

Exercice



- Dans votre contrôleur todo, créer une méthode qui permet d'ajouter un Todo.
- La génération de l'id doit être automatique. Penser à utiliser un générateur d'id unique comme le composant **uuid**.
- La date de création doit aussi être automatique.
- Le statut par défaut est « En attente ».

```
export enum TodoStatusEnum {
  'actif' = "En cours",
  'waiting' = "En attente",
  'done' = "Finalisé"
}
```

Response

- Lorsque vous retourner une réponse, un code **200 sera attribué automatiquement** (en arrière plan par Nest) à la réponse.
- Dans le cas d'un **POST,** le code sera de **201**.
- Vous pouvez aussi définir votre propre code avec l'annotation `@HttpCode(votreCode)`
- Nest vérifiera aussi le type de votre valeur de retour. Si c'est un objet ou un tableau il le sérialisera automatiquement.
- Si c'est une primitive JS, Nest envoi la valeur sans la sérialiser.
- Vous pouvez aussi gérer manuellement votre Réponse en injectant l'objet Response (avec le décorateur **@Res()**) du Framework de base de Nest (Express par défaut). Cette méthode n'est pas recommandée.

Routing
Définir des paramètres d'une route

- Lorsque vous créer une route et que vous voulez qu'un ou plusieurs de ces fragments soient dynamiques, préfixer les par **:**.

```
@Get('/post/:year/:id')
getPost(): string {
  return 'Post';
}
```

- En ajoutant un **?** devant le nom du paramètre, vous informer le routeur que ce paramètre est optionnel.

```
@Get('/post/:year/:id?')
getPost(): string {
  return 'Post';
}
```

Routing
Récupérer les paramètres d'une route

- Pour récupérer ces paramètres au niveau de votre action, utiliser le décorateur @Param() au niveau des paramètres de votre méthode.
- Si vous ne passez aucun paramètre au décorateur @Param, il vous retourne un tableau contenant tous les paramètres. Accéder ensuite via cet objet à la propriété que vous voulez avec le nom du paramètre.
- Si vous voulez accéder directement au paramètre, ajouter son nom comme paramètre de l'@annotation **@Param**

```
@Get('/post/:year/:id')
getPost(@Param() mesRoutesParams): string {
  return 'Post créer en :' + mesRoutesParams.year;
}
```

```
@Get('/post/:year/:id')
getPost(
  @Param('id') id,
  @Param('year') year
  ): string {
  return `Post d'id ${id}
  crée en ${year}`;
}
```

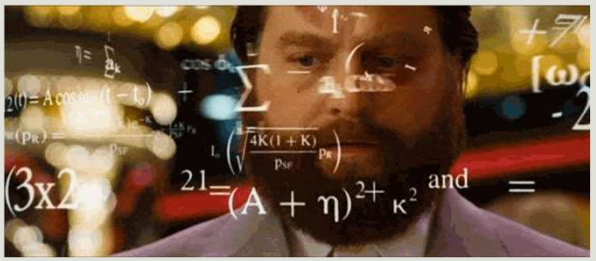
Routing
Les routes génériques

- Vous pouvez utiliser les expressions régulières pour définir vos routes.

- Vous pouvez utiliser par exemple les quantificateurs des expressions régulières :
  - * : 0 ou plusieurs
  - + : 1 ou plusieurs
  - ? : 0 ou 1 occurrence.

```
// cette route matchera n'importe quelle uri //commençant par test
@Get('test*')
getHello(@Req() req: Request): string {
  return 'HELLO NEST';
}
```

Exercice



Dans votre contrôleur todo créer les méthodes

- permettant de récupérer un todo via son id.
- permettant de supprimer un todo via son id.
- permettant de modifier un todo.

Contrôleur Asynchrone

- Nest supporte les fonctions asynchrone
- Lorsque vous retourner une Promesse, Nest au niveau du handler la gère automatiquement.
- Les routes vont automatiquement transformer ca en un observable et s'y inscrire.
- Dès la complétion, le handler récupère la dernière valeur et la renvoi.

Qu'est ce que la programmation réactive

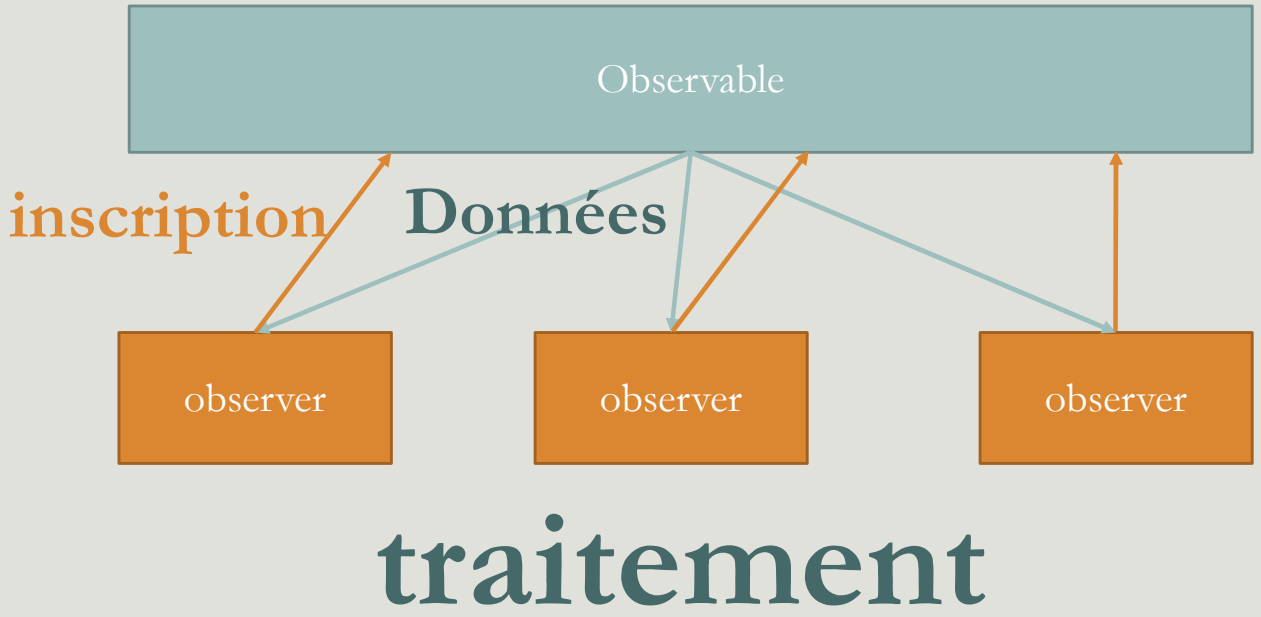
1. Nouvelle manière d'appréhender les appels asynchrones
2. Programmation avec des flux de données asynchrones

Programmation reactive =
Flux de données (observable) + écouteurs d'événements(observer).

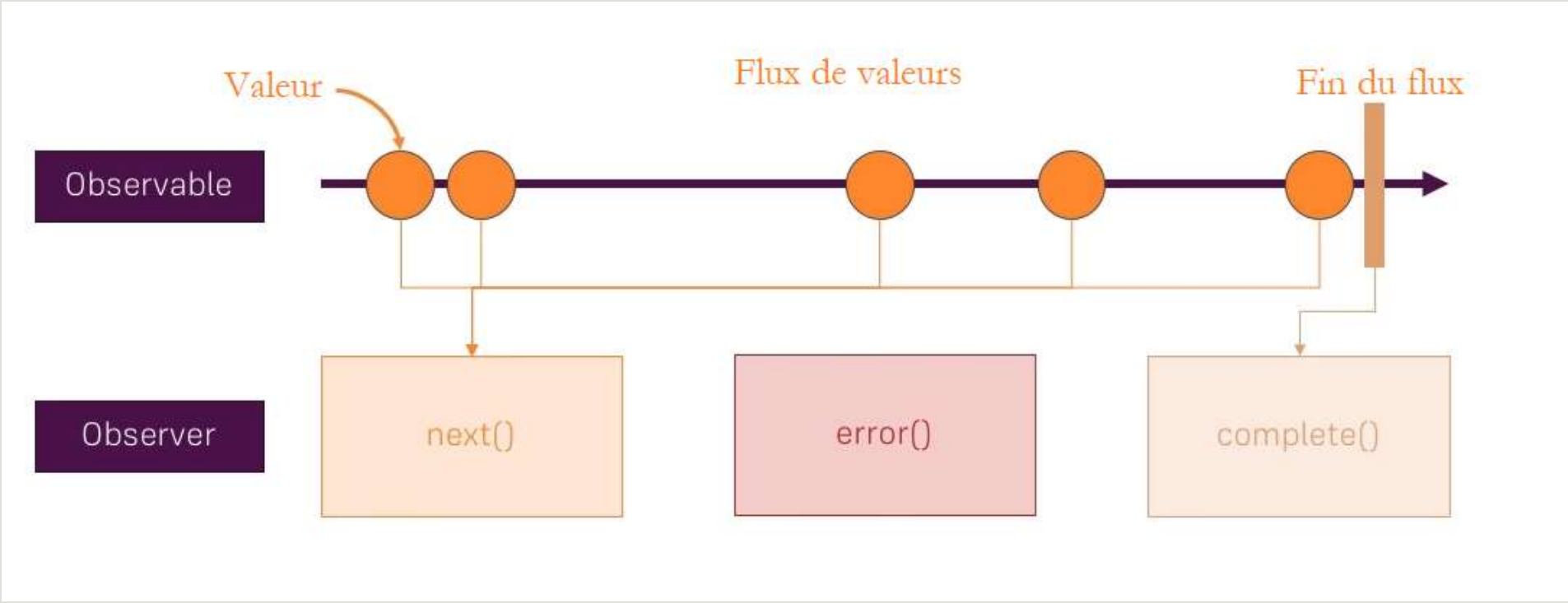
Le pattern « Observer »

- Le patron de conception **observer** permet à un objet de garder la trace d'autres objets, intéressés par l'état de ce dernier.
- **Il définit une relation entre objets de type un-à-plusieurs.**
- Lorsque l'état de cet objet **change**, il **notifie** ces **observateurs**.

Observables, Observers et subscriptions



Fonctionnement



Promesse Vs Observable

Promesse	Observable
Un promesse gère un seul événement	Un observable gère un « flux » d'événements.
Non annulable.	Annulable.
Traitement immédiat.	Lazy : le traitement n'est déclenché qu'à la première utilisation du résultat.
Deux méthodes uniquement (then/catch).	Une centaine d'opérateurs de transformation natifs (map, reduce, merge, filter, …).
	Operateurs tels que retry, replay

Observable

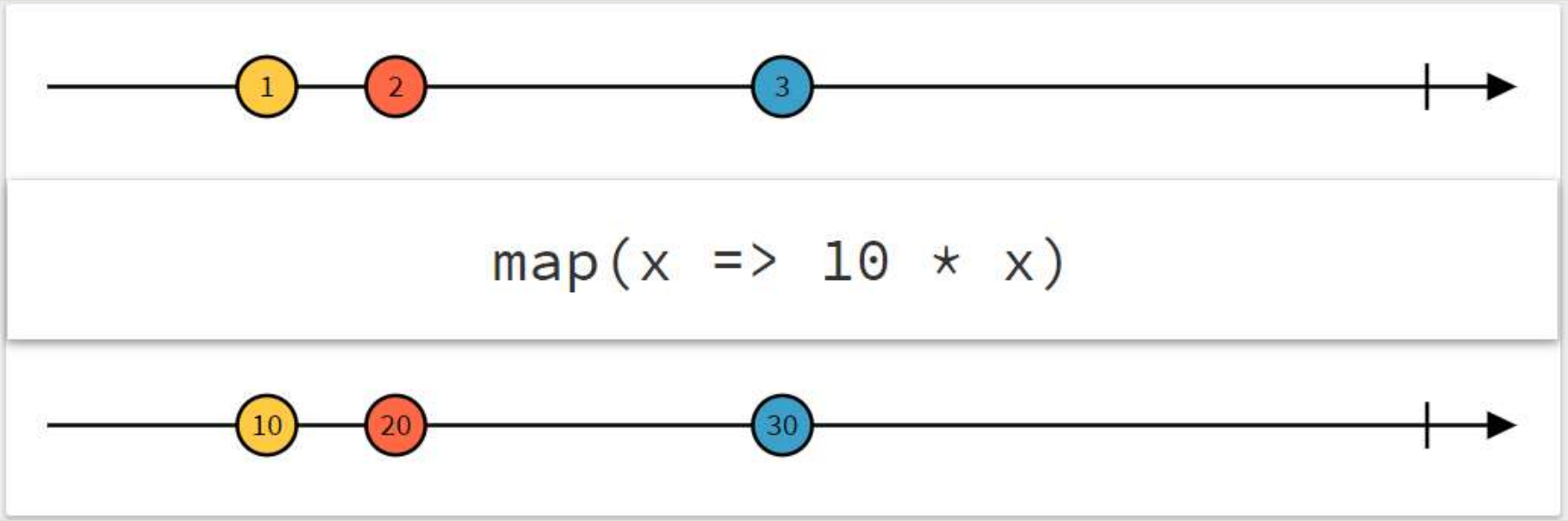
```
const observable = new Observable(
  (observer) => {
    let i = 5;
    setInterval(() => {
      if (!i) {
        observer.complete();
      }
      observer.next(i--);
    }, 1000);
  });
observable.subscribe(
  (val) => {
    console.log(val);
  }
);
```

Les operateurs de l'observable

- Les opérateurs sont des fonctions. Il y a deux types d'opérateurs :
- **Un opérateur pipeable** est une fonction qui prend un observable comme entrée et renvoie un autre observable. C'est une opération pure : le précédent Observable reste inchangé.
  - Syntaxe : monObservable.pipe(opertaeur1(), operateur2(), …).
- **Les opérateurs de création** sont l'autre type d'opérateur, qui peut être appelé comme fonctions autonomes pour créer un nouvel Observable. Par exemple : of(1, 2, 3) crée un observable qui va émettre 1, 2, et 3, l'un après l'autre.

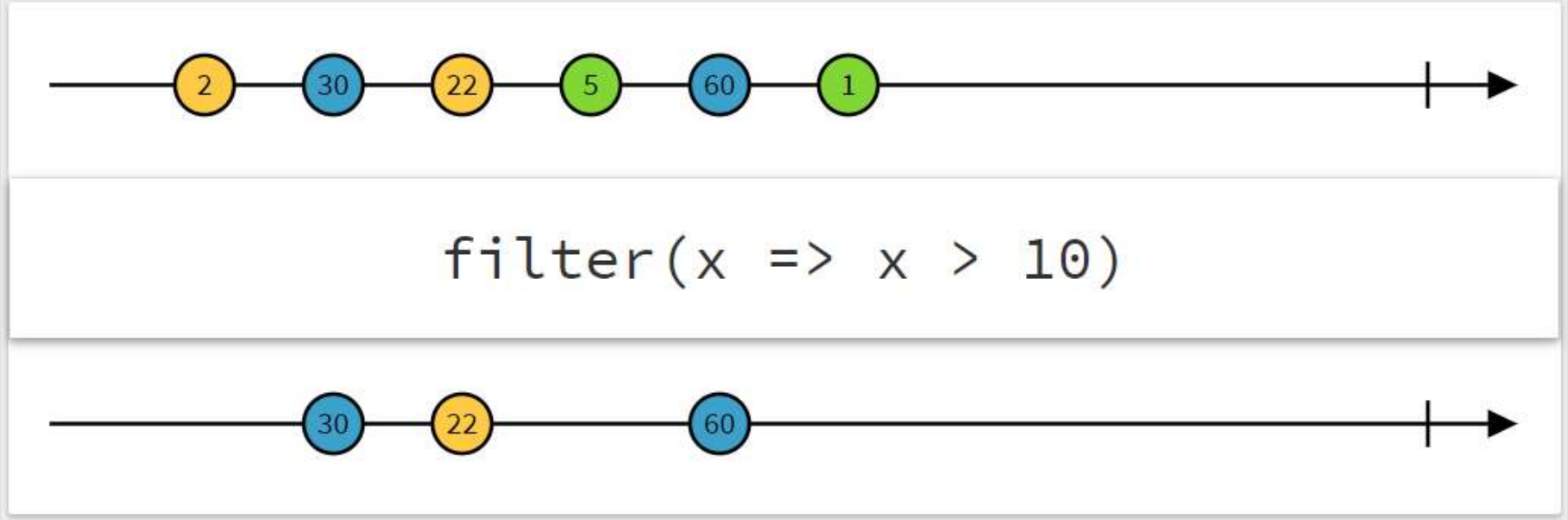
Quelques opérateurs utiles de l'Observable

map

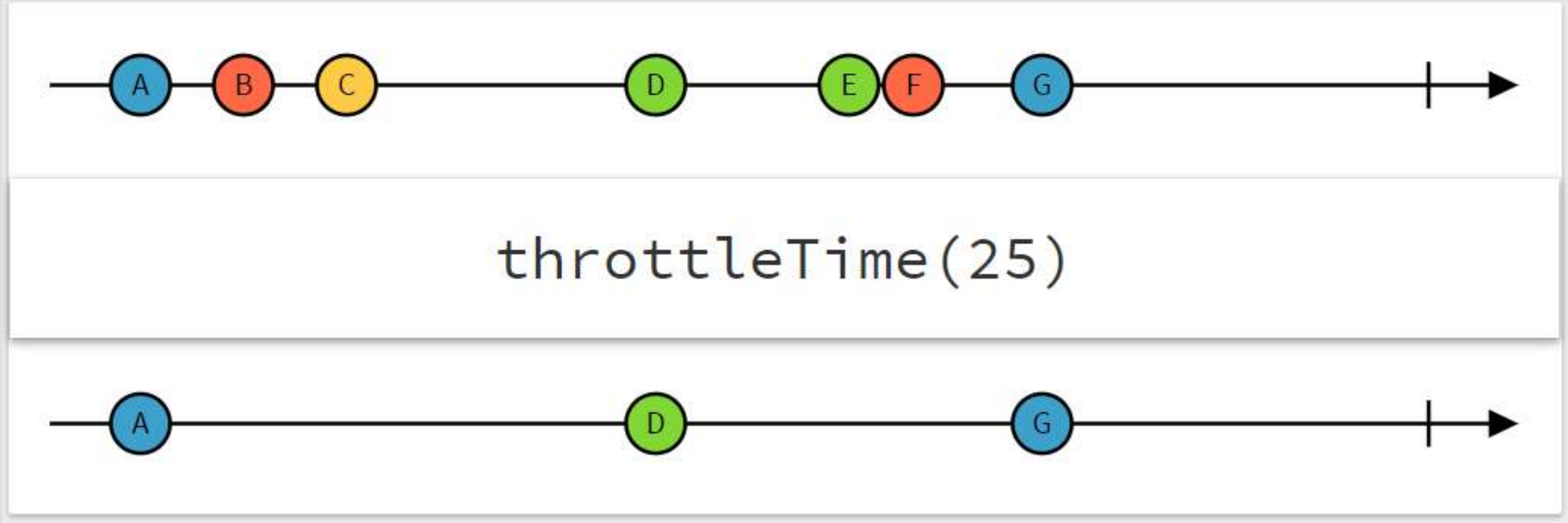


Quelques opérateurs utiles de l'Observable

filter



Quelques opérateurs utiles de l'Observable



Quelques opérateurs utiles de l'Observable

https://angular.io/guide/rx-library

http://reactivex.io/rxjs/manual/overview.html#operators

http://rxmarbles.com/

DTO : Data Transfert Object

- Etant donnée que Typescript est typé et afin d'ajouter de la robustesse à vos API, Nest préconise l'utilisation des **DTO**.

- DTO est un objet qui permet d'encapsuler les données qui sont envoyées via les réseau d'une application à une autre.

- Elles permettent ainsi de définir le modèle de transfert de données entre deux systèmes en l'encapsulant dans le DTO.

DTO : Data Transfert Object

- Elles facilitent la validation des données

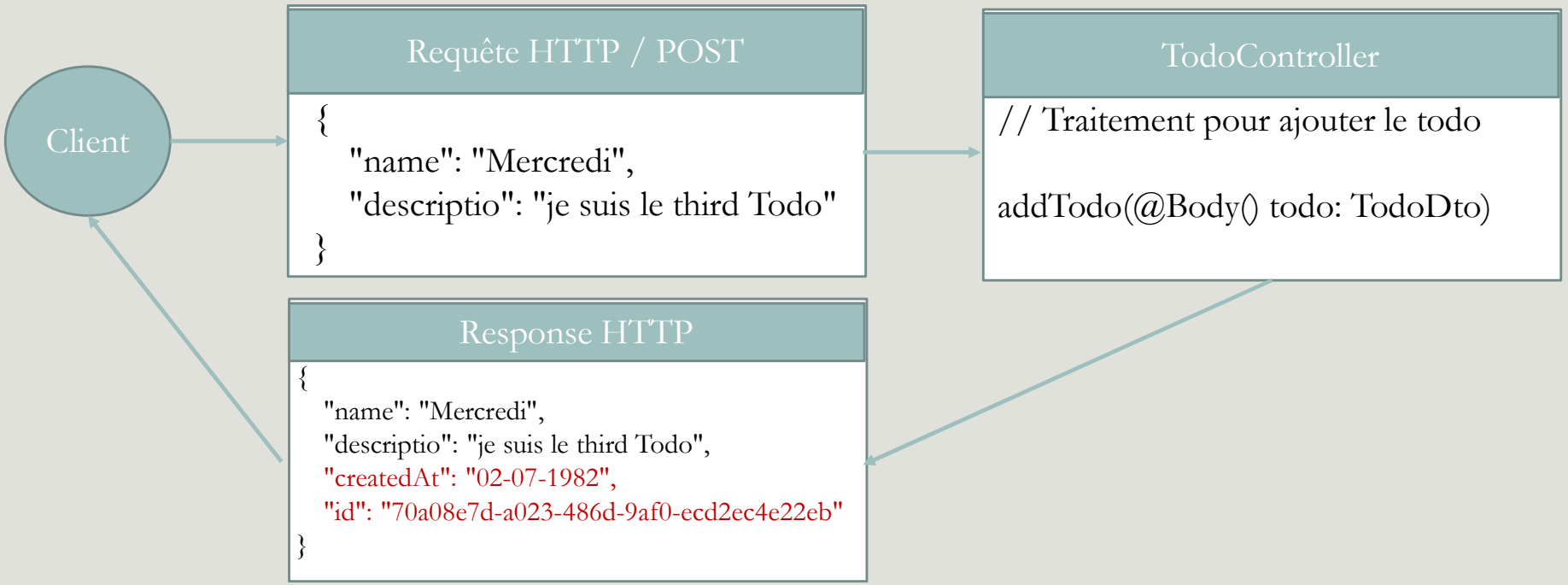
- Elles peuvent être définies en utilisant des classes ou des interfaces, mais Nest **recommande l'utilisation des classes** vu que TypeScript ne sauvegarde pas les metadata pour les generics et les interfaces ce qui peut provoquer un disfonctionnement lors de leur validation (https://docs.nestjs.com/techniques/validation).

- Les DTO **ne sont pas les modèles**, dans plusieurs cas le modèle et les données que vous souhaitez recevoir sont différents.

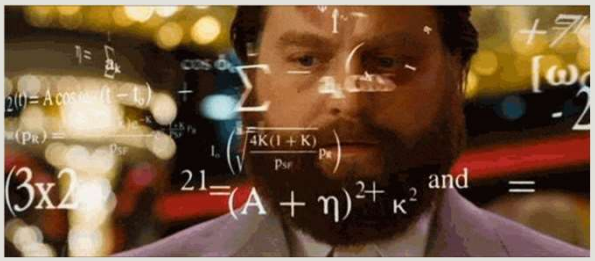
DTO
Exemple

```
export class TodoDto {
  name: string;
  description: string;
}
```

DTO


Client
Requête HTTP / POST
{
"name": "Mercredi",
"descriptio": "je suis le third Todo"
}
TodoController
// Traitement pour ajouter le todo
addTodo(@Body() todo: TodoDto)
Response HTTP
{
"name": "Mercredi",
"descriptio": "je suis le third Todo",
"createdAt": "02-07-1982",
"id": "70a08e7d-a023-486d-9af0-ecd2ec4e22eb"
}

Exercice



- Ajouter le DTO nécessaire pour ajouter un TODO.
- Modifier votre code en utilisant le DTO.
- Créer un autre DTO pour la mise à jour.

Les providers

- Les **providers** sont un concept fondamental de Nest.
- La plupart des classes Nest de base peuvent être traitées comme un provider (les services, les repositories, les fabriques(Factories)).
- L'idée principale d'un provider est qu'il peut injecter des dépendances; cela signifie que les objets peuvent créer diverses relations et dépendances les unes avec les autres, tout en délégant l'instanciation au système d'exécution Nest.
- Pour faire simple, en général, un provider est simplement une classe annotée avec un décorateur **@Injectable()**.
- Il fournit des fonctionnalités et il est **injectable**. On peut **y injecter d'autres providers.**

Les providers
Les services

- Le contrôleur a pour rôle d'intercepter les requêtes des clients puis de dispatcher les différentes tâches liées à cette requête aux différentes composantes de l'application.
- La couche qui doit gérer l'aspect **métier** est la **couche Service**. C'est un provider qui se charge de l'aspect métier.
- Pour créer un service, vous pouvez simplement créer une **classe** et l'annoter avec **@Injectable()** ou utiliser le Cli via la commande **nest generate service nomService.**
- Vous pouvez aussi utiliser le raccourci **nest g s nomService**

Qu'est ce qu'un service ?

