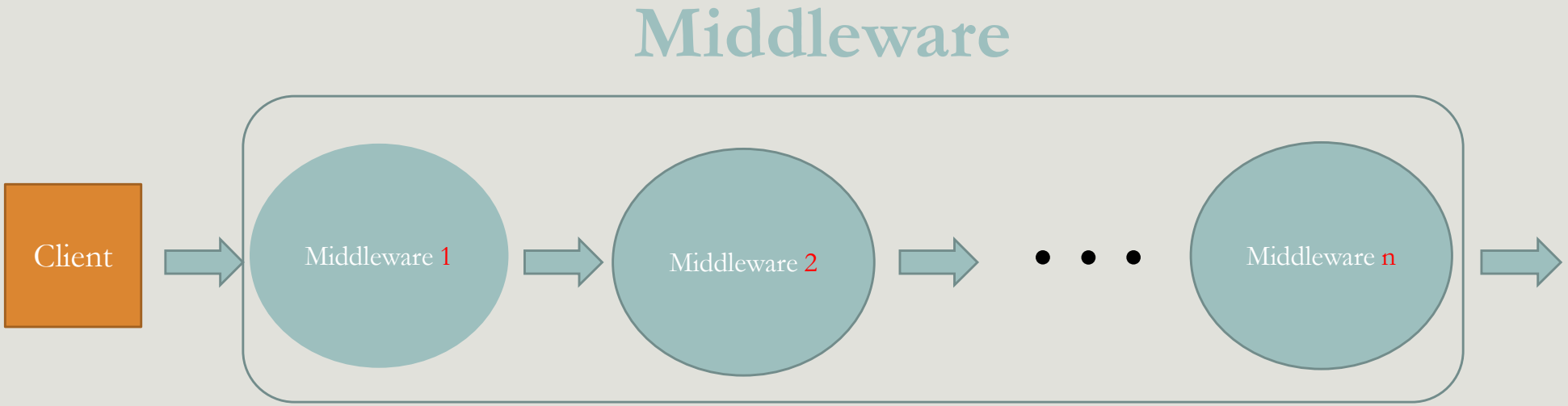
NestJs

AYMEN SELLAOUTI

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



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');
}
```

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();
}
```

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

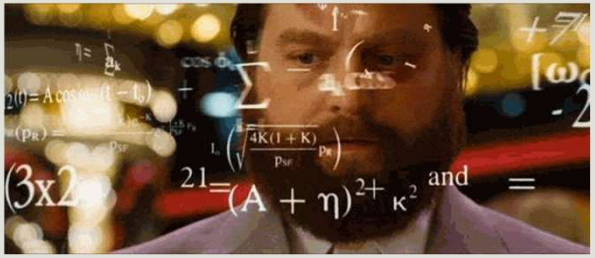
- Vous pouvez aussi déclarer vos ***functions middleware*** au niveau de main.js à travers la méthode use de votre app.
- Le middleware sera **global**.

```
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);
```

