- 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 (string,int…) JS, Nest envoi la valeur sans la sérialiser (sans la rendre format json).=>si objet,tableau le rend format json.
- Observable:annuable =>On peut **arrêter l'observation quand on veut.**

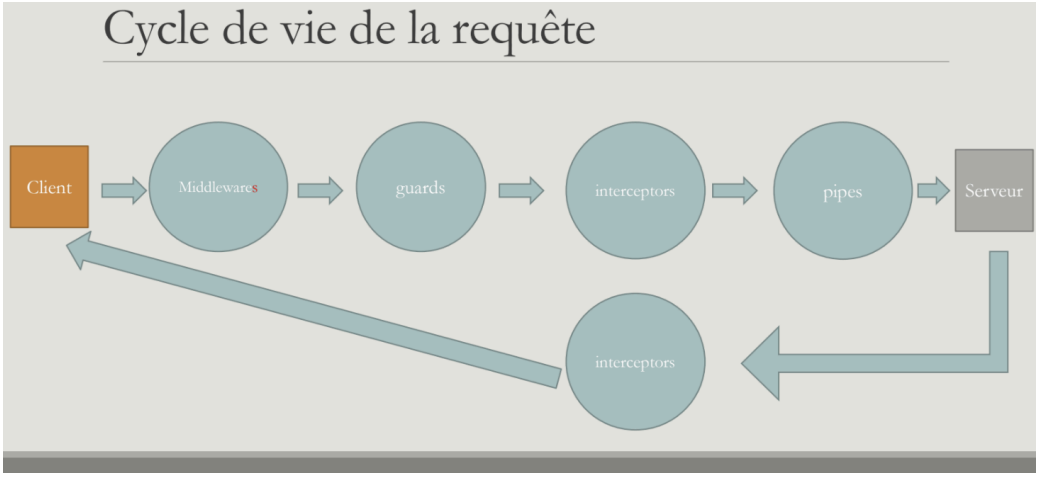
Un Observable est une source de données asynchrones qui peut émettre plusieurs valeurs dans le temps et qui peut être annulée.

- Un **opérateur pipeable** :

- prend **un Observable en entrée**
- retourne **un nouvel Observable**
- **ne modifie pas** l'Observable original (opération pure)

- `@Injectable()` **n'est pas obligatoire** pour dire "c'est un service" MAIS `@Injectable()` est **obligatoire** si la classe :
  - a un **constructor avec des paramètres**
  - a des **dépendances à injecter**

Dans NestJS, lorsque `transform: true` est activé, `class-transformer` est exécuté avant `class-validator`.



- Sans `next()` → la requête **s'arrête ici**

➡ Avec `next()` → elle continue vers le contrôleur

- Un **Interceptor** est une classe qui :

- s'exécute **avant ET après** l'exécution du contrôleur
- peut **intercepter** la requête **et** la réponse
- travaille avec des **Observables**

Différence importante :

- **Middleware** → avant le contrôleur seulement
- **Interceptor** → avant **et après** le contrôleur

next.handle() :Lance l'exécution du contrôleur
`tap` permet d'exécuter du code **sans modifier la réponse**

`forRoot()`	`forFeature()`
Configuration globale	**Configuration locale**
Appelé une seule fois	**Appelé dans plusieurs modules**
Initialise la connexion	**Fournit des repositories**
AppModule	**Modules métier**

`findOne()` → lire
`save()` → enregistrer
`preload()` → lire + préparer la mise à jour(ne sauvegarde pas, il faut faire un save apres)

- const newEntity = await repository.preload({id, name, firstname}); => trouve l'entité avec `id,remplace name et firstname,garde les autres champs,retourne l'objet` **`sans l'enregistrer.`**

`softRemove()` nécessite une entité chargée et déclenche les hooks, tandis que `softDelete()` effectue une suppression logique directement en base sans charger l'entité.

- getMany() ~ find() et getOne() ~ findOne()
- createQueryBuilder("alias")

Méthode	Ce qu'elle retourne
`getOne()`	Une **entité**
`getMany()`	Tableau d'**entités**
`getRawOne()`	1 objet **brut**
`getRawMany()`	Tableau d'objets **bruts**

-**`eager: true`** veut dire : Quand je charge une entité, charge aussi automatiquement ses relations lors d'un find.
-**Cascade** : permet de sauvegarder ou mettre à jour les entités liées automatiquement