- Un service est une classe qui permet d'exécuter un traitement.

- Permet d'encapsuler des fonctionnalités redondantes permettant ainsi d'éviter la redondance de code.

```
Controller 1
f(){};
g(){};
k(){};
```

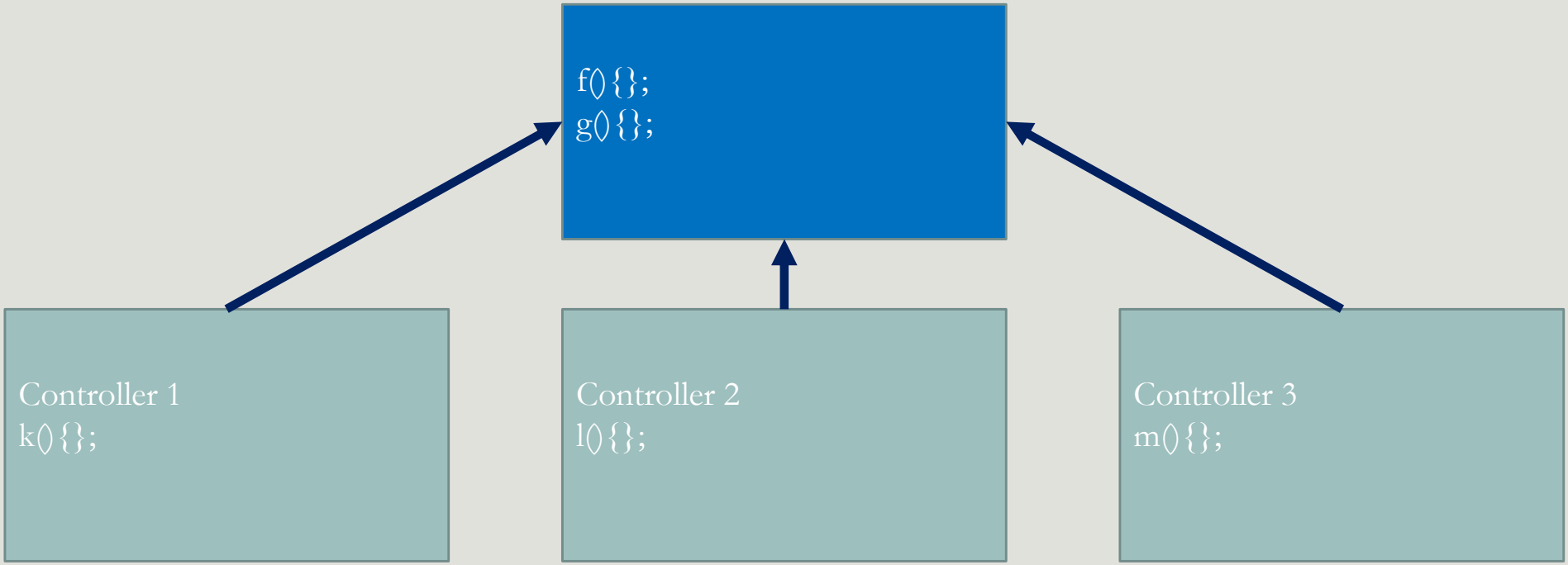
```
Controller 2
f(){};
g(){};
l(){};
```

```
Controller 3
f(){};
g(){};
m(){};
```

Redondance de code

Maintenabilité difficile

Qu'est ce qu'un service ?



Premier Service

```
import { Injectable} from '@nestjs/common';
import { Produit } from './produit.model';

@Injectable()
export class ProduitService {

}
```

```
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';

@Module({
  imports: [ProduitModule],
  controllers: [AppController],
  providers: [AppService],
})
export class AppModule {}
```

- Ici on est entrain d'enregistrer le provider dans le **conteneur IoC de NestJs**.

Premier Service

```ts
import { Injectable} from '@nestjs/common';
import { Produit } from './produit.model';

@Injectable()
export class ProduitService {

}
```

```ts
import { Module } from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
import { ProduitModule } from './produit/produit.module';

@Module({
 imports: [ProduitModule],
 controllers: [AppController]
 providers: [
  {
    provide: AppService,
    useClass: AppService,
  }
],
})
export class AppModule {}
```

Le TOKEN

Injection de dépendance (DI)

- L'injection de dépendance est un patron de conception.

```
Classe A1{
 ClasseB b;
 ClasseC c;
 …
}
```

```
Classe A2{
 ClasseB b;
…
}
```

```
Classe A3{
ClasseC c;
 …
}
```

Que se passera t-il si on change quelque chose dans le constructeur de B ou C ?

Qui va modifier l'instanciation de ces classes dans les différentes classes qui en dépendent?

Injection de dépendance (DI)

- **Déléguer cette tache à une entité tierce.**

```
Classe A1{
 Constructor(B b, C c)
 …
}
```

```
Classe A2{
Constructor(B b)
…
}
```

```
Classe A3{
Constructor(C c)
 …
}
```

NestJS
DI Container

Injection de dépendance (DI)
Le Workflow du **IOC** Container

Au bootstraping de l'application, il enregistre toutes les classes avec le conteneur (IOC Container (NestJS runtime system))

Pour chaque classe, le container va identifier ses dépendances en créant un graphe de dépendances.

Le container va générer toutes les dépendances en se basant sur le graphe de dépendance pour nous créer les instance souhaitées.

Les instances crées vont être sauvegardées et réutilisées en cas de besoin.

@Injectable()

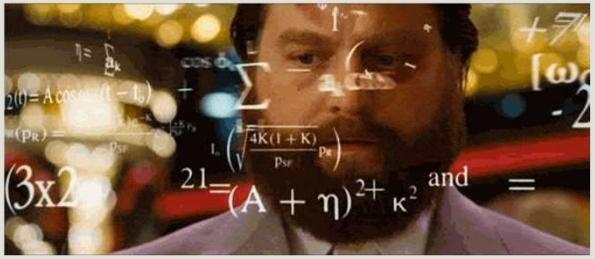
- Afin de **spécifier au NestJs IoC container** qu'il **doit se charger** d'une classe pour **y injecter ses dépendances**, vous devez le **décorer** par **@Injectable**.
- **@Injectable** n'est pas obligatoire pour dire que la classe est un service.
- Par contre **si ce provider a besoin de dépendances**, il doit se déclarer au prêt du **NestJs IoC container** via cet **@Injectable.**

Injection de dépendance (DI)

- Afin d'injecter un service, il suffit de le passer comme paramètre du constructeur de la classe qui en a besoin.
- Le service doit être provide par le module parent ou exporté par l'un des modules importés.

```
import { Body, Controller, Get, Param, Patch, Post } from '@nestjs/common';
import { ProduitService } from './produit.service';
import { Produit } from './produit.model';
@Controller('produit')
export class ProduitController {
                constructor(private produitService: ProduitService) {
}
  @Get('')
  getAllProducts() {
    return this.produitService.getAllProduits();
  }
}
```

Exercice



- Créer un service todoService permettant de centraliser les fonctionnalités liées au todo.

Les providers personnalisés

- Dans certains cas d'utilisation, l'utilisation standard des providers ne convient pas, imaginer l'un des cas suivants :
  - Vous souhaiter créer une instance personnalisée au lieu de laisser le container le faire pour vous.
  - Vous voulez injecter une bibliothèque externe
  - Vous voulez mocker une classe pour le test
  - Vous voulez injecter des instances différentes selon le contexte …
- Nest vous permet de définir des providers particuliers selon votre besoin.

Les providers personnalisés useValue

- La syntaxe **useValue** est utile pour injecter
  - Une valeur constante,
  - Une bibliothèque externe
  - Remplacer une implémentation réelle par un objet fictif.

```
providers: [
  {
    useValue: [{ lundi: 'nestJs' }, { mardi: 'Still NestJs' }],
    provide: 'TODOS_LIST',
  },
  TodoService,
 ],
```

Les providers personnalisés useValue

- Si vous injecter une classe, l'utilisation de l'injection via le constructeur reste d'actualité.
- Sinon, pour injecter ce provider utiliser la syntaxe **@Inject, qui prend en paramètre le Token.**

```
providers: [
  {
    provide: 'TODOS_LIST',
    useValue: [{ lundi: 'nestJs' }, { mardi: 'Still NestJs' }],
  },
  TodoService,
],
```

```
  constructor(
    @Inject('TODOS_LIST') todoList,
  ) {
      console.log('Fake Todo List', todoList);
  }
```

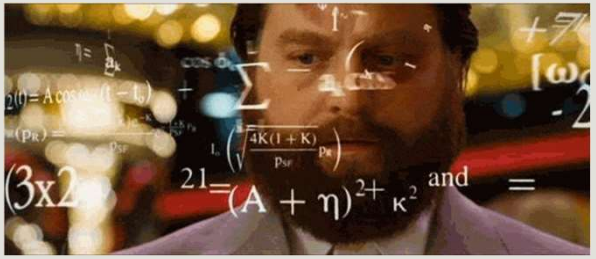
```
@Controller('todo')
export class TodoController {
  @Inject('TODOS_LIST') todoList;
  constructor() {}
```

Les providers personnalisés export

- Vous pouvez exporter les provider personnalisés en utilisant
  - le token
  - l'objet du provider.

```
const TodoListProvider = {
    provide: 'TODOS_LIST',
    useValue: [{ lundi: 'nestJs' }, { mardi: 'Still NestJs' }],
  },

@Module({
 providers: [
  TodoListProvider
  FirstService,
 ],
 exports: [
  TodoListProvider
],
})
```

Exercice



- Créer un module CommonModule
- Ce module va être souvent utilisé pensez à ca en le créant
- Provider la fonction uuid
- Faite en sorte de l'utiliser dans le TodoService

Les providers personnalisés useClass

- La syntaxe **useClass** est utile pour injecter dynamiquement une classe.
- Imaginez que le service de mailing ou de PDF dépend de l'environnement de développement.

```
@Module({
 providers: [
   {
       provide: PdfService,
       useClass:
            process.env.NODE_ENV === 'development'
                 ? DevelopmentPdfService
                 : ProductionPdfService,
   }
 ],
}) export class AppModule {}
```

Les providers personnalisés useFactory

- La syntaxe useFactory (comme son nom l'indique) permet aussi de créer dynamiquement des providers à travers une usine de provider.
- Le provider utilisé sera fourni par la valeur de retour envoyé par le factory.

```
const ExampleFactoryProvider = {
  provide: 'FACTORY_EXAMPLE_PROVIDER',
  useFactory: () => {
    // provide something
  },
};
```

Les providers personnalisés useFactory

- La fonction du factory peut recevoir des paramètres.
- La **clé inject** peut vous permettre d'injecter des providers en acceptant un tableau de providers que Nest va gérer et les passer à la fonction du factory pendant le processus d'instanciation.
- Les providers peuvent être marqués comme **optionnels**.
- L'**ordre d'apparition** des providers dans **inject doit être le même** que dans la **fonction de votre factory**.

Les providers personnalisés useFactory

```
const UuidProvider = {
  provide: 'UUID',
  useValue: uuid,
};
const RandomProvider = {
  provide: 'RANDOM',
  useFactory: (uuid) => {
    if (new Date().getMilliseconds() % 2 == 0) {
      return uuid;
    }
    return undefined;
  },
  inject: ['UUID'],
};
```

```
class FirstService {
  constructor(private uniqueId: string = ' default') {}
  sayHello() {
    console.log('hello :)' + this.uniqueId);
  }
}
const ExampleFactoryProvider = {
  provide: 'FACTORY_EXAMPLE_PROVIDER',
  useFactory: (randomProvider?) => {
    if (randomProvider) {
      return new FirstService(randomProvider());
    }
    return new FirstService();
    //provide something
  },
  inject: [{ token: 'RANDOM', optional: true }],
};
```

https://docs.nestjs.com/fundamentals/custom-providers#factory-providers-usefactory

Les providers personnalisés useFactory

- Imaginez que vous voulez **providez** quelque chose qui **dépend d'une opération Asynchrone**. Exemple vous ne voulez lancez votre application que si votre connexion à la base de données est établie.
- Afin de gérer ca, vous pouvez **utiliser les providers asynchrones**.
- Les providers asynchrones sont injectés de la même manière.

```
{
  provide: 'ASYNC_CONNECTION',
  useFactory: async () => {
    const connection = await createConnection(options);
    return connection;
  },
}
```

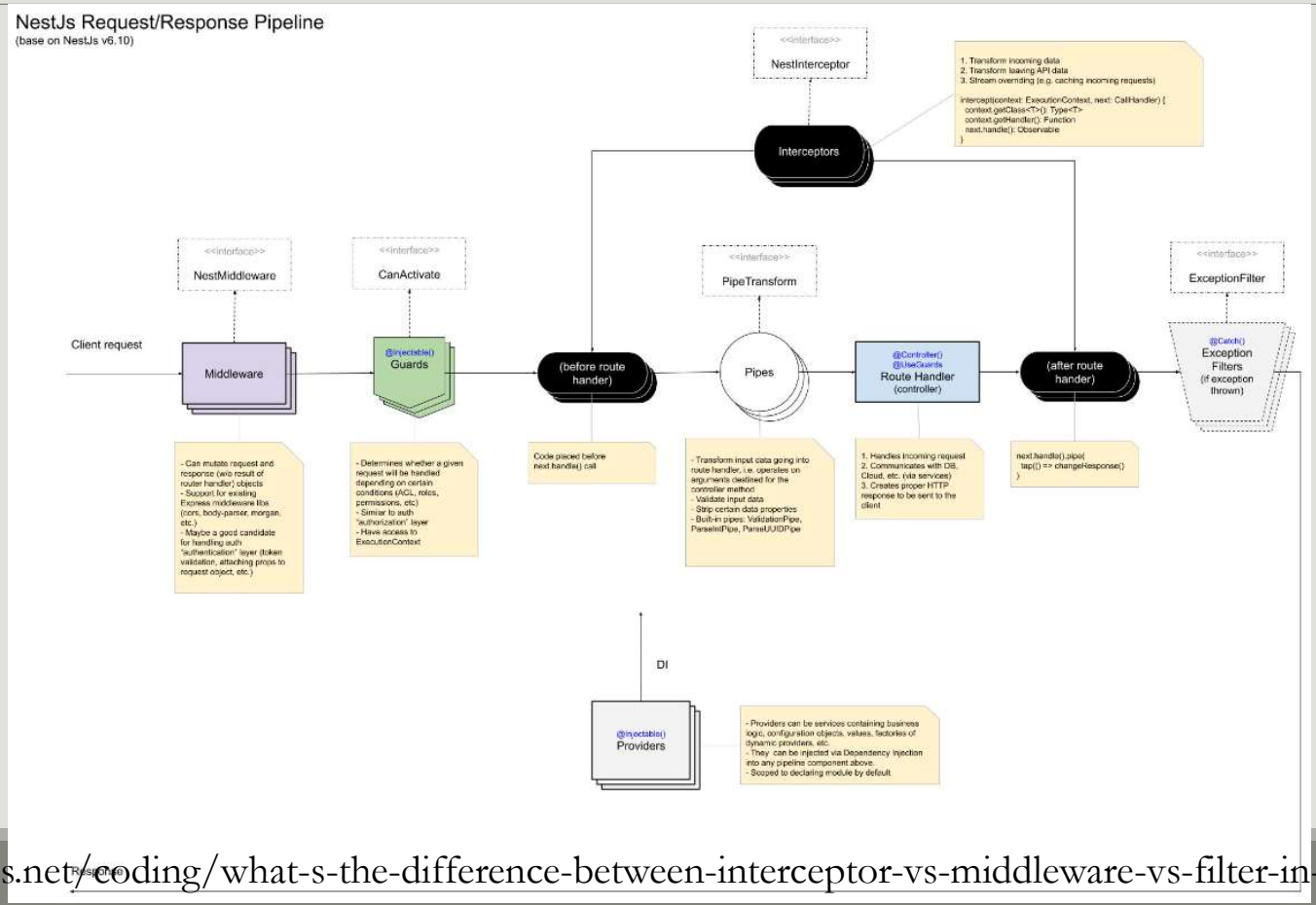
```
@Inject('ASYNC_CONNECTION')
```

Request lifecycle

Une requête passe par les couches suivantes avant d'atteindre le contrôleur qui va la traiter :

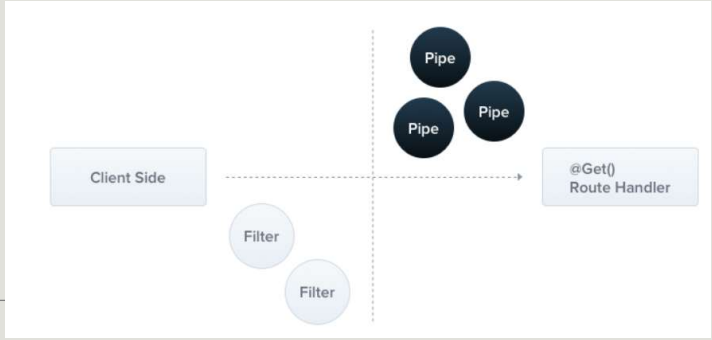
1) middlewares,
2) guards,
3) interceptors,
4) pipes
5) Pour retourner finalement aux interceptors lorsque la réponse est générée.

Request lifecycle



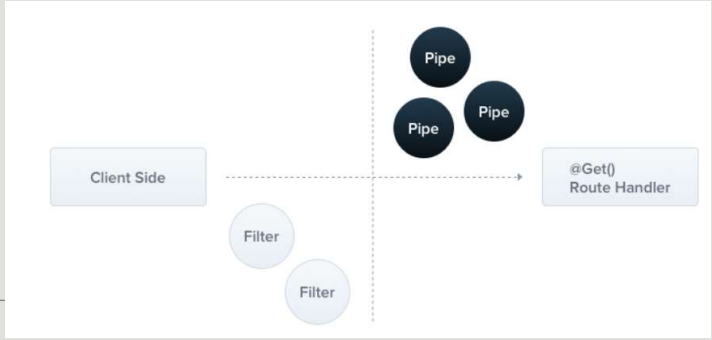
https://askcodes.net/coding/what-s-the-difference-between-interceptor-vs-middleware-vs-filter-in-nest-js-

Pipes
Définition et fonctionnement



- Un pipe est une **classe** qui a deux principales fonctionnalités.
- **transformation** : transformez les données d'entrée sous la forme souhaitée (par exemple, de chaîne en entier).
- **validation** : évaluez les données d'entrée et, si elles sont valides, passez-les simplement telles quelles; sinon, lever une exception lorsque les données sont incorrectes.
- Dans les deux cas, les **pipes fonctionnent sur les arguments gérés par l'action du contrôleur**.

Pipes
Définition et fonctionnement



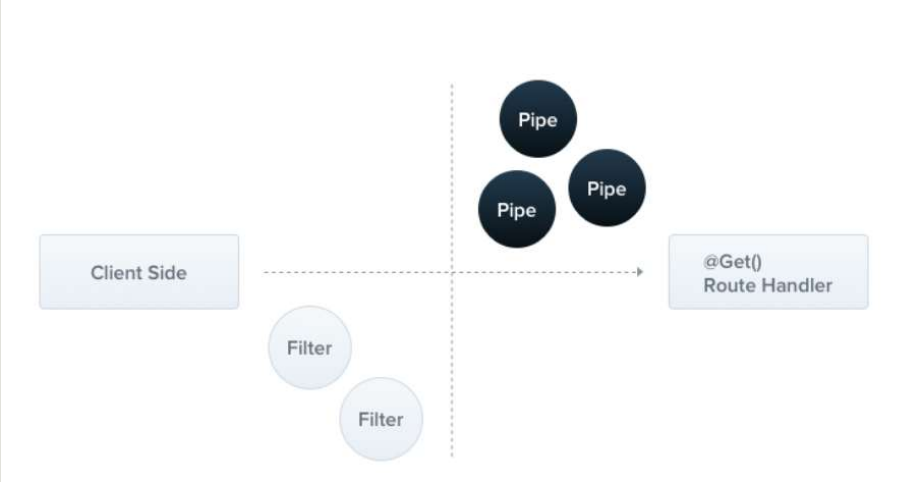
- Nest appelle le pipe juste avant l'invocation d'une méthode (handler).

- Le pipe reçoit les arguments destinés à la méthode et les exploite.

- Toute opération de transformation ou de validation a lieu à ce moment. Ensuite, l'action est appelée avec tous les arguments passés ou non par le pipe.

Pipes

- Nest est livré avec un certain nombre de pipe prêt à l'emploi.
- Vous pouvez aussi créer vos propres pipes.
- Les pipes offerts par Nest (elles sont dans le package @nestjs/common) sont :
  - ValidationPipe
  - ParseIntPipe
  - ParseBoolPipe
  - ParseArrayPipe
  - ParseUUIDPipe
  - DefaultValuePipe



Pipes
Utilisation

- Afin d'utiliser un pipe, on a besoin de **l'associer à la propriété qu'il doit 'piper'** et dans le contexte dans lequel vous voulez l'exécuter (Body, Param, Quey).
- Si vous voulez par exemple transformer un paramètre de votre requête en un entier avant l'exécution de l'action, vous devez le faire lors de la récupération de ce paramètre.

```
@Patch(':id')
updateProduct(
  @Param('id', ParseIntPipe) id: number,
  @Body() newProduct: ProductEditDto): Produit {
  return this.produitService.updateProduit(id, newProduct);
}
```

Pipes
Utilisation

- Dans le premier exemple, nous avons passé la classe, laissons l'instanciation du pipe à Nest. Ceci permet **d'activer l'injection de dépendance.**
- Dans le cas ou vous voulez **passer un paramètre à votre pipe**, vous devez **l'instancier** et lui passer les paramètres nécessaires.

```
@Patch(':id')
updateProduct(
  @Param('id', new ParseIntPipe({errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE})) id: number,
  @Body() newProduct: ProductEditDto
): Produit {
  return this.produitService.updateProduit(id, newProduct);
}
```

Pipe DefaultValuePipe

- Lorsque vous avez un **paramètre optionnel(?)** et que vous voulez lui associer une valeur par défaut, vous pouvez utiliser le pipe **DefaultValuePipe**.
- Le pipe **DefaultValuePipe** vous permet d'affecter une valeur par défaut.

```
@Patch(':id?')
updateProduct(
  @Param('id', new DefaultValuePipe(0) ,
        new ParseIntPipe({errorHttpStatusCode: HttpStatus.NOT_ACCEPTABLE})) id: number,
  @Body() newProduct: ProductEditDto
): Produit {
  return this.produitService.updateProduit(id, newProduct);
}
```

Pipes
Validation et Transformation Pipe

- Les **validationPipe ('@nestjs/common')** sont des pipes qui permettent de **valider vos données.**
- Ils utilisent le package **class-validator** et **class-transformer** (https://github.com/typestack/class-validator, https://www.npmjs.com/package/class-transformer) et l'ensemble de ces décorateurs permettant de valider et de transformer différents types .
- Ils permettent, à travers les décorateurs, de valider les données entrantes représentées par des classes ou des DTO.
- Afin de les installer, utiliser la commande suivante :

npm i --save class-validator class-transformer

Pipes
Validation et Transformation Pipe

- Afin **d'activer la validation**, vous pouvez le faire **globalement**, au niveau de la fonction **bootstrap,** dans 'main.ts', en utilisant la méthode **useGlobalPipes** de votre app.

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

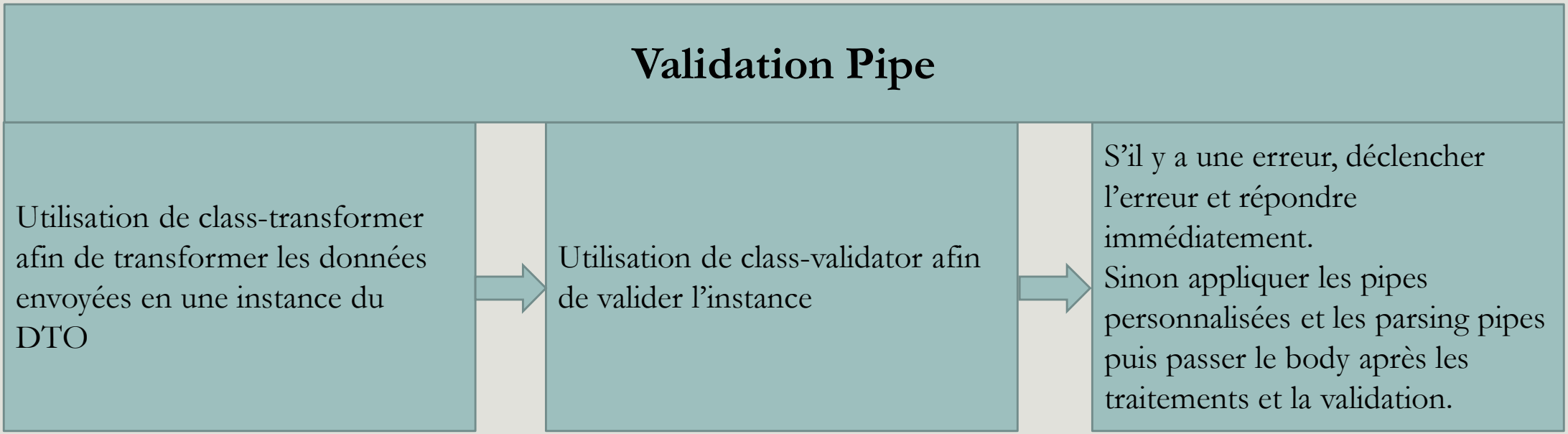
- De cette façon, **toutes les classes et tous les DTO** qui sont **annotés** avec des **validateurs** seront **validés automatiquement**.