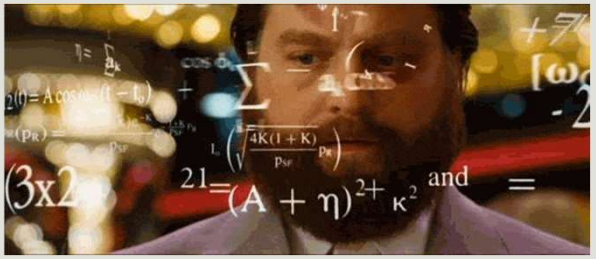
Exercices



- 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";
```

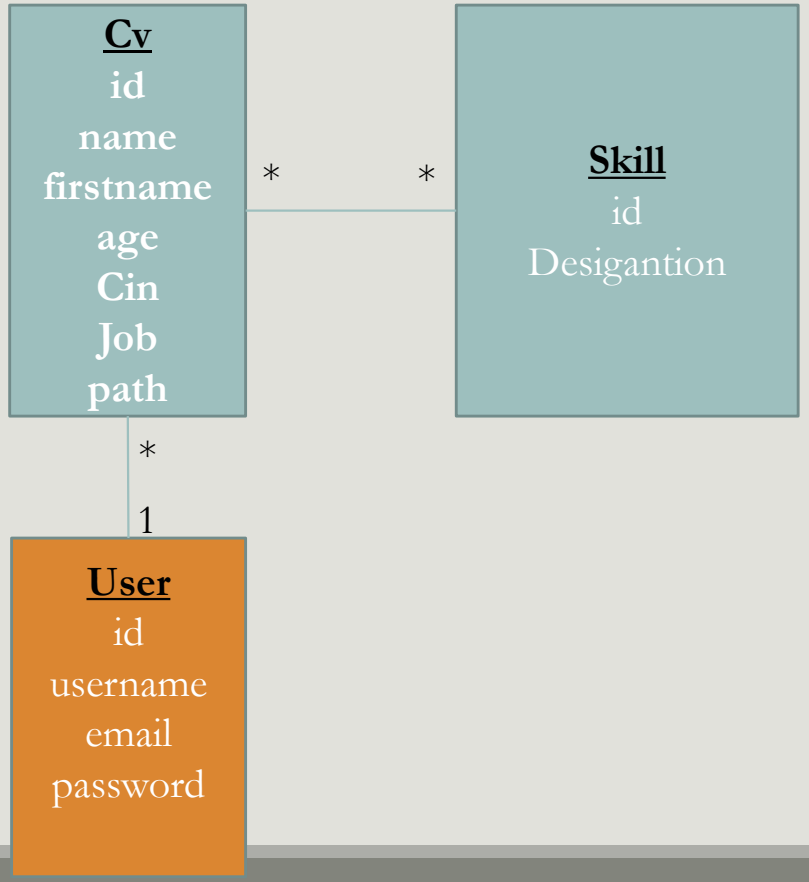
Exercice



- Nous voulons reproduire le schéma relatif à un petit gestionnaire de cvs.
- Créer les modules, contrôleurs, services et entités relatives à ce schéma ainsi que les relations qui y sont associées. Il faut que le CRUD des trois entités soit fonctionnel. **Factoriser votre code.**

Astuce : vous pouvez utiliser la commande : **nest generate resource**

- Cette commande génère le **squelette vide** de 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).



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

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 exportée par les modules que vous importer.
- Vérifiez que vous utilisez des chemins relatifs et non absolus dans vos import pour tous vos éléments du projet.

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
            // Exemple cvService = app.get(CvService);
  await app.close();
}
bootstrap();
```

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

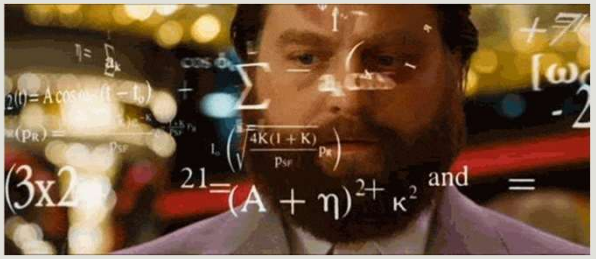
Seed de la base de données
Standalone applications

- 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.
- **Pour que les seeds fonctionnent, il faut que les imports que vous utiliser pour vos différents éléments soient relatifs et non absolues.**

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

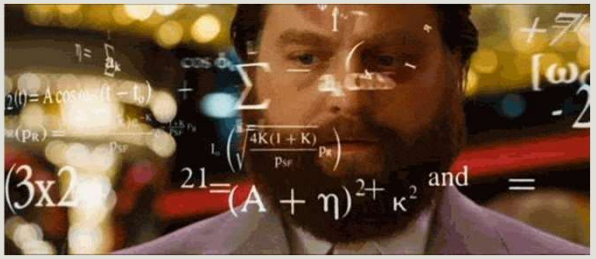
https://docs.nestjs.com/standalone-applications#standalone-applications

Exercice



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

Exercice



- Terminer les différents CRUDs

aymen.sellaouti@gmail.com