Pipes
Validation et Transformation Pipe

- Si vous voulez uniquement **valider des routes particulières**, utilisez le décorateur **@UsePipes(PipeClass1, PipeClass2,…)**

```
@Get('/:id')
@UsePipes(ValidationPipe)
getTaskById(
  @Param('id', ParseIntPipe) id: number,
    user: User
  ): Promise<Task> {
  return this.tasksService.getTaskById(id, user);
}
```

Pipes
Validation et Transformation Pipe
Flux d'application sur le Body



Pipes
Transformation des objets

- Lorsque vous recevez vos requêtes, le corps de ces requêtes (payload) est un objet **Js standard**.
- Les décorateurs de ValidationPipe peuvent automatiquement transformer vos **payload d'un objet générique *object* vers une instance de votre DTO**.
- Afin d'activer cette transformation automatique, passer à votre ValidationPipe un **objet d'option** contenant la propriété **transform** à **true**.
- Ceci permettra d'activer la **transformation des types primitives**.
- Utilisons **instanceOf** pour vérifier ca.

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

Pipes
Les options de classValidator

Option	Type	Description
whitelist	boolean	Si elle est à true, elle n'acceptera que les propriétés définies dans le DTO. Toutes les autres propriétés seront ignorées**. Ceci n'est valide que si vous @nnoter vos propriétés**. Une propriété non @nnotée sera ignorée.
forbidNonWhitelisted	boolean	Si elle est à true, si elle détecte une propriété non définie dans votre DTO elle déclenche une erreur. Elle **doit** être associé à la propriété whitelist
disableErrorMessages	boolean	Si elle est à true, les erreurs de validations ne seront pas envoyées au client.

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

https://docs.nestjs.com/techniques/validation

Pipes
Validation Pipe
Utilisation

- Afin d'utiliser un validationPipe, vous devez **@nnoter ou Décorer** la **propriété cible** au niveau de votre Model ou DTO avec le décorateur correspondant à la validation que vous souhaitez.

```
import { IsNotEmpty, MinLength, ValidationArguments } from 'class-validator';
import ErrorMessages from './../errorMessages';
export class CreateTaskDto {
  @IsNotEmpty()
  @MinLength(20)
  title: string;
  @IsNotEmpty({
    message: ErrorMessages.isEmpty
  })
  description: string;
}
```

Pipes
Validation Pipe
Validation Decorator

- Les décorateurs fournis par class-validator et que vous pouvez utiliser pour valider vos classes et DTO sont nombreux et permettent de valider plusieurs types.

Decorator	Description
Common validation decorators	
@IsDefined(value: any)	Vérifie si la valeur est définie (! == indéfini,! == null). C'est le seul décorateur qui ignore l'option skipMissingProperties
@IsOptional()	Vérifie si la valeur donnée est vide(=== null, === undefined) et si c'est le cas, ignore tous les validateurs de la propriété.
@Equals(comparison: any)	Vérifie si la valeur est égale à la comparaison ("===").
@NotEquals(comparison: any)	Vérifie si la valeur n'est pas égale ("! ==") la comparaison.
@IsEmpty()	Vérifie si la valeur donnée est vide(=== '', null, undefined).
@IsNotEmpty()	Vérifie si la valeur donnée n'est pas vide(!== '', !== null, !== undefined).
@IsIn(values: any[])	Vérifie si la valeur appartient au tableau passé en paramètre.
@IsNotIn(values: any[])	Vérifie si la valeur n'appartient pas au tableau passé en paramètre.

https://github.com/typestack/class-validator#validation-decorators

Pipes
Validation Pipe
Validation Decorator

- Les décorateurs fournis par class-validator et que vous pouvez utiliser pour valider vos classes et DTO sont nombreux et permettent de valider plusieurs types.

Type validation decorators	
@IsBoolean()	Checks if a value is a boolean.
@IsDate()	Checks if the value is a date.
@IsString()	Checks if the string is a string.
@IsNumber(options: IsNumberOptions)	Checks if the value is a number.
@IsInt()	Checks if the value is an integer number.
@IsArray()	Checks if the value is an array
@IsEnum(entity: object)	Checks if the value is an valid enum

https://github.com/typestack/class-validator#validation-decorators

Pipes
Validation Pipe
Validation Decorator

- Les décorateurs fournis par class-validator et que vous pouvez utiliser pour valider vos classes et DTO sont nombreux et permettent de valider plusieurs types.

Number validation decorators	
@IsDivisibleBy(num: number)	Checks if the value is a number that's divisible by another.
@IsPositive()	Checks if the value is a positive number greater than zero.
@IsNegative()	Checks if the value is a negative number smaller than zero.
@Min(min: number)	Checks if the given number is greater than or equal to given number.
@Max(max: number)	Checks if the given number is less than or equal to given number.

Date validation decorators	
@MinDate(date: Date)	Checks if the value is a date that's after the specified date.
@MaxDate(date: Date)	Checks if the value is a date that's before the specified date.

https://github.com/typestack/class-validator#validation-decorators

Pipes
Validation Pipe
Validation Decorator

String validation decorators	
@Contains(seed: string)	Checks if the string contains the seed.
@NotContains(seed: string)	Checks if the string not contains the seed.
@IsAlpha()	Checks if the string contains only letters (a-zA-Z).
@IsAlphanumeric()	Checks if the string contains only letters and numbers.
@IsDecimal(options?: IsDecimalOptions)	Checks if the string is a valid decimal value. Default IsDecimalOptions are force_decimal=False, decimal_digits: '1,', locale: 'en-US',
@IsAscii()	Checks if the string contains ASCII chars only.
@IsBase32()	Checks if a string is base32 encoded.
@IsBase64()	Checks if a string is base64 encoded.
@IsIBAN()	Checks if a string is a IBAN (International Bank Account Number).
@IsBIC()	Checks if a string is a BIC (Bank Identification Code) or SWIFT code.
@IsByteLength(min: number, max?: number)	Checks if the string's length (in bytes) falls in a range.
@IsCreditCard()	Checks if the string is a credit card.
@IsCurrency(options?: IsCurrencyOptions)	Checks if the string is a valid currency amount.

Pipes
Validation Pipe
Validation Decorator

@IsEthereumAddress()	Checks if the string is an Ethereum address using basic regex. Does not validate address checksums.
@IsBtcAddress()	Checks if the string is a valid BTC address.
@IsDataURI()	Checks if the string is a data uri format.
@IsEmail(options?: IsEmailOptions)	Checks if the string is an email.
@IsFQDN(options?: IsFQDNOptions)	Checks if the string is a fully qualified domain name (e.g. domain.com).
@IsFullWidth()	Checks if the string contains any full-width chars.
@IsHalfWidth()	Checks if the string contains any half-width chars.
@IsVariableWidth()	Checks if the string contains a mixture of full and half-width chars.
@IsHexColor()	Checks if the string is a hexadecimal color.
@IsHSLColor()	Checks if the string is an HSL (hue, saturation, lightness, optional alpha) color based on CSS Colors Level 4 specification.
@IsRgbColor(options?: IsRgbOptions)	Checks if the string is a rgb or rgba color.
@IsIdentityCard(locale?: string)	Checks if the string is a valid identity card code.
@IsPassportNumber(countryCode?: string)	Checks if the string is a valid passport number relative to a specific country code.
@IsPostalCode(locale?: string)	Checks if the string is a postal code.

Pipes
Validation Pipe
Validation Decorator

@IsHexadecimal()	Checks if the string is a hexadecimal number.
@IsOctal()	Checks if the string is a octal number.
@IsMACAddress(options?: IsMACAddressOptions)	Checks if the string is a MAC Address.
@IsIP(version?: "4"|"6")	Checks if the string is an IP (version 4 or 6).
@IsPort()	Check if the string is a valid port number.
@IsISBN(version?: "10"|"13")	Checks if the string is an ISBN (version 10 or 13).
@IsEAN()	Checks if the string is an if the string is an EAN (European Article Number).
@IsISIN()	Checks if the string is an ISIN (stock/security identifier).
@IsISO8601(options?: IsISO8601Options)	Checks if the string is a valid ISO 8601 date. Use the option strict = true for additional checks for a valid date, e.g. invalidates dates like 2019-02-29.
@IsJSON()	Checks if the string is valid JSON.
@IsJWT()	Checks if the string is valid JWT.
@IsObject()	Checks if the object is valid Object (null, functions, arrays will return false).
@IsNotEmptyObject()	Checks if the object is not empty.
@IsLowercase()	Checks if the string is lowercase.

Pipes
Validation Pipe
Validation Decorator

@IsLatLong()	Checks if the string is a valid latitude-longitude coordinate in the format lat,long
@IsLatitude()	Checks if the string or number is a valid latitude coordinate
@IsLongitude()	Checks if the string or number is a valid longitude coordinate
@IsMobilePhone(locale: string)	Checks if the string is a mobile phone number.
@IsLocale()	Checks if the string is a locale.
@IsPhoneNumber(region: string)	Checks if the string is a valid phone number. "region" accepts 2 characters uppercase country code (e.g. DE, US, CH).If users must enter the intl. prefix (e.g. +41), then you may pass "ZZ" or null as region. See google-libphonenumber, metadata.js:countryCodeToRegionCodeMap on github
@IsMongoId()	Checks if the string is a valid hex-encoded representation of a MongoDB ObjectId.
@IsNumberString(options?: IsNumericOptions)	Checks if the string is numeric.
@IsUrl(options?: IsURLOptions)	Checks if the string is an url.
@IsMagnetURI()	Checks if the string is a magnet uri format.
@IsUUID(version?: "3"|"4"|"5"|"all")	Checks if the string is a UUID (version 3, 4, 5 or all).
@IsFirebasePushId()	Checks if the string is a Firebase Push id

Pipes
Validation Pipe
Validation Decorator

@IsUppercase()	Checks if the string is uppercase.
@Length(min: number, max?: number)	Checks if the string's length falls in a range.
@MinLength(min: number)	Checks if the string's length is not less than given number.
@MaxLength(max: number)	Checks if the string's length is not more than given number.
@Matches(pattern: RegExp, modifiers?: string)	Checks if string matches the pattern. Either matches('foo', /foo/i) or matches('foo', 'foo', 'i').
@IsHash(algorithm: string)	Checks if the string is a hash of type algorithm. Algorithm is one of ['md4', 'md5', 'sha1', 'sha256', 'sha384', 'sha512', 'ripemd128', 'ripemd160', 'tiger128', 'tiger160', 'tiger192', 'crc32', 'crc32b']
@IsMimeType()	Checks if the string matches to a valid MIME type format

Pipes
Validation Pipe
Validation Decorator

Array validation decorators	
@ArrayContains(values: any[])	Checks if array contains all values from the given array of values.
@ArrayNotContains(values: any[])	Checks if array does not contain any of the given values.
@ArrayNotEmpty()	Checks if given array is not empty.
@ArrayMinSize(min: number)	Checks if array's length is as minimal this number.
@ArrayMaxSize(max: number)	Checks if array's length is as maximal this number.
@ArrayUnique()	Checks if all array's values are unique. Comparison for objects is reference-based.
Object validation decorators	
@IsInstance(value: any)	Checks if the property is an instance of the passed value.
Other decorators	
@Allow()	Prevent stripping off the property when no other constraint is specified for it.

Pipes
Validation Pipe
Message d'erreur

- Vous pouvez personnaliser votre message d'erreur en passant en paramètre de votre décorateur un objet avec une propriété message.

```
@IsNotEmpty({
  message: "Vous devez spécifier un titre"
})
title: string;
```

Pipes
Validation Pipe
Message d'erreur

- Il existe des propriétés spéciales fournies automatiquement et que vous pouvez utiliser dans votre message :
  - $value : la valeur validée
  - $property : le nom de la propriété de l'objet qui a été validée
  - $target : nom de la classe de l'objet validé
  - $constraint1, $constraint2, ... $constraintN : les contraintes spécifiées par la validation

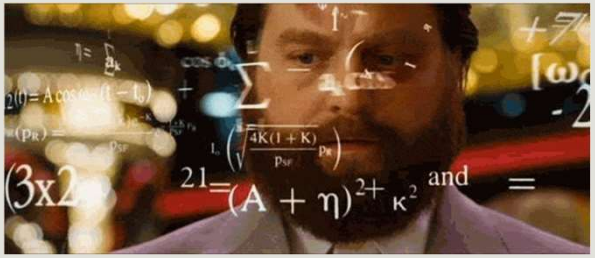
```
@MinLength(20, {
  message: "La taille de votre $property $value est courte, la taille minimale de $property est $constraint1"
})
title: string;
```

Pipes
Validation Pipe

- Vous pouvez passer à votre message une **fonction** qui prend en paramètre un objet de type **ValidationArguments**.
- Cet objet contiendra les informations sur votre validation, à savoir, value, property, target et constraints.

```
@MinLength(20, {
  message: (validationData: ValidationArguments) => {
    return `La taille de votre ${validationData.property}        ${validationData.value} est courte,
      la taille minimale de ${validationData.property} est       ${validationData.constraints[0]}`
  }
})
title: string;
```

Exercices



1- Appliquez les contraintes suivantes pour l'ajout d'un Todo.

- La description doit au moins avoir 10 caractères et elle est obligatoire.
- Le name est obligatoire et doit avoir une taille minimale de 3 caractères et une taille maximale de 15 caractères.
- Créer vos propres messages d'erreurs et centraliser les dans un même fichier pour optimiser la réutilisation et faciliter la maintenance de votre application. Faite en sorte d'avoir une même méthode pour l'affichage des messages de taille (minLength et maxLength) et dont le message s'adapte au champ et à la condition gérée.

2- Créer un DTO propre à la méthode de mise à jour. Aucun champs n'est obligatoire pour l'update. Garder les contraintes de la partie ajout (taille).

- Le statut doit être une des valeurs de vos statuts.

Mapped Type @nestjs/mapped-types

- Généralement, vos DTO sont des variantes de votre entité. Vous pouvez aussi les créer en vous basant sur un des DTO que vous avez déjà.
- Pour faciliter ca, Nest nous fournit un ensemble de fonctionnalités qui crée des transformations facilitant cette tâche.
- Commencez par installer la bibliothèque **@nestjs/mapped-types**
- **PartialType** : Retourne la classe ciblée en mettant tous les champs à Optional.

```
export class UpdateTodoDto extends PartialType(AddTodoDto)
```

Fait en sorte que UpdateTodoDto contiennent tous les champs de AddTodoDto mais optionnels.

Mapped Type @nestjs/mapped-types

- **PickType** vous permet de créer un nouveau type (une classe) en **sélectionnant un ensemble de champ** d'une classe existante.

```
export class UpdateTodoDto extends PickType(AddTodoDto, ['name'])
```

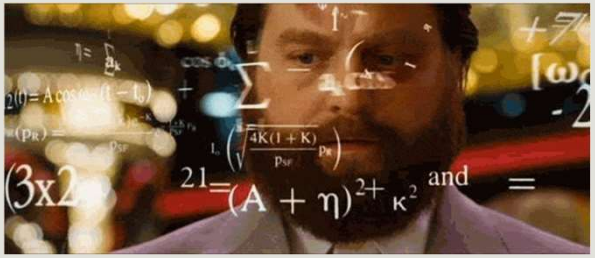
- **OmitType** vous permet de créer un nouveau type (une classe) **en enlevant un ensemble de champ** d'une classe existante.

```
export class UpdateTodoDto extends OmitType(AddTodoDto, ['name'])
```

- **IntersectionType** vous permet de créer un nouveau type (une classe) en **sélectionnant les champs qui existent dans deux types.**

```
export class UpdateTodoDto extends IntersectionType
(AddTodoDto,PatchTodoDto)
```

Exercices



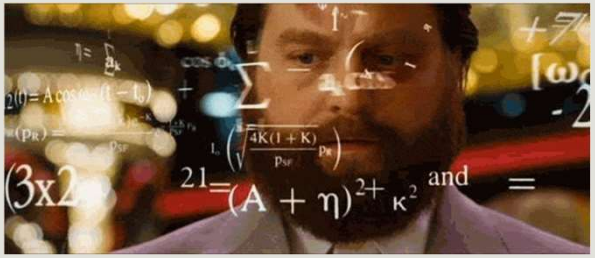
Simplifiez vos DTO's en utilisant les mapped types.

Class Transformer

- La validation se fait avant la transformation automatique
- Afin de gérer ca, Class Transformer vous permet de transformer vos données.
- En utilisant le décorateur @Type(() => TypeVersLequelTransformer)

```
@IsNotEmpty()
@Type(() => Number )
@IsNumber()
cin: number;
```

Exercices



- Ajouter un champ priority à votre model.
- Ce champ doit être un nombre, ajouter le validateur nécessaire.
- Faites les modifications nécessaires dans votre service afin de gérer le nouveau champ.
- Tester la fonctionnalité d'ajout en passant une chaine contenant un nombre.
- Résolvez la problématique soulevée en ajoutant un Type Transformer

Valider les objets imbriqués

- Afin de valider les objets imbriqués, utilisez le validateur @ValidateNested().
- N'oublier pas de transformer vos données avant de les valider en utilisant le @Type.

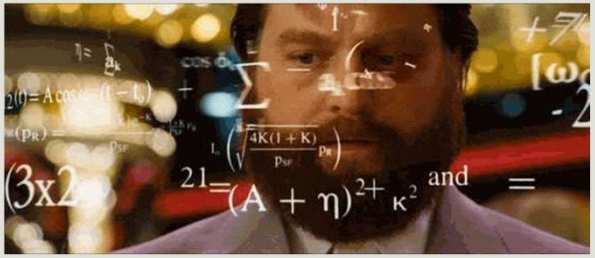
```
export class Post {
  @ValidateNested()
  @Type(() => User)
  user: User;
}
```

Valider les objets imbriqués

- Si vous avez un tableau d'objet à valider, utilisez l'option **each de ValidateNested** et mettez la à ***true(dans les anciennes versions)***.

```
export class User {
  @ValidateNested({ each: true })
  @Type(() => Post)
  posts: Post[];
}
```

Exercices



- Ajouter un champ owner qui représente un model avec un champ name et un champ age.
- Le name doit avoir au moin 3 caractère et l'age doit être un numérique
- Vérifier la validation de ces champs.

Pipes
Custom Pipe

- A part les pipes qui sont fournies avec Nest, vous pouvez créer vos propres pipes.
- Un pipe est une classe qui implémente l'interface **PipeTransform**.
- Cette interface vous demande d'implémenter la méthode **transform**.
- Cette méthode prends en paramètre la valeur à transformer et des metadata.

Pipes
Custom Pipe

- Les metdata fournies contiennent les informations suivantes :

  - **type** : indique le type de l'argument qui peut être un body avec @Body, un queryParam avec @Query, un paramètre avec @Param. ['body', 'query', 'param', 'custom']

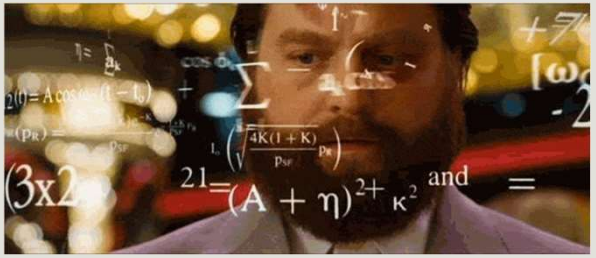
  - **metatype** : indique le type du paramètre, par exemple String

  - **data** : le nom de la donnée passée au décorateur

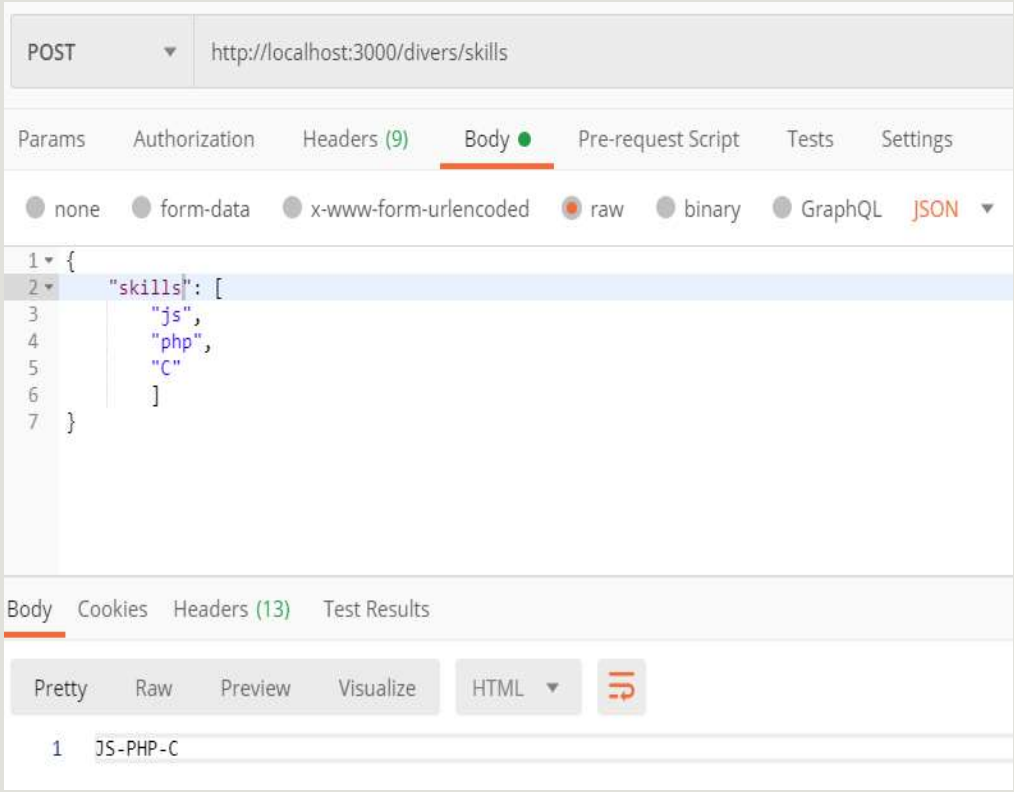
Pipes
Custom Pipe

```
import { ArgumentMetadata, PipeTransform } from '@nestjs/common';

export class FusionUpperPipe implements PipeTransform{
  transform(value: any, metadata: ArgumentMetadata): any {
    console.log(metadata);
    return value;
  }
}
```

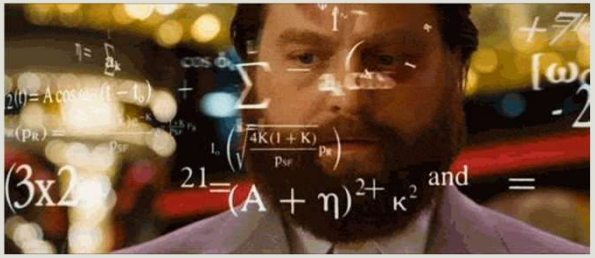
Pipes
Custom Pipe Exercice



- Créer un contrôleur qui prend une requête POST qui reçoit un tableau de chaine de caractères appelé skills dans le body.
- Créer un pipe qui ne traite que le Body et qui transforme tous les chaines en Majuscules, les fusionne en les séparant avec '-' et les retourne.
- Si le pipe ne reçoit pas la données skills il doit retourner un BADRequestException.



Pipes
FreezePipe Exercice



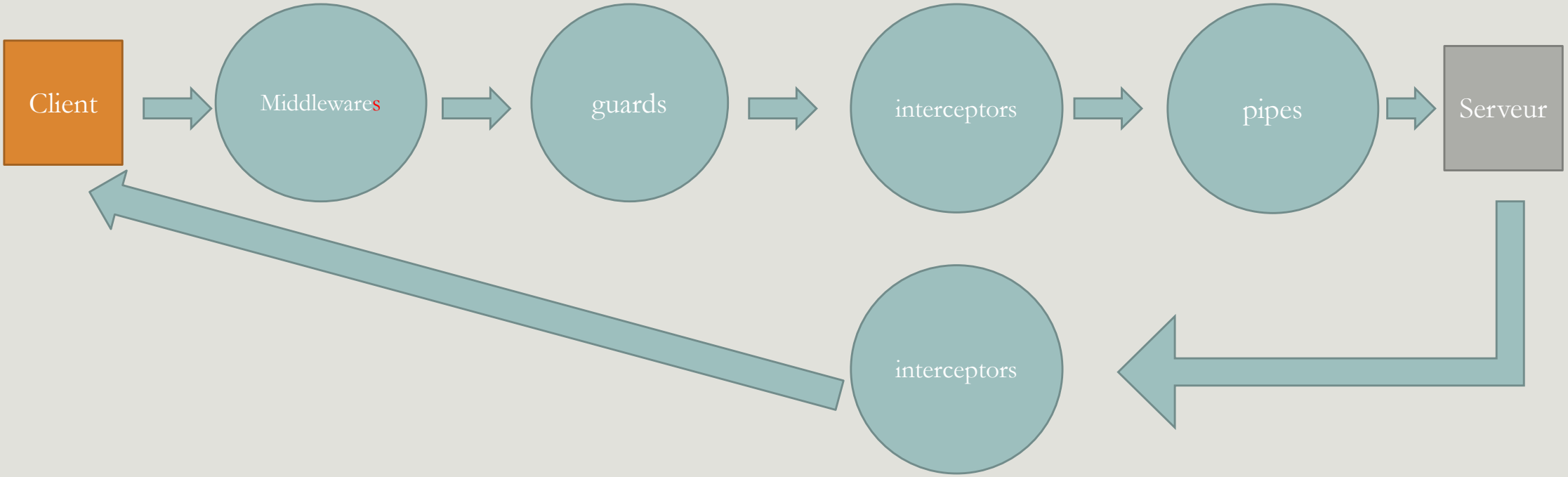
- Si vous voulez freezer un objet (le rendre non modifiable on ne peut ni lui ajouter des propriétés ni lui enlever) vous pouvez utiliser un pipe.

- Créer un pipe qui lorsqu'on l'appelle rend un objet gelé avec la méthode Object.freeze

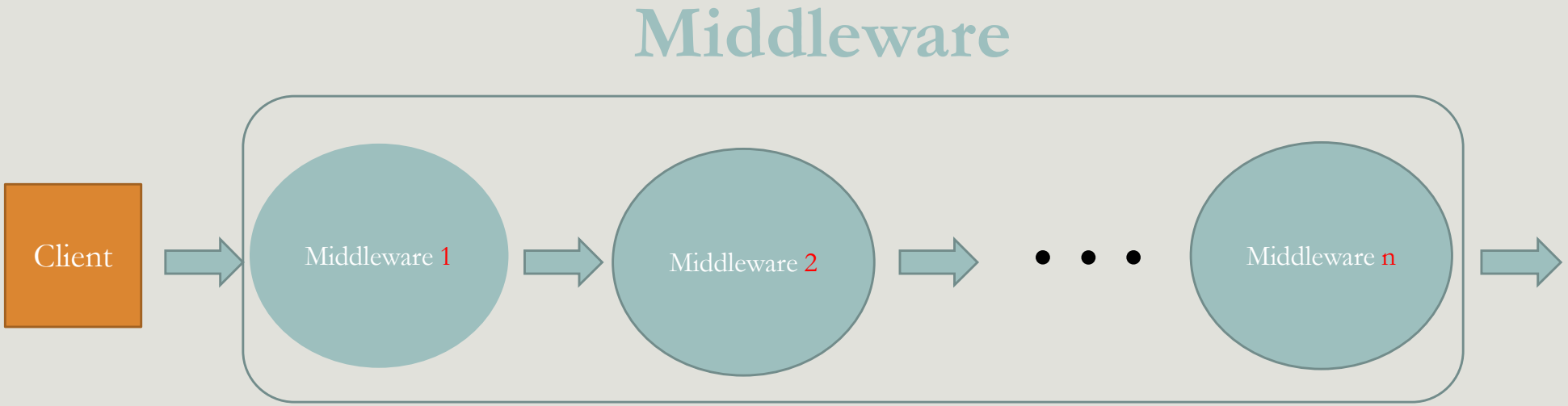
Middleware

- Un Middleware est tout simplement **une fonction appelée avant que la requête ne soit traitée par le contrôleur.**
- Un middleware a accès aux objets Request et Response et la fonction next.
- Un middleware est utilisé généralement pour l'une des tâches suivantes :
  - apporter des modifications à la requête ou à la réponse.
  - mettre fin au cycle requête-réponse.
  - si la fonction middleware actuelle **ne met pas fin au cycle requête-réponse**, elle **doit appeler la méthode next()** pour passer le contrôle à la fonction middleware suivante ou laisser la requête terminer son chemin. Sinon, la demande sera laissée en suspens.

Cycle de vie de la requête



Cycle de vie de la requête



Middleware

Afin d'implémenter votre Middleware vous pouvez le faire via :

- une **classe**
- une **fonction**.

Middleware Classe

Afin d'implémenter votre Middleware via une classe vous devez faire en sorte que :

- La classe doit **implémenter l'interface NestMiddleware**.
- En implémentant cette interface, vous devez **implémenter la méthode use**
- Cette méthode prend en paramètre la **requête**, la **réponse** et la méthode **next**.

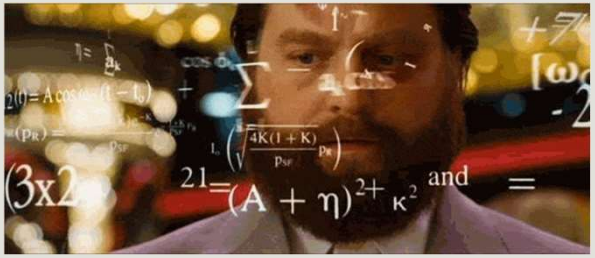
```
import { NestMiddleware } from '@nestjs/common';

export class FirstMiddlewar implements NestMiddleware{
  use(req: any, res: any, next: () => void): any {
  }
}
```

Middleware

- Afin d'appliquer un Middleware, nous devons tout d'abord, faire en sorte que votre **AppModule implémente l'interface NestModule**.
- Ensuite implémenter la méthode **configure** afin de **spécifier les Middlewares** à utiliser et **ou les appliquer.**
- Cette méthode reçoit en paramètre un **MiddlewareConsumer,** qui à travers sa méthode **apply,** permet de spécifier **quel Middleware appliquer** et avec la méthode **forRoutes sur quelle (s) ressource (s)** elle doit s'exécuter.

```
configure(consumer: MiddlewareConsumer): MiddlewareConsumer | void {
 consumer.apply(FirstMiddleware)
       // j'accepte toutes les routes commençant par 'courses'
       .forRoutes('courses');
}
```

Exercices



Créer votre premier Middleware FirstMiddlware.

Faite en sorte qu'il logue chaque requête entrante.

Middleware Fonction

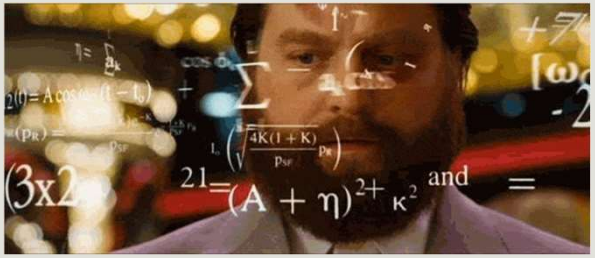
Afin d'implémenter votre Middleware via une fonction, vous devez simplement créer une fonction qui prend en paramètre :

- La requête,
- la réponse
- et la méthode next.

Appliquez le de la même manière qu'un middleware Classe.

```
import { Request, Response } from 'express';

export function logger(req: Request, res: Response, next: () => void) {
  // Todo Do what you want with your middleware
  next();
}
```

Exercices



Créer un Middleware loggerMiddlware en utilisant une fonction.

Afficher l'ip, le user-agent (à travers request.get('user-agent') et la méthode de chaque requête rentrante.

Middleware

- Vous pouvez aussi restreindre votre Middleware à certaines méthodes HTTP.
- En appelant forRoutes, passez un objet contenant la ressource avec la clé **path** et la méthode avec la clé **method**.

```
export class AppModule implements NestModule{
 configure(consumer: MiddlewareConsumer): MiddlewareConsumer | void {
  consumer.apply(FirstMiddlewar)
        .forRoutes(
         {path: 'courses', method: RequestMethod.GET},
         {path: 'courses', method: RequestMethod.POST}
         );
 }
```

Middleware

- La méthode forRoutes peut prendre en paramètre une **chaine**, une **séquences de chaine**, un **objet** de type **RouteInfo**, un **Contrôleur** ou une **séquence de contrôleurs**.

```
forRoutes(...routes: (string | Type<any> | RouteInfo)[]): MiddlewareConsumer;
```

```
export class AppModule implements NestModule{
  configure(consumer: MiddlewareConsumer): MiddlewareConsumer | void {
    consumer.apply(FirstMiddlewar)
            .forRoutes(CoursesController);
  }
}
```

Middleware

- Vous pouvez aussi déclarer vos ***functions middleware*** au niveau de main.js à travers la méthode use de votre app.
- Le middleware sera **global**.

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(secondMiddleware);
  await app.listen(3000);
}
bootstrap();
```

- Vous pouvez aussi le restreindre à un path que vous passez en premier paramètre

```
app.use(['/skill', '/*/todo'], secondMiddleware);
```

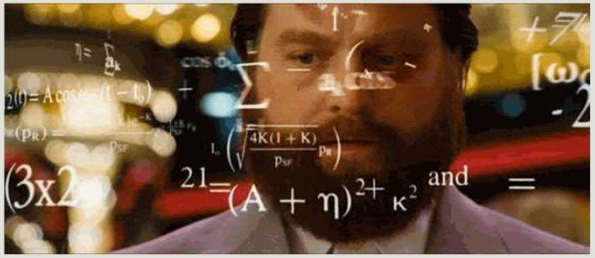
Middleware

- Les middlewares sont exécutés dans un ordre particulier.

1. Tout d'abord, Nest exécute les **middlewares globaux liés à app.use**
2. Ensuite il exécute les **middlewares liés au module principal**.
3. Ces middlewares sont exécutés **séquentiellement dans l'ordre dans lequel ils sont appelés**, de la même manière que les middlewares d'Express fonctionnent.
4. Dans le cas de middlewares **liés à d'autres modules**, **les middlewares liés aux modules racine s'exécuteront en premier**, puis les middlewares s'exécuteront dans **l'ordre dans lequel les modules sont ajoutés au tableau des importations**.

https://docs.nestjs.com/faq/request-lifecycle

Exercices



- Créer un Middleware qui simule un processus d'authentification
- Pour se faire, il devra en cas d'accès au contrôleur Todo pour la partie persistance vérifier l'existence d'un header 'auth-user' et qui contiendra un token jwt **(https://jwt.io/)**.
- S'il existe, il devra le décoder **(https://www.npmjs.com/package/jsonwebtoken, https://www.npmjs.com/package/@types/jsonwebtoken)** et en extraire une propriété userId.
- Ensuite il devra l'injecter dans l'objet request. Dans le todoController, faite en sorte que pour l'ajout, on ajoute le user et que pour l'update et le delete, uniquement le user qui a crée le todo puisse le modifier ou le supprimer.
- Si le Token n'existe pas, ou s'il ne contient pas de userId, retourner une réponse au client (https://expressjs.com/en/api.html#res.json) lui indiquant qu'il ne peut pas accéder à la ressource.

```
import { verify } from "jsonwebtoken";
```

Middleware third part Morgan logger

- Vous pouvez **utiliser tous les middlewares Express** que vous connaissez.
- Parmi eux, **Morgan** est un request logger pour node.js.
- Il vous offre une fonctionnalité de log sur les requêtes que vous lui spécifier.
- Il possède plusieurs formats de log que vous pouvez paramétrer et adapter.
- Site officiel Morgan : https://www.npmjs.com/package/morgan

Middleware third part Morgan logger

- L'utilisation est simple, installer le

```
npm install morgan
```

- **Importer morgan** au niveau de **main.ts,** ensuite utiliser la méthode **use** de votre **app**.

```
import * as morgan from 'morgan';
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(morgan('dev'));
  //....
}
```

Middleware third part Helmet

- **Helmet** est un **middleware express** qui vous permet de **sécuriser vos requêtes en ajoutant des HTTP headers**.
- C'est une **collection** de **15 fonctions middleware**.
- Ces fonctions permettent de gérer certaines attaques connues tels que :
  - **XSS** (Cross-site scripting) (Content-Security-Police)
  - **Clickjacking** (détournement de click)
  - ect

Middleware third part Helmet

- Site officiel Helmet : https://www.npmjs.com/package/helmet
- Pour l'installer lancer la commande : npm install --save helmet
- Faite la même chose que pour Morgan

Middleware third part Helmet

```
import * as helmet from 'helmet';
async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.use(helmet());
  /*...*/
}
```

Middleware third part Helmet

Module	Included by default?
contentSecurityPolicy for setting Content Security Policy	
dnsPrefetchControl controls browser DNS prefetching	✓
expectCt for handling Certificate Transparency	
featurePolicy to limit your site's features	
frameguard to prevent clickjacking	✓
hidePoweredBy to remove the X-Powered-By header	✓
hsts for HTTP Strict Transport Security	✓
ieNoOpen sets X-Download-Options for IE8+	✓
noSniff to keep clients from sniffing the MIME type	✓
permittedCrossDomainPolicies for handling Adobe products' crossdomain requests	
referrerPolicy to hide the Referer header	
xssFilter adds some small XSS protections	✓

Middleware third part Helmet

Preview | Header 16 | Cookie | Timeline

NAME	VALUE
Content-Security-Policy	default-src 'self';base-uri 'self';block-all-mixed-content;font-src 'self' https: data:;frame-ancestors 'self';img-src 'self' data:;object-src 'none';script-src 'self';script-src-attr 'none';style-src 'self' https: 'unsafe-inline';upgrade-insecure-requests
X-DNS-Prefetch-Control	off
Expect-CT	max-age=0
X-Frame-Options	SAMEORIGIN
Strict-Transport-Security	max-age=15552000; includeSubDomains
X-Download-Options	noopen
X-Content-Type-Options	nosniff
X-Permitted-Cross-Domain-Policies	none
Referrer-Policy	no-referrer
X-XSS-Protection	0
Content-Type	text/html; charset=utf-8
Content-Length	4
ETag	W/"4-2YaflKguFkfmXVCGRTlKcG6gj4o"
Date	Thu, 12 Aug 2021 09:22:38 GMT
Connection	keep-alive
Keep-Alive	timeout=5

Headers avec Helmet

Preview | Header 7 | Cookie | Timeline

NAME	VALUE
X-Powered-By	Express
Content-Type	text/html; charset=utf-8
Content-Length	4
ETag	W/"4-2YaflKguFkfmXVCGRTlKcG6gj4o"
Date	Thu, 12 Aug 2021 09:22:06 GMT
Connection	keep-alive
Keep-Alive	timeout=5

Headers sans Helmet

Middleware third part Cors

- **Cors (Cross-origin resource sharing)** est un middleware qui vous permet de gérer les permissions aux ressources à partir d'un autre domaine.
- NestJs vous permet d'utiliser ses propres fonctions de gestion de cors ou d'ajouter le middleware Cors qui se base sur celui de express.
- Même la fonction Cors utilise ce Middleware : https://www.npmjs.com/package/cors#enable-cors-for-a-single-route.
- L'application que vous avez crée (l'objet app) vous propose la méthode enableCors pour ajouter des cors.
- Elle prend en paramètre un objet d'options de type **CorsOptions**.

Middleware
third part
Cors, enableCors quelques options

- **origin**: Configure les origines qui peuvent accéder à la ressource.
  - Boolean
  - String
  - RegExp tableau
  - Fonction
- **methods**: Les méthodes acceptées. (ex: 'GET,PUT,POST') ou un tableau (ex: ['GET', 'PUT', 'POST']).
- **allowedHeaders**
- **optionsSuccessStatus**: Le code à utiliser en cas de succès.

Middleware third part Cors

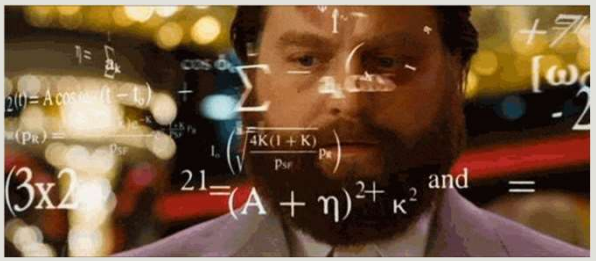
```
const corsOptions = {
  origin: ['http://localhost:4201', 'http://localhost:4200'],
  optionsSuccessStatus: 200
}
const app = await NestFactory.create(AppModule);
app.use(morgan('dev'));
app.enableCors(corsOptions);
```

Middleware third part Cors

```
async function bootstrap() {
  const app = await NestFactory.create(AppModule, {cors: true}););
  // app.use(cors());
  await app.listen(process.env.PORT || 3005);
}
bootstrap();
```

```
app.enableCors({
  origin: true
})
```

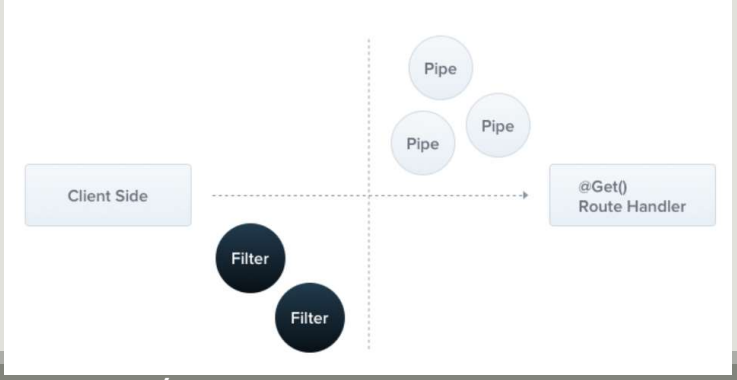
Exercice



- Testons ensemble les cors en le consultant via une application angular

Filters

- Dans NestJs les **filtres** sont utilisés afin de **gérer les exceptions**.
- NestJs vient avec une **couche de gestion des exceptions toute prête**.
- Si une exception **n'est pas gérée par votre application**, cette couche l'**intercepte** et **retourne l'erreur appropriée**.
- La couche de gestion des exceptions récupère l'exception et renvoi un message d'erreur approprié.



https://docs.nestjs.com/exception-filters

Filters
Fonctionnement

- Ceci est **géré par une Exception Filter globale qui gère toutes les exceptions de types HttpException** ou de ses **sous classes**.
- Lorsqu'une **exception n'est pas reconnue** (elle n'est pas de type HttpException ni une classe qui en hérite). Nest déclenche une réponse par défaut :

```
{
        "statusCode": 500,
        "message": "Internal server error“
 }
```

Filters
déclencher une erreur

- Nest fournit une classe HttpException.
- Son constructeur prend en paramètre deux arguments :
  - response : corps de la réponse qui peut être un string ou un objet.
  - status : le code http de la réponse. La bonne pratique est d'utiliser l'ENUM **HttpStatus** importé de **@nestjs/common.**
- Exemple :

```
if (course.id) {
  throw new HttpException('Vous ne pouvez pas mentionner d'id', 400);
}
```

Filters

Nest fournit plusieurs exceptions standards qui **héritent du HttpException**

BadRequestException
UnauthorizedException
NotFoundException
ForbiddenException
NotAcceptableException
RequestTimeoutException
ConflictException
GoneException
HttpVersionNotSupportedException
UnsupportedMediaTypeException
UnprocessableEntityException
InternalServerErrorException
NotImplementedException
ImATeapotException
MethodNotAllowedException
BadGatewayException
ServiceUnavailableException
GatewayTimeoutException
PayloadTooLargeException

Filters
Filters personnalisés

Afin de créer votre **propre Filter,** suivez les étapes suivantes :

1. Créer une classe qui implémente l'interface **ExceptionFilter**
2. L'annoter avec **@Catch** qui prend en paramètre le type de l'erreur à gérer (exp HttpException)
3. Implémenter la méthode **catch** qui prend en paramètre l'exception, et un objet de la classe ArgumentHost.
4. La classe ArgumentHost possède une méthode **switchToHttp** qui vous retourne un objet context vous permettant de récupérer la requête et la réponse.
5. Vous pouvez donc retourner la réponse que vous voulez.

Filters

```
import { ArgumentsHost, Catch, ExceptionFilter, HttpException } from '@nestjs/common';
import { Request, Response } from 'express';
@Catch(HttpException)
export class CustomFilter implements ExceptionFilter{
  catch(exception: HttpException, host: ArgumentsHost): any {
    const ctx = host.switchToHttp();
    const response = ctx.getResponse<Response>();
    const request = ctx.getRequest<Request>();
    // Custom response
    }
}
```

Filters
Filters personnalisés

- Afin que ce filtre soit celui utilisé pour tous vos requêtes, vous pouvez le provider au niveau du module principale avec le token ***APP_FILTER.***

```
import { APP_FILTER } from '@nestjs/core';
import { AllExceptionsFilter } from './filters/exception.filter';
import {   MiddlewareConsumer, Module} from '@nestjs/common';
import { AppController } from './app.controller';
import { AppService } from './app.service';
@Module({
 //...
 providers: [
  {
    provide: APP_FILTER,
    useClass: AllExceptionsFilter,
  }
 ],
})
export class AppModule  {}
```

Filters
Filters personnalisés

- La deuxième méthode est d'utiliser la méthode **useGlobalFilters** de votre **app** dans **main.js**

```
import { NestFactory } from '@nestjs/core';
import { AppModule } from './app.module';
import { AllExceptionsFilter } from './filters/exception.filter';

async function bootstrap() {
  const app = await NestFactory.create(AppModule);
  app.useGlobalFilters(new AllExceptionsFilter())
  await app.listen(3000);
}
bootstrap();
```

Filters
Filters personnalisés

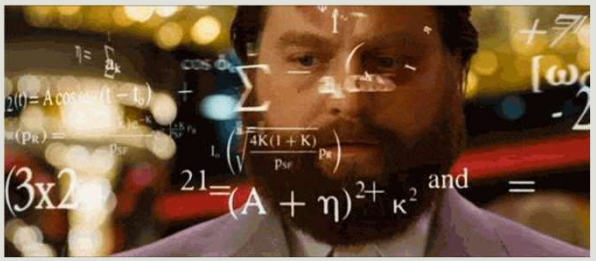
- Pour attacher ce filtre à une méthode particulière, décorer la avec **@UseFilter** qui prend en paramètre une instance du filtre à appliquer ou la classe elle même. Si elle est vide elle acceptera tous les types d'exceptions.

Remarque: préférez la classe, ceci déléguera l'instanciation au Framework qui utilisera l'injection de dépendance, vous aurez ainsi un gain en mémoire.

- Vous pouvez l'appliquer aussi sur toutes les méthodes d'un Controller en le décorons avec **@UseFilter**

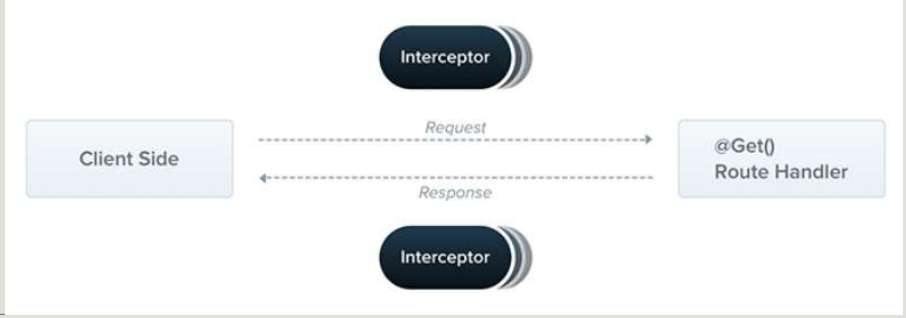
- Si vous voulez l'appliquer partout, il faut l'ajouter avec **app.useGlobalFilters(Filters). L'odre des filtres doit être du filtre le plus générique vers le plus spécifique.**

Exercice



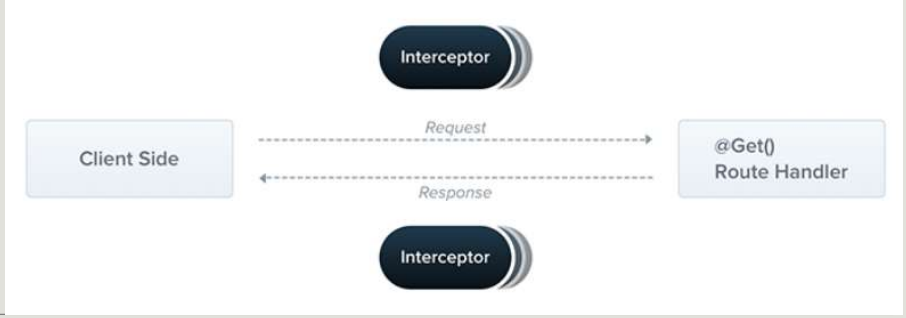
- Créez un filtre qui vous permet de retourner une réponse à l'utilisateur plus explicite avec le statut de l'erreur, la date ainsi que le message de l'erreur préfixé par 'Le message de l'erreur est :'

Interceptors



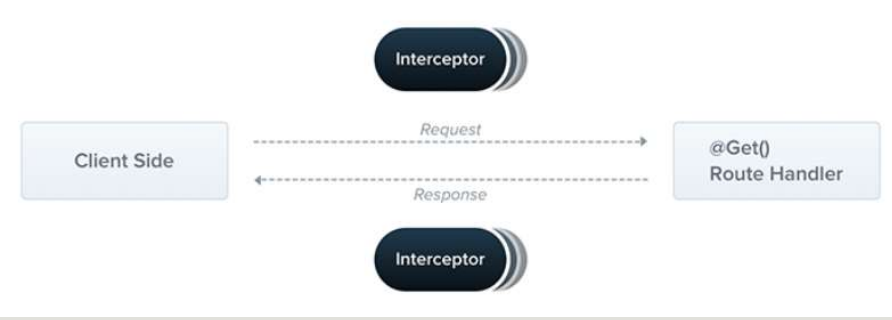
- Un ***interceptor*** est une classe annotée avec le décorateur @Injectable et qui implémente l'interface NestInterceptor.
- Il a pour rôle de :
  - lier une **logique supplémentaire avant / après** l'exécution de la méthode
  - **transformer** le résultat renvoyé par une fonction
  - **transformer** l'exception levée à partir d'une fonction
  - étendre le comportement de la fonction de base
  - remplacer complètement une fonction en fonction de conditions spécifiques (par exemple, à des fins de mise en cache)

Interceptors



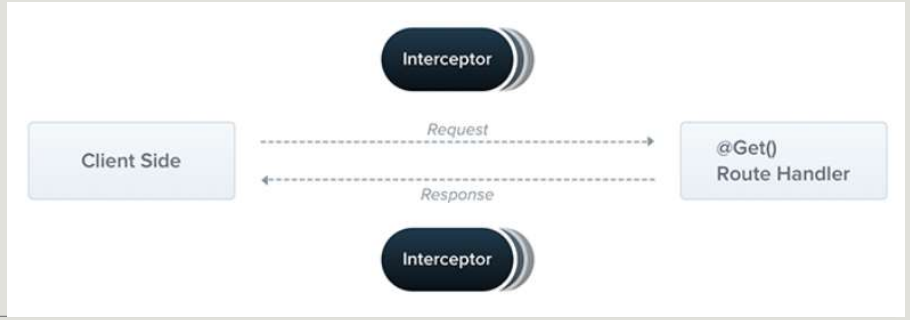
- Un **interceptor** doit **implémenter la méthode intercept** qui prend en paramètre
  - **le contexte d'exécution**
  - l'objet **next** qui est un **CallHundler**. Il a comme attribut la méthode **handle** qui retourne un **observabe**
- En appelant la méthode **handle** vous remettez la requête dans son chemin
- Afin de pouvoir **intercepter la réponse**, la méthode **handle retournant un observable**, vous pouvez utiliser ces opérateurs afin de faire ce que vous voulez à ce niveau

Interceptors




```
import {
  Injectable, NestInterceptor, ExecutionContext, CallHandler
} from '@nestjs/common';
import { Observable } from 'rxjs';
import { tap } from 'rxjs/operators';
@Injectable()
export class MyFirstInterceptor implements NestInterceptor {
  intercept(context: ExecutionContext, next: CallHandler): Observable<any> {
    console.log('Before...');
    return next.handle().pipe(tap(() => console.log(`After...`)));
  }
}
```

Interceptors



- Afin de définir le domaine d'exécution de l'intercepteur, vous pouvez le spécifier pour
  1. une route,
  2. un controller
  3. globalement

```
@Get('')
@UseInterceptors(RequestDurationInterceptor)
getAllProducts() {
  return this.produitService.getAllProduits();
}
```

1

```
@UseInterceptors(RequestDurationInterceptor)
@Controller('post')
export class PostController {
  constructor(
    public service: PostService
  ) {
  }
```

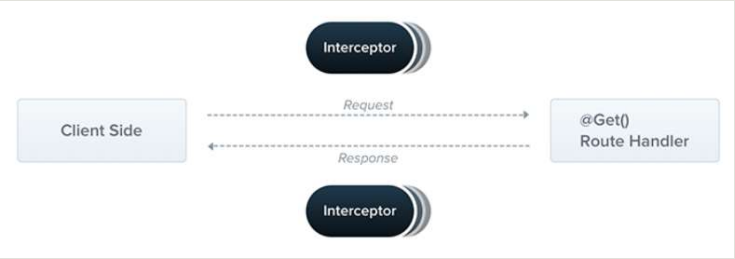
2

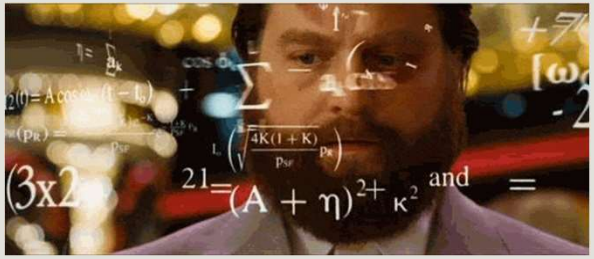
```
app.useGlobalInterceptors(
  new ErrorHandlerInterceptor(),
  new RequestDurationInterceptor(),
  new TransformInterceptor(),
  new ExcludNullInterceptor()
);
```

3

https://docs.nestjs.com/interceptors

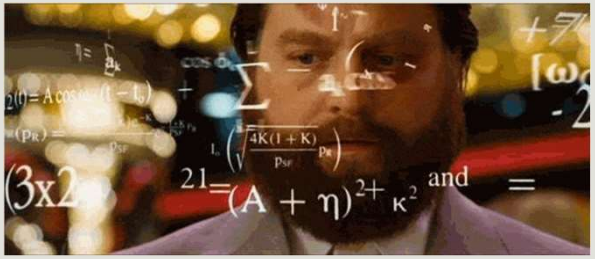
Interceptors





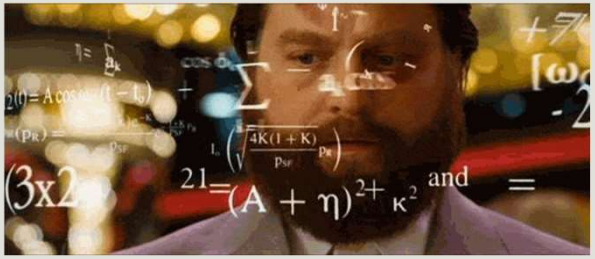
- Nous voulons calculer le temps d'exécution d'une requête.
- L'idée est de faire la différence entre la date de récupération de la requête et la date de la réponse.

Interceptors Exercice



- Faites en sortes que toutes vos réponses soient dans un champs data.

Interceptors Exercice



- Faites en sortes que toute réponse qui est Null est transformée en une chaine vide.

Variables de configuration
.env

- Généralement les variables de configuration de vos applications sont définies au niveau de fichier .env.
- Ces fichiers sont définies avec des paires clé valeur.
- Mettez les au niveau de la racine de votre projet

```
PROJECT_PORT=3000
DATABASE_HOST=localhost
```

Variables de configuration
.env

- La première façon de gérer ca et d'utiliser le module express **dotenv** en l'installant avec la commande : npm i dotenv.
- Ce module ira charger les fichiers .env

```
import * as dotenv from 'dotenv';
```

- Importer le module dotenv dans le fichier main.ts.
- Appeler ensuite la méthode config qui ajoute toutes les variables de .env à une variable globale **process.env** : dotenv.config();
- Finalement pour accéder à une de vos variables d'environnement utiliser la variable process.

process.env.nomVariable exemple process.env.PORT

Configuration
.env

- NestJs offre un Module de configuration: **ConfigurationModule**
- Pour l'installer, utiliser la commande suivante :

npm i --save @nestjs/config

- Importer ce Module (généralement au niveau du AppModule) et contrôler son fonctionnement avec la méthode forRoot qui prend en paramètre un objet d'option de type **ConfigModuleOptions**.
- Cet appel permettra de charger tous les fichier .env.
- Pour rendre ce module accessible d'une manière globale, ajouter l'option **isGlobal** et mettez a à **true**

```
ConfigModule.forRoot({
  isGlobal: true,
}),
```

https://docs.nestjs.com/techniques/configuration

Configuration
Fichiers de configuration

- Le module **ConfigModule** peut prendre en paramètre des **fichiers de configuration** qui **retourne un objet de paramètres** permettant ainsi une meilleur flexibilité.
- Pour ce faire, utiliser la clé load qui prend en paramètre un tableau de Factory de configuration.

```
import configuration from './config/configuration';
ConfigModule.forRoot({
  load: [configuration],
}),
```

```
export default () => ({
  database: {
    name: 'bd_nest',
  },
});
```

configuration

Configuration
Fichiers de configuration

- Vous pouvez aussi définir **plusieurs fichiers de configuration** que vous voudrez **charger selon l'environnement de développement**.
- Pour ce faire, définissez plusieurs fichiers de config et **selon votre environnement charger le fichier adéquat**.

```
load: [
  process.env.NODE_ENV == 'development'
    ? devConfiguration
    : prodConfiguration,
],
```

Configuration
Fichiers de configuration

- Vous pouvez modifier votre fichier **package.json** afin de **définir le mode de développement au lancement de votre script**.
- Utiliser la bibliothèque **cross-env** afin que le script s'adapte à l'OS avec lequel vous travaillez.

```
"start:dev": "cross-env NODE_ENV=development nest start --watch",
"start:prod": "cross-env NODE_ENV=production node dist/main",
```

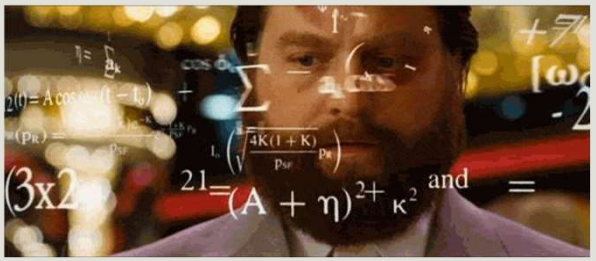
Configuration
.env

- Maintenant pour récupérer les paramètres, injecter le service offert par le module de configuration et qui est une instance de la classe **ConfigService**.
- Le service offre une méthode **get** qui prend en paramètre la **clé de la variable** d'environnement que vous souhaitez récupérer.

```
export class ProductController {
 constructor(
  private productService: ProductService,
  private configService: ConfigService
  ) {
 }
 @Get()
 @UseGuards(AuthGuard('jwt'), AdminGuard)
 async getProducts() {
  console.log(this.configService.get('PROJECT_PORT'));
  return await this.productService.getProducts();
 }
```

https://docs.nestjs.com/techniques/configuration

Interceptors Exercice



- Faite en sorte d'avoir trois fichier de configuration. Un pour la prod, un pour la prerod et un pour le test.
- L'environnement par défaut doit être l'environnement de développement.
- Tester le bon fonctionnement de vos paramètres de configuration selon l'environnement dans lequel vous êtes.

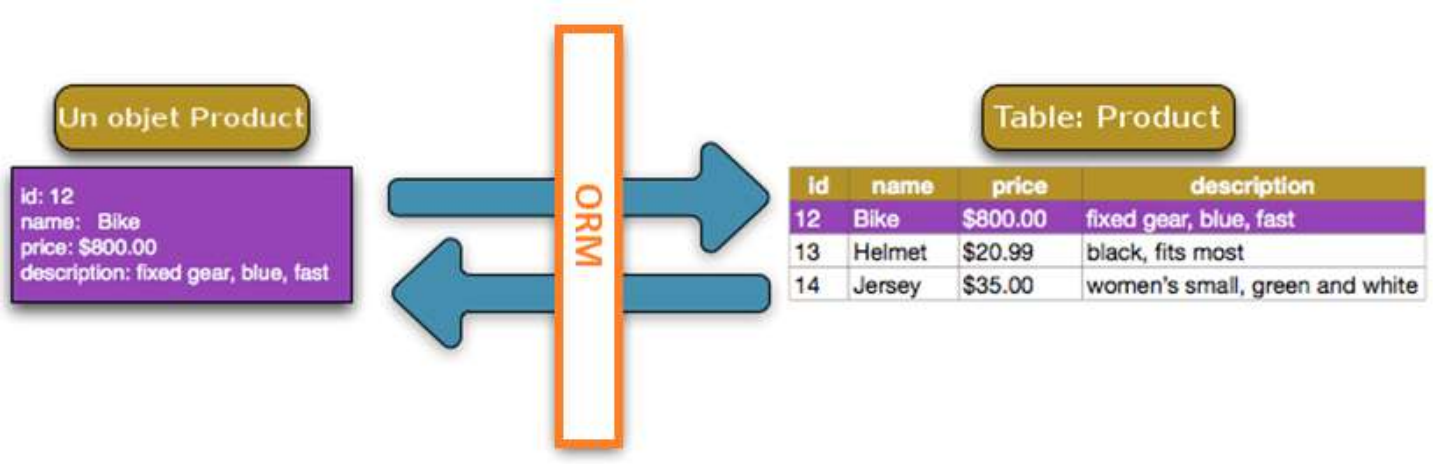
Interaction avec une base de données

- NodeJs est **agnostique** vis-à-vis des bases de données.
- Il peut **s'interfacer** avec **n'importe quelle base de données Sql ou NoSql**
- Connecter NestJs à une base de données consiste au **chargement du driver node.js approprié** pour cette base.
- Vous pouvez aussi utiliser n'importe quelle librairie utilisée avec nodeJs pour s'interfacer à une base de données telle que **Sequelize** ou **TypeOrm**.

ORM

- ORM : Object Relation Mapper
- Couche d'abstraction
- Gérer la persistance des données
- Mapper les tables de la base de données relationnelle avec des objets
- Crée l'illusion d'une base de données orientée objet à partir d'une base de données relationnelle en définissant des correspondances entre cette base de données et les objets du langage utilisé.
- Propose des méthodes prédéfinies

ORM



id	name	price	description
12	Bike	$800.00	fixed gear, blue, fast
13	Helmet	$20.99	black, fits most
14	Jersey	$35.00	women's small, green and white

- TypeORM : ORM TypeScript conseillé par la communauté NestJs

TypeORM

- TypeORM est un ORM en TypeScript.
- Actuellement c'est l'ORM le plus mature en TypeScript.
- Afin d'utiliser TypeORM, NestJs vous offre un package permettant de vous faciliter la tâche. C'est le package @nestjs/typeorm.
- Ce package utilise le package typeorm

TypeORM

- Installer TypeORM et ses bibliothèques nest :

npm install --save @nestjs/typeorm typeorm

- Maintenant et selon la base de données que vous utiliser, installer le driver Node.Js correspondant.
- Dans notre cas nous allons utiliser MySql.
- Nous lançons donc la commande

npm install mysql2

TypeORM Configuration

- Une fois installé, nous devons **configurer Typeorm** en **important** le Module **TypeOrmModule** dans **app.module.ts**.
- Ensuite, appeler la méthode **forRoot** de ce module et passez y un objet **config** contenant les **informations relatives à la configuration** de votre base de données.
- Comme bonne pratique, utilisez les **variables d'environnement** pour y stocker les données.

```
imports: [
  ProduitModule,
  TypeOrmModule.forRoot(
   {
    type: 'mysql',
    host: process.env.DB_HOST,
    port: 3306,
    username: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,
    entities: [],
    synchronize: true,
   }
  )
],
```

TypeORM Configuration

- Afin de pouvoir utiliser le configService, vous devez utiliser la méthode for **RootAsync** qui utilise un **factory** et auquel on peut **injecter** notre service de configuration.

```
TypeOrmModule.forRootAsync({
  useFactory: (configService: ConfigService) => ({
    //...
    database: configService.get('database.name'),
  }),
  inject: [ConfigService],
}),
```

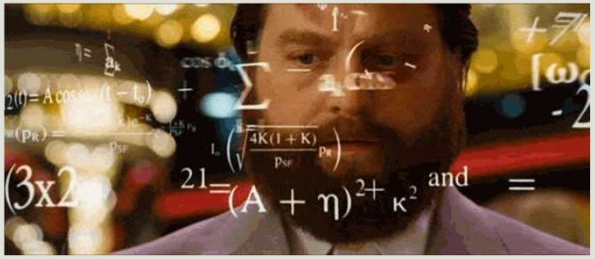
TypeORM Configuration

- Dans l'objet de configuration passé à forRoot, vous pouvez ajouter d'autres options :

retryAttempts	Nombre de tentatives de connexion à la base de données (par défaut: 10)
retryDelay	Le délai entre nouvelles tentatives de connexion (ms) (par défaut: 3000)
autoLoadEntities	Si true, les entités seront chargées automatiquement (par défaut: false)

https://typeorm.io/data-source-options

Exercice



- Configurer TypeOrm au niveau de votre application
- Utiliser le useFactory afin de récupérer les paramètres directement du Service ConfigService.

TypeOrm
Entity

- Etant donné que TypeORM est un ORM, il se base donc sur des classes qui vont représenter l'image des tables de votre base de données.
- Ces classes sont appelées **Entity**.
- Afin de spécifier à TypeORM qu'une classe est une entité, vous devez l'annoter avec **@Entity**. Ce décorateur prend en **paramètre** le **nom** qu'aura la **table associée** à votre entité. S'il **n'est pas mentionné**, il sera **identique au nom de l'entité**.

```
import { Entity } from 'typeorm';
```

TypeOrm
Entity

- Afin de spécifier à TypeORM qu'une propriété d'une entité est une colonne de la table, vous devez l'annoter avec **@Column**.
- Tous les décorateurs sont importé de la bibliothèque **typeorm**.
- Chaque entité doit être **enregistrée dans vos options de connexion sous la clé entities. Sinon elle ne sera pas prise en considération**.
- Une fois l'entité définie, TypeORM se charge de créer automatiquement la table associée à votre entité. Chaque mise à jour de votre entité se reflétera automatiquement à la table en question.

```
import { Entity, Column} from 'typeorm';
```

TypeOrm Entity

```
import {Column, Entity, PrimaryGeneratedColumn } from 'typeorm';
import { Roles } from '../../Enums/roles.enum';
import { PostEntity } from '../../post/entity/post.entity';
@Entity('user')
export class UserEntity {
  @PrimaryGeneratedColumn()
  id: number;
  @Column({length:50, unique: true})
  username: string;
  @Column()
  password: string;
  @Column({unique: true})
  email: string;
  @Column({type: 'enum', enum: Roles, default: Roles.user})
  role: Roles;
}
```

```
TypeOrmModule.forRoot(
  {
    type: 'mysql',
    host: process.env.DB_HOST,
    port: 3306,
    username: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_NAME,
    entities: [UserEntity],
    synchronize: true,
  }
),
```

TypeOrm
Entity
Clé primaire

- Chaque entité **doit obligatoirement avoir une clé primaire**
- L'annotation **@PrimaryColumn** permet de spécifier une **clé primaire**. Cette clé doit être affectée manuellement.
- Pour spécifier qu'une propriété de l'entité est une clé primaire auto générée, utiliser l'annotation **@PrimaryGeneratedColumn.** La valeur de la clé est gérée automatiquement. Vous aurez une séquence numérique incrémentée à chaque fois.

```
@Entity('post')
export class PostEntity {

 @PrimaryGeneratedColumn()
 id: number;
```

TypeOrm
Entity
Clé primaire

- Si vous voulez avoir une chaine de caractère unique en tant que clé primaire auto générée, ajouter le paramètre « **uuid** » à votre décorateur.

```
@Entity('post')
export class PostEntity {

 @PrimaryGeneratedColumn("uuid")
 id: string;
```

TypeOrm
Entity
Clé primaire composite

- Vous pouvez avoir une clé primaire composite

```
@PrimaryColumn()
firstName: string;
@PrimaryColumn()
lastName: string;
```

TypeOrm
Entity
Options des colonnes

Vous pouvez spécifier des options à vos colonnes. Les options dépendent du type du champ :

- **type** : type du champs, par défaut extrait du type de l'attribut.
- **name** : nom du champ, par défaut c'est le nom de l'attribut.
- **length** : taille
- **nullable** : booléen informant si un champ est nullable ou non, par défaut la valeur est à false.

TypeOrm
Entity
Options des colonnes

- **unique** : booléen informant si un champ est unique ou non, par défaut la valeur est à false.
- **update**: boolean - Indique si la valeur de la colonne est mise à jour par l'opération "save". Si faux, vous ne pourrez écrire cette valeur que lors de la première insertion de l'objet. La valeur par défaut est vraie.
- **select**: boolean - Définit s'il faut ou non masquer cette colonne par défaut lors des requêtes. Lorsqu'elle est définie sur false, les données de la colonne ne s'afficheront pas avec une requête standard. La colonne par défaut est select: true

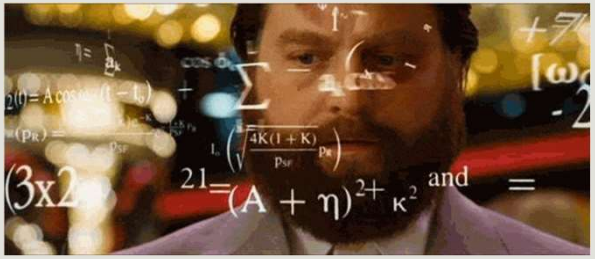
TypeOrm
Entity
les types

- TypeORM **supporte la plupart des types** de colonnes utilisés par les différents SGBD.
- Les colonnes sont **database-type spécifique** ce qui permet d'avoir beaucoup de flexibilités.
- Prenons l'exemple du type **enum** supporté par **mysql et postgres**, dans les options de la colonne et en spécifiant le type à **enum**, vous pouvez spécifier la propriété **enum** et informer sur **l'enum utilisé** et la propriété **default** spécifiant la **valeur par défaut**.

```
@Column({
  type: "enum",
  enum: UserRole,
  default: UserRole.GHOST
})
role: UserRole
```

https://typeorm.io/#/entities/column-types

Exercice



- Créer une entité TodoEntity qui représente votre todo au niveau de la base de données.
- Pour rappel un todo est caractérisé par
  - Id : chaine en uuid
  - Name
  - Description
  - CreatedAt
  - Status : de type StatusEnum que vous avez déjà défini

TypeOrm
Entity
Colonne spéciales

- Nest nous fournit un ensemble de **colonnes spéciales** qui permettent un certain nombre de fonctionnalités.
- **@CreateDateColumn** permet, lorsqu'elle est associé à une colonne, **d'automatiquement définir la date d'insertion de l'entité**. Vous n'avez pas besoin de remplir cette colonne - elle sera automatiquement remplie.
- **@UpdateDateColumn** permet, lorsqu'elle est associé à une colonne, de gérer automatiquement la **date de mise à jour de l'entité** chaque fois que vous appelez la sauvegarde du gestionnaire d'entités ou du repository.

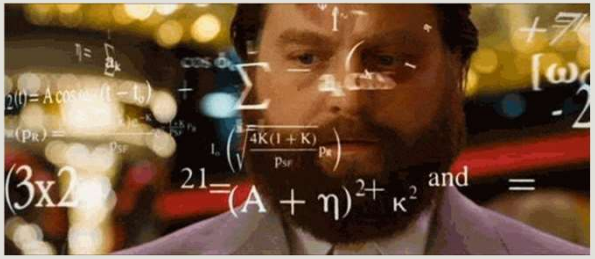
TypeOrm
Entity
Colonne spéciales

- **@DeleteDateColumn** est une colonne spéciale qui permet, lorsqu'elle est associé à une colonne, **de gérer automatiquement l'heure de suppression de l'entité** chaque fois que vous appelez la **méthode softDelete** du gestionnaire d'entités ou du repository.
- **@VersionColumn** est une colonne spéciale qui permet, lorsqu'elle est associé à une colonne, de gérer **la version de l'entité** (**numéro incrémentiel**) chaque fois que vous appelez la **sauvegarde du gestionnaire d'entités ou du repository.**

TypeOrm Entity

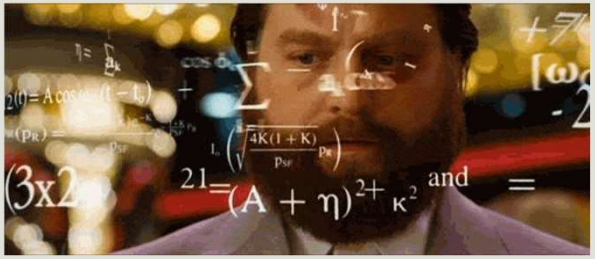
```
@Entity('todo')
export class TodoEntity {
  @PrimaryGeneratedColumn()
  id: number;
 //...
  @CreateDateColumn()
  createdAt: Date;
  @UpdateDateColumn()
  updatedAt: Date;
  @DeleteDateColumn()
  deletedAt: Date;
  @VersionColumn()
  version: number;
  //...
}
```

Exercice



- Mettez à jour votre entité TodoEntity en y ajoutant deux champs updatedAt et deletedAt et faite en sortes que ces deux champs soient automatiquement gérés par TypeOrm.
- Faite la modification nécessaire pour que le champ createdAt ne puisse pas être modifié une fois crée.

Exercice



- Etant donné que les champs cretaedAt, updatedAt et deletedAt sont des champs qu'on peut utiliser plusieurs fois, proposer une méthodes pour les rendre réutilisable.

TypeOrm
Le patron de conception Repository

- TypeORM supporte le **patron de conception Repository**.
- **Chaque Entité** aura donc son **propre Repository**.
- Ce Repository ou dépôt **héritera d'un ensemble de fonctionnalités**. Vous **pouvez aussi définir vos propres fonctionnalités**.
- Vous avez donc le choix, ou vous utiliser le Repository de base, ou vous créer votre propre Repository qui va étendre le Repository de base.

TypeOrm
Le patron de conception Repository
Utiliser le Repository de base

- Afin d'utiliser le repository de base, vous devez:

1. **Importer le TypeOrmModule** dans le **module** de l'**entité**.
2. Appeler la méthode **forFeature** du **TypeOrmModule** et passer y comme paramètre les **entités associées à ce module**.

```
@Module({
  imports: [
    TypeOrmModule.forFeature(
    [PostEntity]
    )
  ],
  providers: [PostService],
  controllers: [PostController]
})
export class PostModule {}
```

TypeOrm
Le patron de conception Repository
Utiliser le Repository de base

3. Aller la ou vous voulez **injecter votre Repository** (généralement le service) et injecter une instance du Repository associée à votre entité.
4. Annoter ce service avec l'annotation **@InjectRepository** auquel vous passez l'Entité que vous traitez.

Votre service est maintenant prêt à l'emploi avec les méthodes qui y sont incluses.

```
export class PostService {
  constructor(
    @InjectRepository(PostEntity)
    private readonly postRepository: Repository<PostEntity>,
  ) {}
```

TypeOrm
Configurer les entités

- Vous pouvez **activer l'auto-chargement** des entités.
- Ceci se fait :
  - En important TypeOrmModule dans les différents modules qui vont l'utiliser ou vous le configurer avec la méthode **forFeature (à la quelle vous passez un tableau d'entités)** et non **forRoot**.
  - En chargeant automatiquement, **toutes les entités enregistré avec la méthode forFeature.**

```
TypeOrmModule.forRoot(
 {
  //..
  //entities: ["dist/**/*.entity{.ts,.js}"],
  autoLoadEntities: true,
 }
),
```

TypeOrm
Le patron de conception Repository
save

- Le repository offre une **multitude de méthodes** et de propriétés. Nous présentons ici une partie de ces méthodes.
- La méthode **save** prend en paramètre **une entité ou un tableau d'entités**. **Si l'entité existe**, elle la **met à jour**, sinon elle l'**ajoute**.
- Dans le cas d'ajout d'un **tableau d'entités**, l'ajout se fait à travers une **transaction**, cad qu'en cas d'échec d'un des save la totalité est annulée via un Rollback.

```
async addSection(section) {
  return await this.sectionRepository.save(section);
}
```

```
async addSections(...sections) {
  return await this.sectionRepository.save([...sections]);
}
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

TypeOrm
Le patron de conception Repository
save

- Save supporte aussi **la mise à jour partielle**. Dans ce cas, les champs inexistants ne seront pas pris en considération.
- Elle **retourne** la **liste des entités modifiées ou mises à jour**.
- La méthode save comme la majorité des méthodes du Repository est **asynchrone**.

```
async addSection(section) {
  return await this.sectionRepository.save(section);
}
```

Versioning

- Dans certains cas, vous voulez avoir **plusieurs versions d'une même fonctionnalité**.
- Imaginez que votre client vous demande une seconde version et vous voulez l'exposer pour test tout en gardant l'ancienne version. NestJs vous le permet à partir de sa version 8.
- Il existe 3 types de versioning :

URI Versioning	La version sera passée dans l'**uri** de la requête et c'est la méthode par défaut
Header Versioning	Un **header personnalisé** permettra de spécifier la version.
Media Type Versioning	L'**Accept header de la requête** spécifiera la version

https://docs.nestjs.com/techniques/versioning

Versioning
URI Versionning

- Afin de configurer le versioning, appeler la méthode **enableVersioning** de votre app dans le fichier main.ts.
- Cette méthode prend en paramètre un objet d'options avec comme propriété fixe, **type,** représentant le type de versioning et d'autres options variables selon le type.
- Le deuxième paramètre est **prefix** et qui a comme valeur par défaut '**v**' c'est ce qui va préfixer l'uri de votre version (/api/**v**1/todo)

```
app.enableVersioning({
  type: VersioningType.URI
});
```

https://docs.nestjs.com/techniques/versioning

Versioning
URI Versionning
Controller

- Vous pouvez **versionner un contrôleur** permettant donc de **versionner toutes ses routes**.
- Pour donner la version de votre contrôleur ajouter la propriété **version** à votre tableau d'options de l'@nnotation **@Controller**

/v1/todo

```
@Controller({
  path: 'todo',
  version: '1',
})
export class TodoDbController {
  @Get('')
  getTodos() {
```

/v2/todo

```
@Controller({
  path: 'todo',
  version: '2',
})
export class TodoController {
  @Get('')
  getTodos() {
```

Versioning
URI Versionning
Route

- Vous pouvez **versionner une route**.
- Pour donner la version de votre route, **@nnoter votre route** avec le décorateur **@Version** et passer lui votre version

```
@Controller('todo')
export class TodoController {
 @Get('')
 @Version('1')
 getTodos() {
  return 'v1';
 }
 @Get('')
 @Version('2')
 getV2Todos() {
  return 'v2';
 }
}
```

/v1/todo

/v2/todo

https://docs.nestjs.com/techniques/versioning

Versioning
Header Versioning

```
app.enableVersioning({
  type: VersioningType.HEADER,
  header: 'Your-Custom-Header',
});
```

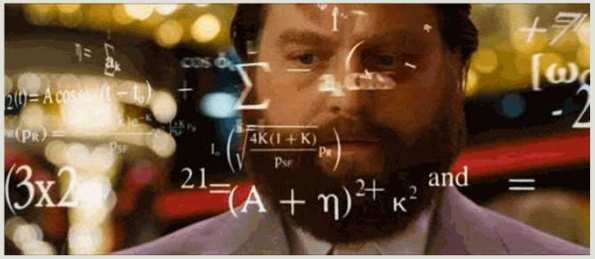
- La propriété **header** doit être le nom de l'en-tête qui contiendra la version de la requête.
- Ici Your-Custom-Header

Versioning
Media Type Versionning

```
app.enableVersioning({
  type: VersioningType.MEDIA_TYPE,
  key: 'v=',
});
```

- Dans l'en-tête **Accept**, la version sera séparée du type de média par un **;**.
- Il doit ensuite contenir une paire clé-valeur qui représente la version à utiliser pour la requête, telle que **Accept : application/json;v=2**

Exercice



- Modifier la méthode addTodo afin qu'elle puisse ajouter un todo dans la base de données.
- Garder les deux versions de votre code (Utiliser l'URI Versionning)

TypeOrm
Le patron de conception Repository
preload

- Afin de créer une EntityType (TodoEntity) à partir d'un objet que vous posséder (DTO par exemple) vous pouvez passer par la méthode **preload de votre Repository**.
- Cette méthode prend en paramètre **l'objet en question.**
- **Si l'entité existe** déjà dans la base de données, elle **la charge** (et tout ce qui y est lié), **remplace toutes les valeurs par les nouvelles valeurs** de l'objet donné et **renvoie l'entité mise à jour**.

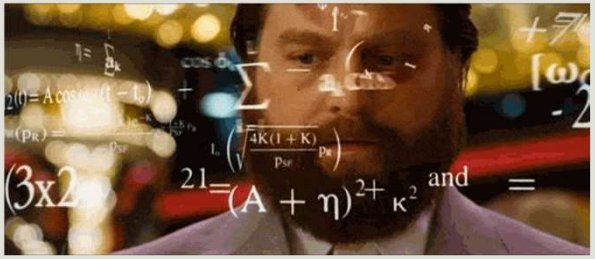
```
const newEntity = await repository.preload({id, name, firstname});
```

TypeOrm
Le patron de conception Repository
preload

- Notez que l'objet de type entité donné **doit avoir un identifiant d'entité** / une clé primaire pour rechercher l'entité.
- Renvoie undefined si **l'entité avec l'ID donné n'a pas été trouvée**.

```
const newEntity = await repository.preload({id, name, firstname});
```

Exercice



- Créer une nouvelle version de la méthode updateTodo afin qu'elle puisse mettre à jour un todo dans la base de données via son id.

TypeOrm
Le patron de conception Repository
update

- Cette méthode **met à jour partiellement une entité** en se basant sur un **ensemble de critères ou** sur **l'id d'une entité**.
- Elle prend en **paramètre** le **critère d'update suivi de l'ensemble des modifications.**
- Retourne un objet de type **UpdateResult.**

```
await repository.update({ name: "Cartouche" }, { Type: "Consommable" });
// va exécuter la requête SQL : UPDATE produit SET Type = Consommable WHERE name = Cartouche

await repository.update(1, { name: "Cartouche" });
// va exécuter la requête SQL : UPDATE produit SET name = Cartouche WHERE id = 1
```

TypeOrm
Le patron de conception Repository
remove

- La méthode **remove** prend en **paramètre** une **entity** ou un **tableau d'entités** à **supprimer**.
- La **suppression** se fait à travers une **transaction**, cad qu'en cas d'échec d'un des remove la totalité est annulée via un Rollback.
- La **valeur de retour** est la **liste des entités supprimées**.

```
async deleteSection(section) {
  return await this.sectionRepository.remove(section);
}
```

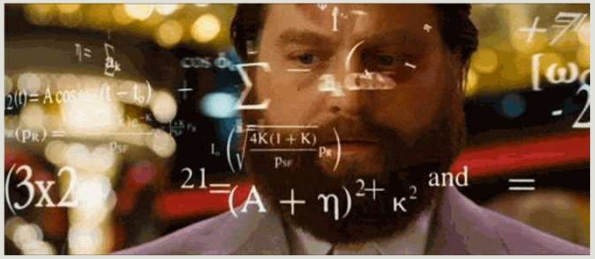
https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

TypeOrm
Le patron de conception Repository
delete

- La méthode **delete** supprime des entités en prenant en paramètre un id, des ids ou un ensemble de condition.

```
async deleteSection(id) {
  return await this.sectionRepository.delete(id);
}
async deleteSetOfSection(id1, id2, id3) {
  return await this.sectionRepository.delete([id1, id2, id3]);
}
async deleteSection2(criteria) {
  return await this.sectionRepository.delete({ designation: criteria });
}
```

Exercice



- Modifier la méthode deleteTodo afin qu'elle puisse supprimer un todo de la base de données via son id.

TypeOrm
Le patron de conception Repository
softDelete et restore

- La méthode **softDelete** supprime une entité en prenant en paramètre un id. La suppression est « soft » dans le sens ou on peut récupérer l'enregistrement supprimé via la méthode **restore en lui passant le même id**.

- Pour que cette fonction soit exécutée, vous devez avoir la colonne spéciale **@DeleteDateColumn()**

```
async softdelete(id: number) {
  return await this.sectionRepository.softDelete(id);
}
async restoreSection(id: number) {
  return await this.sectionRepository.restore(id);
}
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

TypeOrm
Le patron de conception Repository
softRemove et recover

- La méthode softReomve est une alternative à **softDelete. Elle prend en paramètre l'entité à supprimé d'une façon soft.**
- la méthode **recover permet de récupérer cet entité**.
- Pour que cette fonction soit exécutée, vous devez avoir la colonne spéciale **@DeleteDateColumn()**

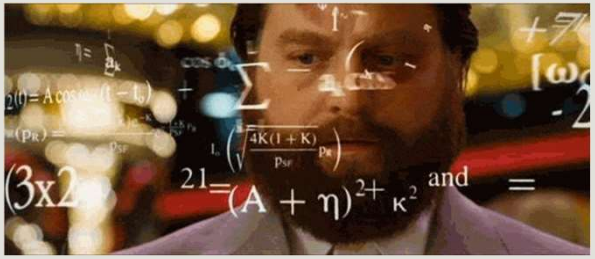
https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

TypeOrm
Le patron de conception Repository
softRemove et recover

```
async softdelete(id: number) {
const sectionToRemove = await this.sectionRepository.findOne(id);
if(! sectionToRemove)
  throw new NotFoundException(`La section d'id ${id} n'existe pas`);
  return await this.sectionRepository.softRemove(sectionToRemove);
}
async restoreSection(id: number) {
 // Problème : un softDeleted Enregistrement ne peut pas être récupéré via l'ORM
 const sectionToRecover = await this.sectionRepository.findOne(id);
  return await this.sectionRepository.recover(sectionToRecover);
}
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

Exercice



- Modifier la méthode deleteTodo afin qu'elle puisse supprimer un todo dans la base de données via son id d'une façon soft.

- Implémenter aussi la méthode permettant de restaurer votre todo.

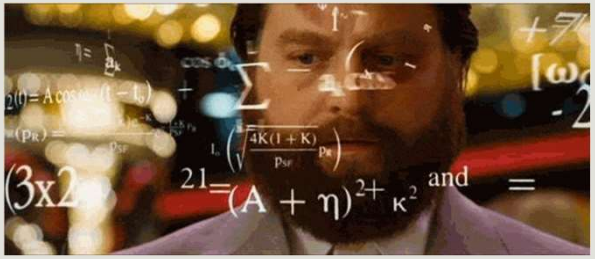
TypeOrm
Le patron de conception Repository
count

- Afin de récupérer le nombre d'enregistrement vous pouvez utiliser la méthode **count**.
- Cette fonction prend en paramètre un objet de type **FindManyOptions**
- Si l'objet est vide elle retourne le nombre de tous les enregistrements.

```
async count() {
  return await this.sectionRepository.count();
}
async countSectionByName(name: string) {
 return await this.sectionRepository.count({where:{name}})
};
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

Exercice



- Préparer une api permettant d'avoir le nombre de todo finalisé

TypeOrm
Le patron de conception Repository
incrémenter et décrémenter

- increment - Incrémente une colonne par un nombre selon un critère donné.

```
this.sectionRepository.increment(
        { name: “GL" }, “studentNumber", 1
);
```

- decrement - Décrémente une colonne par un nombre selon un critère donné.

```
this.sectionRepository.decrement(
        { name: “GL" }, “studentNumber", 1
);
```

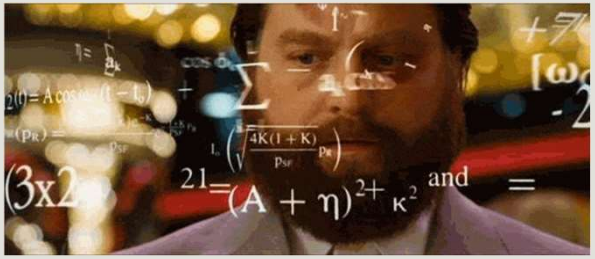
TypeOrm
Le patron de conception Repository
find

- Sélectionne les entités validant certains critères
- Cette fonction prend en paramètre un objet de type FindManyOptions
- Si l'objet est vide elle retourne tous les enregistrements (équivalent à select *).

```
async getSections() {
  await this.sectionRepository.find();
}
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

Exercice



- Faite en sorte d'avoir un endpoint permettant de récupérer l'ensemble des todos.

TypeOrm
Le patron de conception Repository
find options

- **select** : indique quels champs récupérer à travers un tableau de champs.
- **relations** : quels champs de relations charger avec l'entité principale. Ceci permet un raccourci pour le join et le leftJoinAndSelect
- **join**: version étendue de relations.
- **where** : permet de spécifier des conditions.
- **order** : permet d'ordonner les enregistrements retournés

TypeOrm
Le patron de conception Repository
find options

- **skip** : permet de spécifier à partir de quel enregistrement chercher (offset)
- **take** : permet de spécifier combien d'enregistrement récupérer (limit)
- **withDeleted**: inclure ou non les enregistrements softDeleted. Par défaut ces enregistrements ne sont pas inclus
- **cache** : permet de cacher cette requête.

https://github.com/typeorm/typeorm/blob/master/docs/find-options.md

TypeOrm
Le patron de conception Repository
find options

```
return await this.commandeRepository.find(
 {
  select: ["id", "status"], // selection le id et le status
  relations: ["details"], // récupère et affiche la relation
  join: {
   alias: "commande",
   leftJoinAndSelect : {
    user: "commande.user"
   }
  },
 // un AND si même objet, un OR si plusieurs objets
  where: [{status: CommandeStatusEnum.pending}],
  order: {id: 'DESC'},
  skip: 2, // offser : à partir de quel enregistrement lire
  take: 2, // limit : combien d'enregistrement lire
  cache: true
 }
```

https://github.com/typeorm/typeorm/blob/master/docs/find-options.md

TypeOrm
Le patron de conception Repository
find options avancées

- **not** : pour la négation title: Not("About #1")
- **LessThan** : tous les enregistrements inférieur (lessThan une propriété)

{age: **LessThan(20)**}

- **LessThanOrEqual**: tous les enregistrements inférieur ou égale

{age: **LessThanOrEqual(20)**}

- **MoreThan** : plus grand

{age: **MoreThanOrEqual(20)**}

- **MoreThanOrEqual** : plus grand ou égale

TypeOrm
Le patron de conception Repository
find options avancées

```
import { MoreThan} from 'typeorm';
```

```
return await this.commandeRepository.find(
  {  where: { createdAt: MoreThan("2020-06-17") }}
)
```

https://github.com/typeorm/typeorm/blob/master/docs/find-options.md

TypeOrm
Le patron de conception Repository
find options avancées

- **Equal** : égale
- **Like** : permet d'exécuter l'opérateur like

{name: **Like("%mohamed%")**}

- **Between** : les enregistrements dont une propriété est dans un intervalle.
- **In** : les enregistrements dont une propriété est dans un ensemble.
- **IsNull :** Sélectionne les champs Null

Remarque : Toutes les méthodes doivent être importées de typeorm

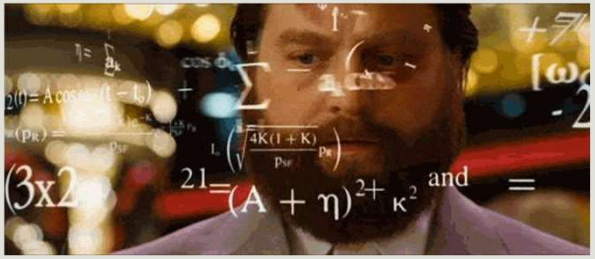
TypeOrm
Le patron de conception Repository
find options avancées

```
return await this.commandeRepository.find(
 where: {
        { status : Like("%En%") }
 }
);
```

```
return await this.commandeRepository.find(
 {
  where: { status : In([
          CommandeStatusEnum.shipping,   CommandeStatusEnum.pending
  ])
 }}
);
```

https://github.com/typeorm/typeorm/blob/master/docs/find-options.md

Exercice



- Modifier l'api get de sorte qu'il puisse ou non prendre en entrée (via des query parameters) une chaine et un statut et retournant l'ensemble des todos dont la description ou le nom contiennent la chaine **ou** dont le statut est celui recherché.
- Créer un DTO permettant de récupérer le couple statut, critère.

TypeOrm
Le patron de conception Repository
findAndCount

- Sélectionne les entités validant certains critères et retourne le nombre d'enregistrements
- Cette fonction prend en paramètre un objet de critères
- Si l'objet est vide elle retourne tous les enregistrements (équivalent à select *).

```
async getSections() {
  await this.sectionRepository.findAndCount();
}
```

```
async getSections() {
  await this.sectionRepository.findAndCount({designation: "GL"}
);
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

TypeOrm
Le patron de conception Repository
findByIds

- Recherche plusieurs entités par identifiants
- Prend en paramètre un tableau d'ids.

```
async getSections() {
  await this.sectionRepository.findByIds([1,2,3]);
}
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

TypeOrm
Le patron de conception Repository
findOne

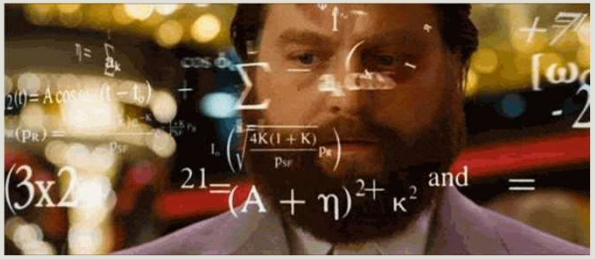
- Recherche la première entité et prend en paramètre un objet de type **FindOneOptions**.

```
async getSections() {
  await this.sectionRepository.findOne({where: {id}});
}
```

```
async getSections() {
  await this.sectionRepository.findOne(
            {where: {designation: "GL"}}
);
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

Exercice



- Créer le endpoint permettant de récupérer un todo par son id.

- Gérer le cas ou le todo n'existe pas

TypeOrm
Le patron de conception Repository
query

- vous permet d'exécuter une requête sql.

```
async getSections() {
  await this.sectionRepository.query("select * from section");
}
```

https://github.com/typeorm/typeorm/blob/master/docs/repository-api.md

TypeOrm
Repository
QueryBuilder

- QueryBuilder est l'une des fonctionnalités les plus puissantes de TypeORM.

- Il permet de créer des requêtes SQL, de les exécuter et d'obtenir des entités automatiquement transformées.

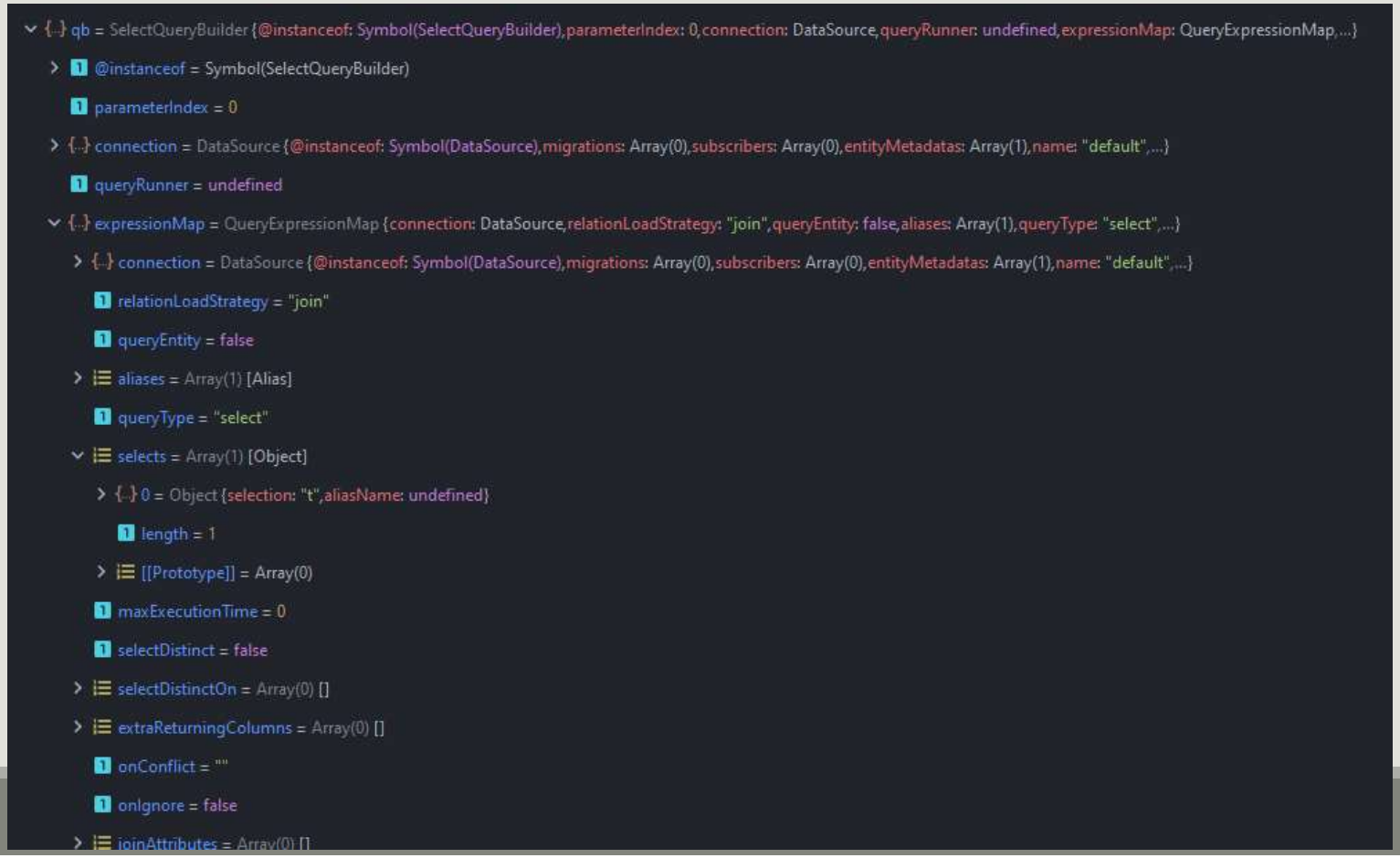
- Vous permet donc de personnaliser vos requêtes.

TypeOrm
Repository
QueryBuilder

- Afin de récupérer le queryBuilder à partir de votre Repository, vous devez utiliser la méthode **createQueryBuilder**.
- Cette méthode prend en paramètre **optionnel l'alias** de la **table représentant votre entité**.
- queryBuilder vous offre plusieurs méthodes vous permettant de créer une requête.
- Par défaut, et dès sa création, le queryBuilder vous génère la requête « select * from tableName »

```
const queryBuilder = this.sectionRepository.createQueryBuilder("section");
```

TypeOrm Repository QueryBuilder



TypeOrm QueryBuilder getMany et getOne

- Afin de récupérer le résultat de la requête vous utilisez la méthode **getMany()** pour récupérer un **ensemble d'entité**.
- Pour récupérer **une entité** on utilise la méthode **getOne()**.

```
const results = this.sectionRepository.createQueryBuilder("section").getMany();
```

```
const result = this.sectionRepository.createQueryBuilder("section").getOne();
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

TypeOrm QueryBuilder Select From

- **select** : vous permet de spécifier les éléments à sélectionner. Elle prend en paramètre une chaine de caractère ou un tableau des champs à sélectionner.

```
queryBuilder.select("section.designation, section.createdAt");
```

```
queryBuilder.select([
 "section.id",
 “section.designation",
 "section.createdAt"]);
```

- **from** : vous permet de spécifier la ou les tables que vous allez requêter. Elle prend en paramètre l’entité représentant la table, et l’alias de cette table.

```
queryBuilder.from(SectionEntity, "section");
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

TypeOrm QueryBuilder where

- Afin d'ajouter une condition dans votre requête, il faut utiliser la méthode **where** de votre queryBuilder.
- Cette méthode prend en **paramètre** une **chaine de caractère** suivi d'un **objet contenant l'ensemble des paramètres**.
- Dans la chaine de caractère pour **identifier un paramètre précéder le de ':'** suivi de **son nom**.
- L'**objet de paramètre** contiendra le **nom** du paramètre et sa **valeur**.

```
queryBuilder.where("section.designation = :designation", {designation: designation});
```

TypeOrm QueryBuilder where

- Vous pouvez concaténer vos paramètre directement. Cependant c'est une **faille de sécurité** car vous laissez le champs libre aux **SQL INJECTION. NE LE FAITE PAS.**
- Le tableau de paramètre est un raccourci de l'appel de la méthode **setParameter** et **setParameters**

```
queryBuilder.where("section.designation = :designation")
            .setParameter('designation', designation);
```

TypeOrm QueryBuilder where IN

- Si vous avez un **ensemble de choix** dans lequel chercher, vous pouvez utiliser l'opérateur **IN** dans votre where.
- N'oubliez pas les **… devant le nom du paramètre** pour indiquer que ca va être un tableau.

```
queryBuilder.where(
        "section.designation IN :...designations",
        {
                designations: ['GL', 'RT', 'IIA', 'IMI']
        }
);
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

TypeOrm QueryBuilder andWhere et orWhere

- Vous pouvez combiner plusieurs where avec les méthodes **andWhere** et **orWhere.**
- Ceci revient à faire un where a and b ou where a or b.

```
queryBuilder.where("section.designation IN (:...designations)",
                   {designations: ['GL', 'RT', 'IIA', 'IMI']})
            .andWhere('sections.id > 10')
            .orWhere('section.createdAt < 10062020 ')
```

```
SELECT `section`.`createdAt` AS `section_createdAt`, `section`.`id` AS `section_id`,
`section`.`designation` AS `section_designation`
FROM `section_entity` `section`
WHERE (`section`.`designation` IN ? AND sections.id > 10 OR `section`.`createdAt` < 10062020 )
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

TypeOrm QueryBuilder where and Brackets

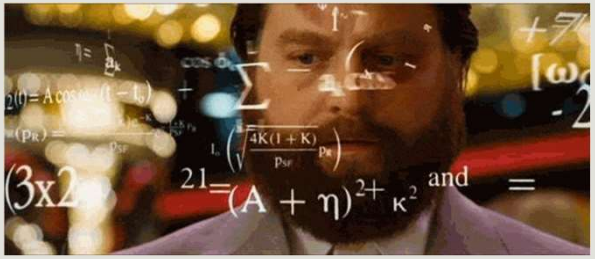
- Afin de manipuler vos requêtes, vous avez généralement besoin de définir des parties en utilisant les parenthèses. Pour ce faire, TypeORM vous offre la classe **Brackets**

```
queryBuilder.where("section.designation IN :...designations", {designations: ['GL', 'RT', 'IIA', 'IMI']})
        .andWhere(new Brackets(
         qb => {
         qb.where('sections.id > 10')
           .orWhere('section.createdAt < 10062020')}))
```

```
SELECT `section`.`createdAt` AS `section_createdAt`, `section`.`id`
AS `section_id`, `section`.`designation` AS `section_designation`
FROM `section_entity` `section`
WHERE (
`section`.`designation` IN ?
 AND (sections.id > 10 OR `section`.`createdAt` < 10062020)
)
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

Exercice



- Modifier l'api get de sorte que maintenant on puisse avoir les Todo dont le name **ou** description contiennent la chaine passée en paramètre **et** ayant le statut passé en paramètre.
- Les deux critères de recherche restent optionnels.

TypeOrm QueryBuilder orderBy

- Afin d'ordonner le résultat de vos requêtes, utiliser la méthode **orderBy**
- orderBy prend en paramètre le champ et l'ordre ('ASC' par défaut 'DESC' second choix).
- **addOrderBy** permet d'ajouter d'autres critères d'ordonnancement.
- Vous pouvez aussi passer un tableau à la méthode orderBy.

TypeOrm QueryBuilder limit, offset

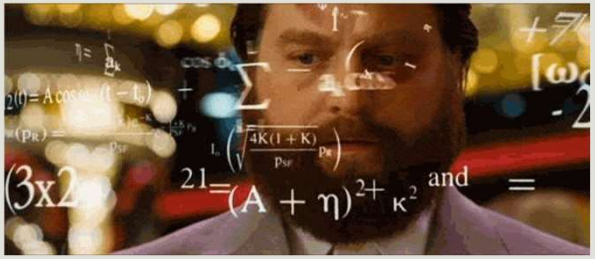
- La méthode **take** permet de spécifier le **nombre d'enregistrement**.
- **skip** permet de spécifier **à partir de quel enregistrement commencer** à sélectionner

```
queryBuilder.select([
  "section.id",   "section.designation", "section.createdAt"
])
  .take(10)
  .skip(10)
;
```

```
SELECT `section`.`createdAt` AS `section_createdAt`, `section`.`id`
AS `section_id`, `section`.`designation` AS `section_designation`
FROM `section_entity` `section`
WHERE `section`.`deletedAt` IS NULL LIMIT 10 OFFSET 10
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

Exercice



- Ajouter le traitement nécessaire pour paginer votre fonction getAll.

- Utiliser les QB et essayer de faire une fonction générique qui permet à chaque besoin d'une pagination de la réutiliser.

TypeOrm QueryBuilder getRawOne et getRawMany

- Il existe **deux types de résultats** que vous pouvez obtenir à l'aide du queryBuilder :
  - **Les entités**
  - **Les résultats bruts**.
- Dans certains cas d'utilisations, vous avez besoins de données spécifiques (statistiques par exemple). Ces données ne sont pas une entité, elles s'intitulent 'données brutes'.
- Pour obtenir des données brutes, vous utilisez getRawOne et getRawMany.

```
const { count } = await this.sectionRepository
  .createQueryBuilder("section")
  .select("count(section.createdAt)", "count")
  .where("section.createdAt > :date",
                { date: '02/02/2020' }
  )
  .getRawOne()
;
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

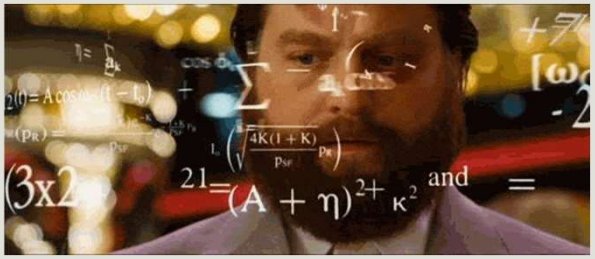
TypeOrm
QueryBuilder
Having et GroupBy

- **GroupBy** pour une clause **groupBy**
- **Having** pour conditionner un **groupBy** et **andHaving** si vous avez plusieurs conditions.
- **orHaving** est l'équivalent de orWhere

```
const qb = this.produitRepository.createQueryBuilder("produit");
return qb.select('designation , sum(produit.price) as somme')
      .groupBy('designation')
      .having("somme > 500")
      .getRawMany();
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

Exercice



- Créer une api Stats qui vous retourne pour chaque status le nombre de todo crée.
- Elle peut aussi, en option, prendre deux dates et vous retourner les stats crées dans cet intervalle la.

Typeorm Relations

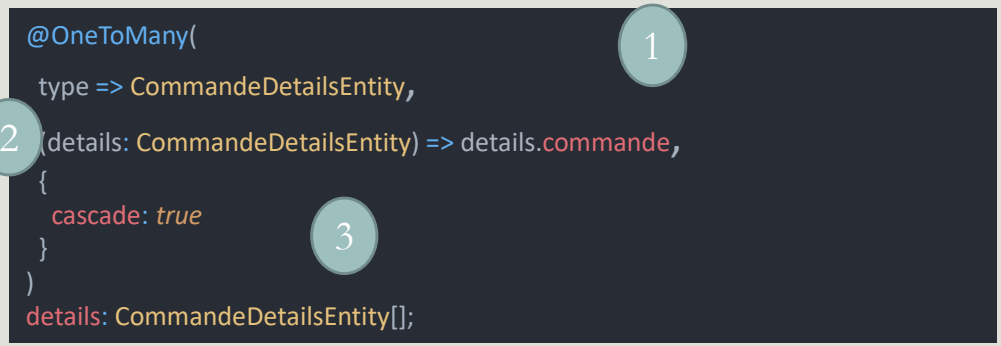
Les entités de la BD présentent des relations d'association :

- A OneToOne B : à une entité A on associe une entité de B et inversement
- A ManyToOne B : à une entité B on associe plusieurs entité de A et à une entité de A on associe une entité de B
- A ManyToMany B : à une entité de A on associe plusieurs entité de B et inversement

Typeorm Relations

Chaque relation prend 3 paramètres.

1. ➢ **typeFunctionOrTarget** : qui est une chaine ou une **fonction fléchée** et qui **retourne l'entité cible**.
2. ➢ **inverseSide**: qui est de type **fonction fléchée** qui prend **en paramètre l'entité avec la quelle vous êtes en relation** et qui **retourne le champs correspondant à l'entité cible**. Ce champ est **optionnel**. Il est utilisé en cas de **relation bidirectionnelle**.
3. ➢ **options** : Objet d'option permettant **d'enrichir votre relation**



Typeorm
Relations
Les options

- **eager**: boolean - Si la valeur est **true**, la relation sera toujours **chargée avec l'entité principale** lors de l'utilisation des méthodes **find * ou QueryBuilder sur cette entité**. Plus simplement, si un champ représente une **relation** et que vous voulez qu'il **apparaisse au moment de lancer la requête find** alors il faut un **eager loading**. **Sinon**, vous allez faire du **lazy loading**. Cette option n'est activable que pour les **relations bidirectionnelles**.

- **cascade**: booléen | ("**insert**" | "**update**") **[]** - Si la valeur est true, l'objet associé sera inséré et mis à jour dans la base de données. **Vous pouvez également spécifier un tableau d'options en cascade**.

```
{ cascade: ["insert", "update"] }
```

Typeorm
Relations
Les options

- **onDelete**: "RESTRICT" | "CASCADE" | "SET NULL" - spécifie comment la clé étrangère doit se comporter lorsque l'objet référencé est supprimé. Elle **doit être gérée au niveau du ManyToOne**
- **primary**: boolean - Indique si la colonne de cette relation sera une clé primaire ou non.
- **nullable**: boolean - Indique si la colonne de cette relation est nullable ou non. **Par défaut, il est nullable**.

Typeorm
Relations
OneToOne

- La relation **OneToOne** est une relation où A ne contient qu'une seule instance de B et B ne contient qu'une seule instance de A. Prenons par exemple les entités Utilisateur et Profil. L'utilisateur ne peut avoir qu'un seul profil et un seul profil n'appartient qu'à un seul utilisateur.
- La relation peut être bidirectionnelle ou unidirectionnelle.
- Dans une relation unidirectionnelle, choisissez la ou vous voulez avoir l'information de relation, créer une propriété et annoter la avec **@OneToOne**.

Typeorm
Relations
OneToOne

- L'annotation **@OneToOne** prend en paramètre une fonction fléchée qui retourne l'Entité correspondante à la relation.
- Afin de spécifier dans quel entité vous voulez mettre la clé secondaire, ajouter l'annotation **@JoinColumn().** Ceci permettra à l'ORM de savoir ou mettre cette clé vu que dans une relation OneToOne le choix est laissé au concepteur de la base.

Typeorm Relations OneToOne

```
@Entity()
export class User {

  @PrimaryGeneratedColumn()
  id: number;

  @Column()
  name: string;

  @OneToOne(type => Profile)
  @JoinColumn() // ici on aura dans la base de données une clé secondaire profile.
  profile: Profile;
}
```

https://github.com/typeorm/typeorm/blob/master/docs/relations.md

Typeorm
Relations
@JoinColum

- Permet de spécifier quel partie de la relation contient la clé secondaire. Elle est **optionnelle** pour le **ManyToOne** mais **obligatoire** pour le **OneToOne**
- Permet de spécifier le **nom de la colonne de jointure** et **son nom** dans la base de données.
- Généralement la jointure se fait avec la clé primaire. Mais si vous voulez joindre un autre champ, utiliser la clé **referencedColumnName**

```
@ManyToOne(type => Category)
@JoinColumn(
  { name: "category_id", referencedColumnName: "id" },
)
category: Category;
```

Typeorm
Relations
ManyToOne et OneToMany

- La relation **ManyToOne** est une relation où A ne contient qu'**une** seule instance de B et B contient **plusieurs** instances de A. Prenons par exemple les entités Utilisateur et Photo. Photo ManyToOne Utilisateur. L'utilisateur peut avoir plusieurs photos et chaque photo apparient à un utilisateur.

- La relation peut être **bidirectionnelle** ou **unidirectionnelle**. Dans le cas de la relation **bidirectionnelle** on ajoute la relation **OneToMany** dans **l'autre entité**.

Typeorm
Relations
ManyToOne et OneToMany

```
@ManyToOne(
  () => UserEntity,
  (user: UserEntity) => user.commandes,
  {eager: true}
)
user: UserEntity;
```

```
@OneToMany(
  () => CommandeEntity,
  (commande: CommandeEntity) => commande.user
)
commandes: CommandeEntity[];
```

https://github.com/typeorm/typeorm/blob/master/docs/relations.md

Typeorm
Relations
ManyToMany

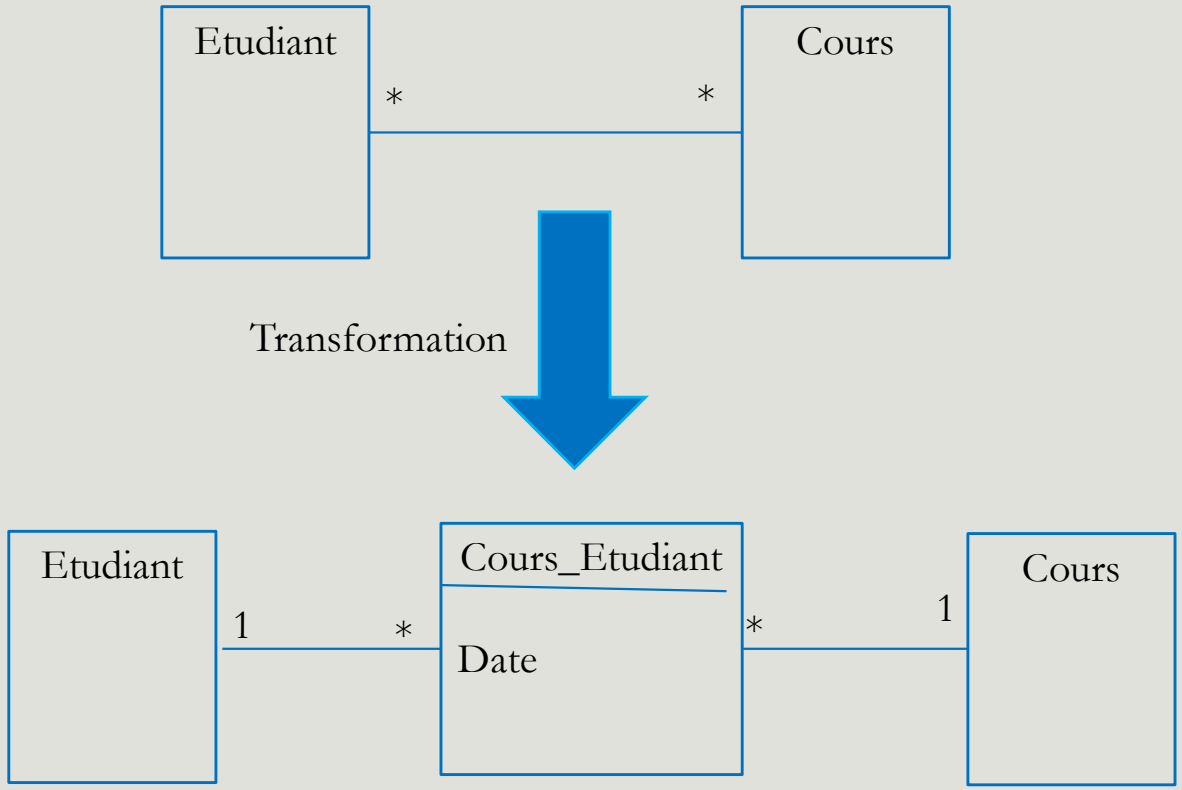
- La relation **ManyToMany** est une relation où A contient **plusieurs** instances de B et B contient **plusieurs** instances de A.
- Cette relation implique la **création automatique d'une table contenant les id des deux tables en relations**.
- La relation peut être bidirectionnelle ou unidirectionnelle.
- L'annotation **@JoinTable() est OBLIGATOIRE et dans un seul coté de la relation.** Vous devez la mettre dans l'entité maitre de la relation.

Typeorm Relations @JoinTable

- Utilisé pour les relations **ManyToMany**
- Permet de **personnaliser la table intermédiaire** crée ainsi que les noms des colonnes de référence.

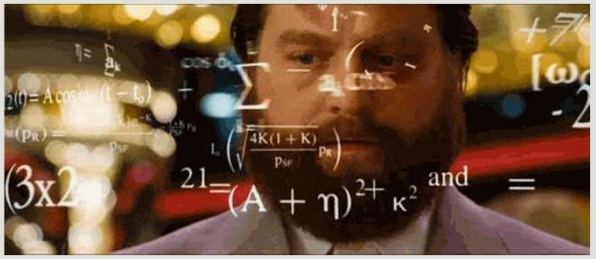
```
@ManyToMany(type => Category)
@JoinTable({
 name: "question_categories", // nom de la table à générer
 joinColumn: {
  name: "question", // nom du champ représentant l'entité actuelle
  referencedColumnName: "id“
 },
 inverseJoinColumn: {
  name: "category", // nom du champ représentant l'entité en relation avec cet entité
  referencedColumnName: "id"
 }
})
categories: Category[];
```

Typeorm
Relations
ManyToMany : Cas particulier



https://github.com/typeorm/typeorm/blob/master/docs/relations.md

Exercice

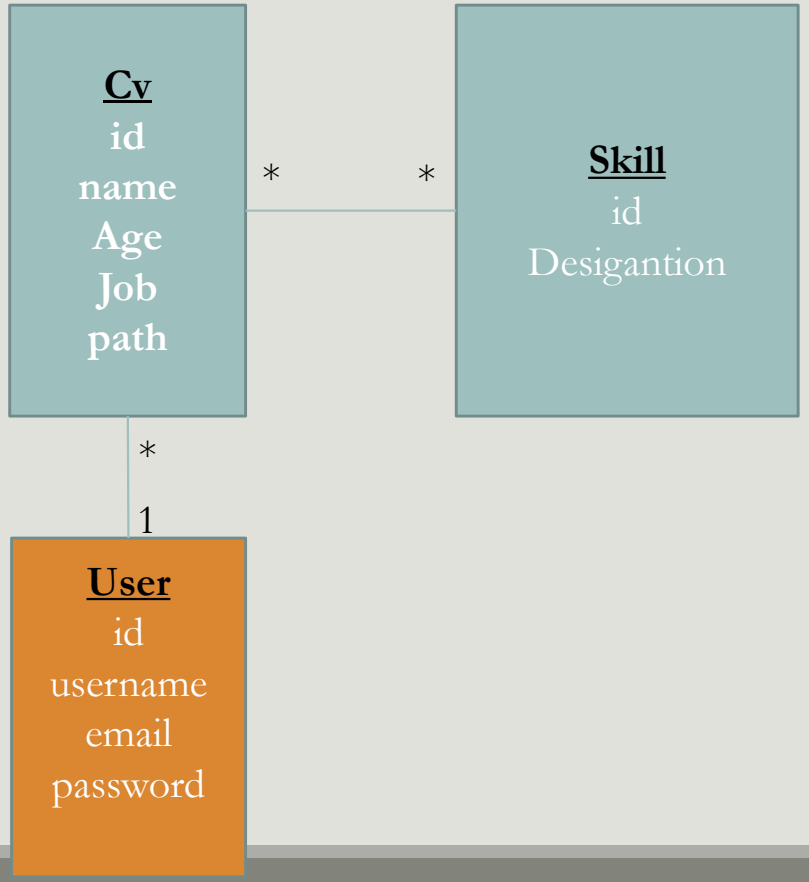


- Nous voulons reproduire le schéma relatif à un petit gestionnaire de cvs.
- Créer les modules, contrôleurs, et services et entités relatives à ce schéma ainsi que les relations qui y sont associées.

Astuce : vous pouvez utiliser la commande :

nest generate resource

- Cette commande génère non seulement tous les blocs de construction NestJS (module, service, classes de contrôleur), mais également une classe d'entité, des classes DTO ainsi que les fichiers de test (.spec).
- Vous pouvez aussi penser à la création d'un CrudService générique



Seed de la base de données
Standalone applications

- Vous pouvez créer vos applications Nest de plusieurs façons : Une application Web, des micro services mais aussi une application Standalone indépendante du contexte Web.
- Une application Nest Standalone est une couche sur le Contenaire IOC de Nest.
- Ceci vous permet donc de récupérer n'importe quelle instance exporté par les modules que vous importer.

https://docs.nestjs.com/standalone-applications#standalone-applications

Seed de la base de données
Standalone applications

- Afin de créer une standalone application, utiliser la méthode **createApplicationContext** de votre **NestFactory**.
- Passer à cette méthode votre Module
- Pour récupérer le service que vous voulez utiliser la méthode **get** et passer lui la classe souhaitée.

```
async function bootstrap() {
  const app = await NestFactory.createApplicationContext(AppModule);
            // Todo :  Do What you want
  await app.close();
}
bootstrap();
```

https://docs.nestjs.com/standalone-applications#standalone-applications

Seed de la base de données
Standalone applications

- Afin d'exécuter votre application, vous devez lancer la commande **ts-node** suivi du **path de votre fichier**.
- Vous pouvez créer une **script** au niveau de votre fichier **package.json** ce qui vous facilitera la tache.

```
"scripts": {
  //...
  "seed:cvs": "ts-node src/commands/cv.seeder.ts"
},
```

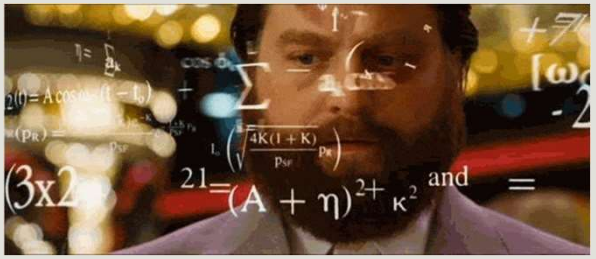
https://docs.nestjs.com/standalone-applications#standalone-applications

Seed de la base de données

- Pour la génération des donnés fictives, vous pouvez utiliser plusieurs bibliothèques.
- L'une d'elle est la bibliothèque ngneat :

https://ngneat.github.io/falso/docs/getting-started

Exercice



- Créer une standalone application permettant le seed de votre Base de données.

TypeOrm
QueryBuilder
Inner et left joins

- Afin de faire une jointure « left » ou « inner » join vous pouvez utiliser les méthodes leftJoin et innerJoin.
- Le premier paramètre est le champ de la relation et le second est l'entité avec laquelle est associée la relation.

```
this.commandeRepository.createQueryBuilder("commande")
 .leftJoin("commande.user", "user")
 .select(["commande.id", "commande.status","user.username"])
 .getRawMany();
```

```
SELECT `commande`.`id` AS `commande_id`,
`commande`.`status` AS `commande_status`, `user`.`username` AS `user_username`
FROM `commande` `commande`
LEFT JOIN `user` `user`
ON `user`.`id`=`commande`.`userId`
```

https://github.com/typeorm/typeorm/blob/master/docs/select-query-builder.md

TypeOrm
Cache

- Vous pouvez cacher le résultat de vos requêtes avec TypeOrm.
- Afin d'activer le cache, vous devez aller dans la configuration de TypeOrm et ajouter l'option **cache** et la mettre à **true**.
- Ensuite, dans le queryBuilder ou dans le repository, déclencher le cache.

```
TypeOrmModule.forRoot(
 {
  type: 'mysql',
  host: process.env.DB_HOST,
  port: 3306,
  username:root,
  password:,
  database: shop,
  ...,
  synchronize: true,
  cache: true
 }
)
```

```
return this.commandeRepository.find({cache: true});
```

```
this.commandeRepository.createQueryBuilder("commande")
 .cache(true);
```

https://github.com/typeorm/typeorm/blob/master/docs/caching.md

TypeOrm Cache

- Vous pouvez aussi définir la durée du cache.
- TypeOrm créera ensuite une table « query-result-cache » dans laquelle il stockera les différentes requête et les informations les concernant.

```
return await this.commandeRepository.find({cache: 6000});
```

```
this.commandeRepository.createQueryBuilder("commande")
  .cache(6000);
```

```
TypeOrmModule.forRoot(
  {
    type: 'mysql',
    host: process.env.DB_HOST,
    port: 3306,
    username:root,
    password:,
    database: shop,
    ...,
    synchronize: true,
    cache: {
      duration: 5000
    }
  }
)
```

https://github.com/typeorm/typeorm/blob/master/docs/caching.md

Authentification

- Le processus d'authentification consiste à authentifier un utilisateur de la plateforme suivant ces identifiants.
- Vous devez donc avoir deux fonctionnalités. L'une permanant à l'utilisateur de s'inscrire et l'autre de s'authentifier.
- Commencez par créer une table qui vous permet de stocker vos utilisateurs.

Authentification

- Les propriétés de la classe utilisateur dépendent de vos besoins. Généralement vous avez besoin d'avoir :
  - Un identifiant unique: username, email ou les deux.
  - Un mot de passe crypté
  - Un ou plusieurs rôles
  - …

Authentification
Crypter un mot de passe

- Afin de crypter un mot de passe, vous pouvez utiliser la bibliothèque **bcrypt**.
- Pour installer bcrypt utiliser la commande npm i bcrypt @types/bcrypt.
- Une fois installé, vous aurez besoin de créer un salt afin de sécuriser vos mots de passes. Pour ce faire, utiliser la méthode **genSalt()** de bcrypt.
- bcrypt vous offre aussi la méthode **hash** qui vous permettra de hacher vos mots de passe. Elle prend en paramètre le mot de passe à hacher et le salt.

```
import * as bcrypt from "bcrypt";
//...
const  salt = await bcrypt.genSalt();

const password = await bcrypt.hash (password, salt);
```

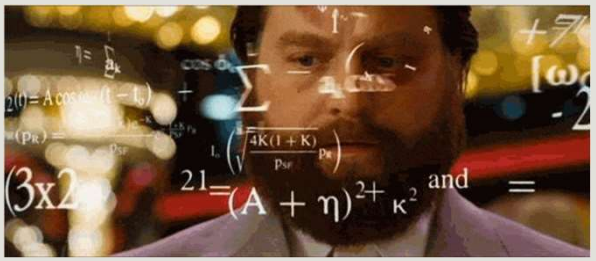
https://www.npmjs.com/package/bcrypt

Authentification
Authentifier vos utilisateurs

Une fois vos utilisateurs inscrits, vous devez leur permettre de s'authentifier.

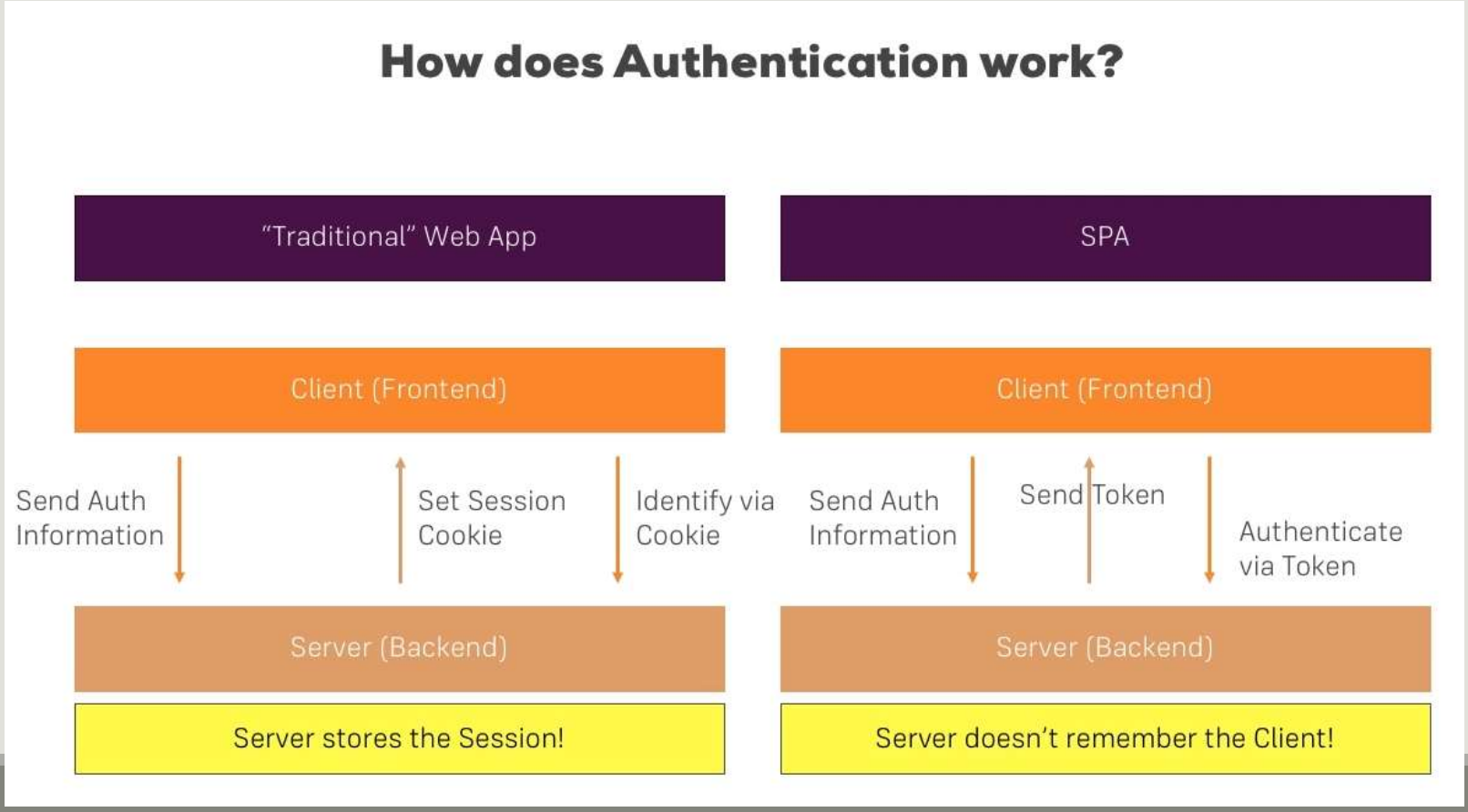
1. Récupérer les données d'authentification de vos utilisateurs.
2. Vérifier que l'utilisateur existe via son identifiant.
3. Récupérer le mot de passe envoyé et vérifier qu'il correspond bien au mot de passe sauvegardé dans la base de données.
4. Si c'est le cas, authentifier votre utilisateur, sinon retourner une UnauthorizedException.

Exercice

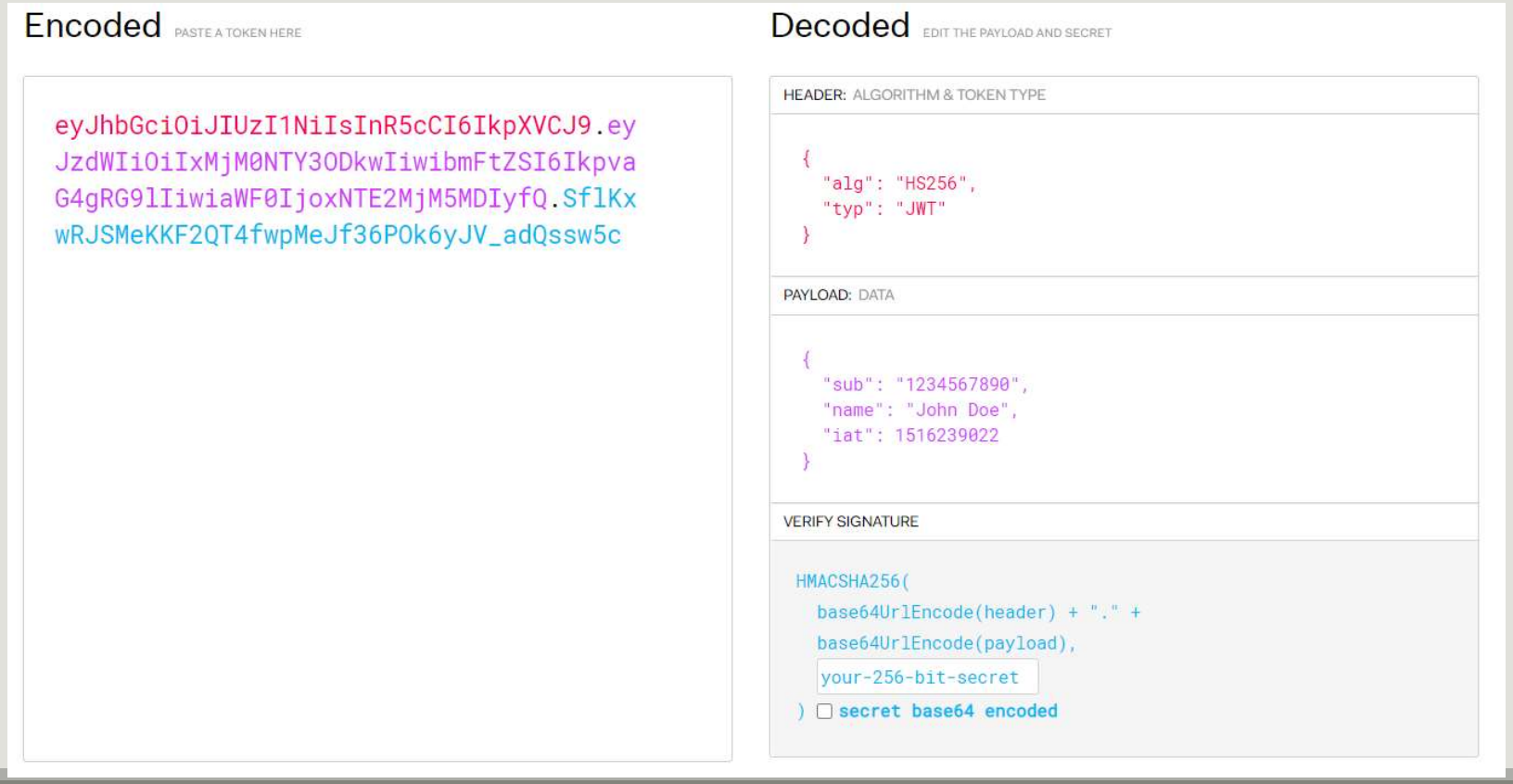


- Créer un Module AuthModule
- Mettez à jour votre entité user avec les informations suivantes :
  - username (unique)
  - password
  - email (unique)
  - role
- Créer les fonctionnalité d'inscription et de login. N'oublier pas de définir vos DTO's.

Authentification
Utilisation des Tokens



Authentification JWT



https://jwt.io/

Authentification
Passport

- NestJs propose d'utiliser la **bibliothèque Passport** pour la partie Authentification.
- C'est la bibliothèque Node.js la plus populaire pour cet aspect.
- Nest propose donc un module pour pouvoir intégrer facilement cette bibliothèque.
- Elle offre plusieurs stratégies d'authentification telle que l'utilisation de JWT.

Authentification
Passport JWT
Configuration

Afin de configurer Passport vous devez passer par les étapes suivantes :

- Installer via npm les différentes bibliothèques nécessaires à savoir :
  - @nestjs/passport
  - passport
  - @nestjs/jwt
  - passport-jwt
  - @types/passport-jwt

Authentification
Passport JWT
Configuration

- Importer le module **PassportModule** dans votre module d'authentification
- Appeler la méthode **register** et passez y un objet content une clé **defaultStrategy**. Cette clé contiendra la stratégie que vous voulez utiliser. Dans notre cas ca sera jwt.

```
imports: [
  PassportModule.register({defaultStrategy: 'jwt'}),
],
```

Authentification Passport JWT Configuration

- Importer le module **JwtModule** qui vous permet d'accéder au service **JwtService**. Ce service vous permettra de manipuler vos jwt.
- En important le module, appeler la méthode **register** qui prend en paramètre un objet d'options..

```
imports: [
  JwtModule.register({
    secret: process.env.SECRET,
    signOptions: {
      expiresIn: 3600,
    }
  }),
  PassportModule.register({defaultStrategy: 'jwt'}),
  TypeOrmModule.forFeature([
    UserEntity
  ])
]
```

Authentification Passport JWT Configuration

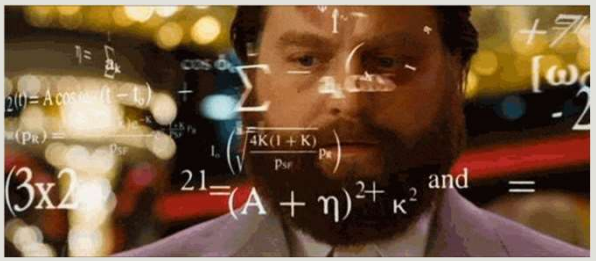
- L'option nécessaire est l'option **secret** qui prend la clé secrète utilisée pour le hachage du token.
- Il y a aussi la propriété **signOptions** qui prend un objet contenant la propriété **expiresIn** qui représente en nombre de seconde la durée de validité du Token.

```
imports: [
  JwtModule.register({
    secret: process.env.SECRET,
    signOptions: {
      expiresIn: 3600,
    }
  }),
  PassportModule.register({defaultStrategy: 'jwt'}),
  TypeOrmModule.forFeature([
    UserEntity
  ])
]
```

Authentification
Passport JWT
JwtService

- Le module **JwtModule** vous offre le service JwtService. Ce service vous offre une panoplie de fonctionnalités qui vous servent à manipuler vos tokens.
- Le **JwtService** utilise le json web token. Il offre les fonctions suivantes :
- **sign** (payload: string | Object | Buffer, options ?: SignOptions): string, elle permet de générer le Token à partir du payload que vous lui passez.
- **verify**, prend en paramètre un token et le vérifie.
- **decode** prend en paramètre un token et le décode.

Exercice



- Mettez en place Passport
- Modifier votre fonction de login en créant votre JWT et en le renvoyant à l'utilisateur.
- Vérifier la validité de votre token dans le site jwt.io

Authentification
Passport JWT
Jwt Strategy

- La dernière étape consiste à définir la **stratégie de gestion du JWT Token**. Ceci est fait en créant une **classe** qui va étendre la class de base PassportsStrategy(Strategy).
- Cette classe **devra être providé** au niveau de votre module d'authentification.
- L'intérêt de cette manœuvre est de spécifier à Passeport **comment il doit interagir avec le token** en lui indiquant par exemple quel secret utiliser pour décoder le token. Que faire avec le token et comment le récupérer.

Authentification
Passport JWT
Jwt Strategy

- Le deuxième intérêt réside en l'implémentation de la méthode validate. Cette méthode est appelé à chaque fois qu'on intercepte le token pour le valider. Vous devez spécifier dans cette fonction comment valider le token. Elle prend en paramètre, le payload.

- Une fois validé, **ce que vous retourner avec cette méthode** est **automatiquement injecté dans la requête sous une propriété user**. Donc si vous voulez avoir votre user, il suffit de le retourner et vous pouvez le récupérer dans la requête dans l'ensemble de vos contrôleurs.

- **Remarque : tout ceci n'est déclenché qu'en cas de besoin d'authentification**

Authentification Passport JWT Jwt Strategy

```
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      secretOrKey: process.env.SECRET
    });
  }
  // La payloadInterface sert à typer votre code à vous de la créer selon votre payload
  async validate(payload: PayloadInterface) {
    // validate jwt ce qu'on retourne ici ca va etre injecté dans la requete

    const user =  //find user ;
    if (!user) {
      throw new UnauthorizedException();
    } return user;
  }
}
```

NestJs Guards

- Les **Guards** sont des classes ayant un rôle unique : **Vérifier si une requête doit être gérée par un Controller ou non** ?
- Un use case très explicite consiste à vérifier si celui qui demande la ressource a le droit d'y accéder ou non.
- Un **guard** est une classe qui **implémente l'interface CanActivate**. Ceci implique qu'il doit **implémenter** la méthode **canActivate**
- La fonction prend en paramètre **l'ExecutionContext**

NestJs Guards

- Le Guard peut être appliqué sur 3 portées (scope) comme pour les pipes, les filtres et les intercepteurs, à savoir : méthode, contrôleur et globale.
- Afin **d'appliquer** le Guard utiliser le décorateur **@UseGuards()**

NestJs Guards Custom Guard

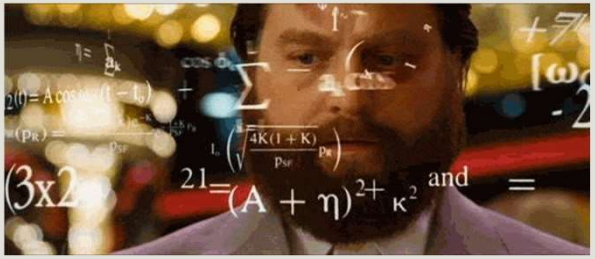

```
import { CanActivate, ExecutionContext, Injectable } from '@nestjs/common';
import { Observable } from 'rxjs';

@Injectable()
export class AdminGuard implements CanActivate {
  canActivate(
    context: ExecutionContext,
  ): boolean | Promise<boolean> | Observable<boolean> {
                // Todo Guard traitement
                return  true;
  }
}
```

Authentification et Autorisation
NestJs Guards
Passport

- Passport vous offre un guard vérifiant si un utilisateur est authentifié ou non: c'est le AuthGuard
- Ce guard prend en paramètre la stratégie à appliquer.
- Pour l'appliquer, utiliser le décorateur @UseGuards qui prend en paramètre un ou plusieurs guards
- Le guard peut être appliqué sur une route ou sur un Controller

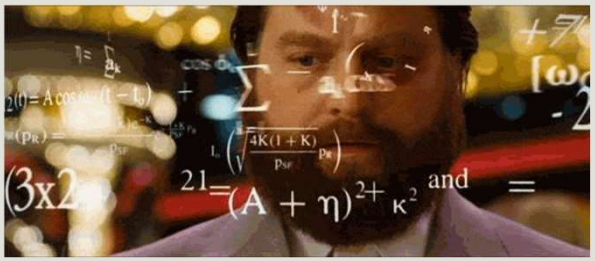
```
@UseGuards(AuthGuard('jwt'))
@Post()
@UsePipes(ValidationPipe)
creatTask(
  @Body() createTaskDto: CreateTaskDto
  ): Promise<Task> {
  return this.tasksService.createTask(createTaskDto);
}
```

Exercice



- Mettez en place Passport
- Modifier votre fonction de login en créant votre JWT et en le renvoyant à l'utilisateur.
- Protéger les fonctionnalités qui doivent être accessible uniquement pour les personnes connectés et Tester les.

Exercice



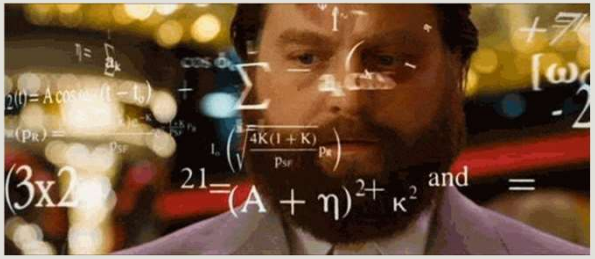
- Créer votre propre Guard appelé AdminGuard.
- Ce guard doit ne donner l'accès qu'aux admins de l'application.

Les Décorateurs de paramètres

- Afin de nous faciliter la tâche, et lorsque vous avez besoin d'extraire des paramètres de vos requêtes, NestJs nous propose la méthode **createParamDecorator**. Elle vous permet de créer des décorateur de paramètres comme @Body, @Param …
- **createParamDecorator** prend en paramètre une callback function qui prend en paramètres les data passées au décorateur et l'Execution context.
- Retourner la valeur que vous souhaitez

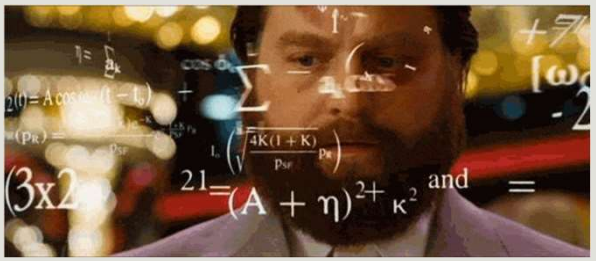
```
export const GetMyData = createParamDecorator((data, ctx: ExecutionContext): User => {
        const req = ctx.switchToHttp().getRequest();
        // Todo return what do you want
});
```

Exercice



- Créer un décorateur qui vous permet de récupérer le user de votre request.

Exercice



- Faite le nécessaire pour affecter le user connecté au propriétaire du Cv
- Faite en sorte que lorsque le user connecté est un Admin il peut récupérer tous les Cvs. Lorsque c'est un user il ne peut voir que ses propres Cvs.

Autorisation

- L'autorisation est un processus permettant d'autoriser un utilisateur à accéder à une ressource selon son rôle.

Le processus d'authentification suit deux étapes.

1- Lors de l'authentification, l'utilisateur est associé à un ensemble de rôles.

2- Lors de l'accès à une ressource, on vérifie si l'utilisateur a le rôle nécessaire pour y accéder.

Autorisation
RBAC

- L'une des manières de gérer les autorisations est le **R**ole **B**ased **A**ccess **C**ontrol.
- Elle permet en se basant sur les **rôles** et les **privilèges** de spécifier les droits d'accès de chaque utilisateur.
- Vous pouvez implémenter ca en utilisant les guards de Nest.
- L'idée est simple, le guard devra récupérer les rôles nécessaires s'il y en a pour les endpoints de votre contrôleur.
- Ces rôles peuvent êtres associés à un endpoint particulier ou d'une manière globale sur le contrôleur.

Reflection et metadata

- NestJs vous offre la possibilité **d'attacher des métadata** à vos classes ou à vos méthodes en utilisant le décorateur **@SetMetadata().**
- Il prend en **premier paramètre une clé** qui représente le nom de la méta et en **second paramètre la valeur**.

```
// La métadata a pour clé roles et pour valeur  ['admin']
@SetMetadata('roles', ['admin'])
 findAll() {
   return this.cvService.findAll();
 }
```

https://docs.nestjs.com/security/authorization

Reflection et metadata

- Même si cette méthode est fonctionnelle, cependant, **ce n'est pas une bonne pratique, vue le manque de lisibilité du code**.
- **Préférer la création d'un décorateur propre à vous** qui expose cette fonctionnalité.

```
import { SetMetadata } from '@nestjs/common';
export const Roles = (...roles: string[]) => SetMetadata('roles', roles);
```

```
@Roles('admin')
 findAll() {
 }
```

Reflection et metadata

- Maintenant qu'on sait passer des paramètres, il faudra pouvoir les récupérer.
- Pour ce faire, il faudra passer par la class **Reflector** du package **@nestjs/core.**
- Cette classe est **injectable** comme n'importe quel provider.
- Maintenant pour récupérer la métadata, vous avez plusieurs méthodes offerte par le reflector.

Reflection et metadata

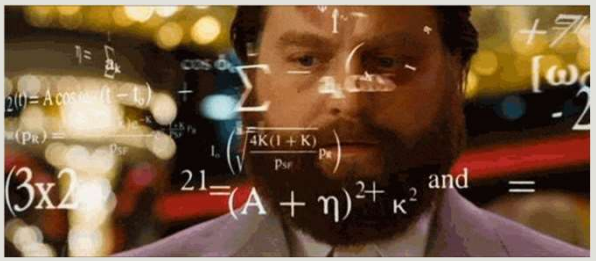
- **get** : prend en paramètre la clé de la métadata à récupérer et le context (le décorateur cible (la fonction ou la classe)).
- **Astuce** : La classe **ExecutionContext** vous offre deux méthodes :
  - **getHandler** qui vous retourne la route actuellement exécutée.
  - **getClass**: Le contrôleur de la route actuellement exécutée.
- **getAll** : retourne un tableau de tableau des métadonnées ayant cette clé
- **getAllAndMerge** : fusionne les tableaux résultant
- **getAllAndOverride** : retourne la première valeur qui n'est pas undefined

Reflection et metadata

```
const all = this.reflector.getAll('roles', [context.getHandler(), context.getClass(),]);
const getAllAndMerge = this.reflector.getAllAndMerge('roles',  [context.getHandler(),
context.getClass(),]);
const getAllAndOverride = this.reflector.getAllAndOverride('roles',  [context.getHandler(),
context.getClass(),]);
console.log('all :', all);
console.log('getAllAndMerge :', getAllAndMerge);
console.log('getAllAndOverride :', getAllAndOverride);
```

```
all : [ [ 'admin' ], [ 'user' ] ]
getAllAndMerge : [ 'admin', 'user' ]
getAllAndOverride : [ 'admin' ]
```

Autorisation RBAC



- Maintenant qu'on sait comment manipuler setMetadata et Reflector, faite en sorte d'avoir un Guard qui ne permet l'accès à la route cible que si le user possède les droits d'accès nécessaires.

aymen.sellaouti@gmail.com