

TITRE EXPERTISE INFORMATIQUE ET SYSTÈME D'INFORMATION

BLOC E7.4 – Conception et développement des solutions applicatives métier et spécifiques

**VUYLSTEKER Vincent
RANDRIANAINA Manoarijaona
wasim**

I1 devB

SOMMAIRE

I - Choix des technologies

- 1.1 Choix des méthodes de développement de l'API
- 1.2 Choix de l'infrastructure réseau de l'API
- 1.3 Choix de la technologie pour l'application mobile
- 1.4 Choix des outils collaboratif et agile

II - Architecture réseau

- 2.1 Schéma de l'architecture réseau
- 2.2 Utilisation d'un VPS OVH
- 2.3 Utilisation de docker sur le VPS
- 2.4 CI : Intégration continue
- 2.5 CD : Déploiement continu

III - Architecture du code technique / application

- 3.1 Organisation du code de l'application
- 3.2 Exemple d'un endpoint de l'API
- 3.3 Configuration et utilisation de Prisma ORM
- 3.4 Tests unitaires avec jest

IV - Sécurisation

- 4.1 Variables d'environnements
- 4.2 Github repository secrets
- 4.3 JWT Token

V- Architecture de l'Application

- 5.1 Schéma d'architecture
- 5.2 Écran de connexion
- 5.3 Affichage de la liste des produits
- 5.4 Détail du produit

VI - Livrable (liens)

I - Choix des technologies

1.1) Choix des méthodes de développement de l'API



Lorsque nous avons entrepris notre projet de MSPR, nous avons consacré beaucoup de temps à réfléchir au choix du framework le plus adapté à nos besoins.

Après avoir étudié plusieurs options, nous avons finalement opté pour Nest, un framework basé sur TypeScript et conçu pour le développement d'applications sous Node.js. Plusieurs raisons ont motivé notre choix. Tout d'abord, nous avons été attirés par la puissance et la flexibilité de TypeScript. En tant qu'étudiants soucieux de la qualité du code et de la prévention des erreurs, TypeScript offre des fonctionnalités de typage statique qui permettent de détecter les erreurs avant l'exécution du code.

Cela nous a permis de développer notre application de manière plus sûre et de réduire les erreurs potentielles. Ensuite, Nest nous a séduits par son approche modulaire et structurée du développement. Le framework suit l'architecture MVC (Modèle-Vue-Contrôleur) et propose des modules, des contrôleurs et des fournisseurs de services pour organiser efficacement notre code.

Cette approche nous a permis de maintenir un code propre, facilement testable et évolutif tout au long du développement de notre application. De plus, nous avons ajouté au projet un Eslint et un prettier pour garantir une meilleure propreté du code et faire en sorte que le code soit toujours développé et rédigé de la même manière par l'ensemble des développeurs du groupe.

Les tests unitaires de l'application ont été fait en jest, qui est un framework couplé avec nest lors de l'initialisation du projet.

De plus, nous avons choisi d'utiliser l'ORM Prisma avec Nest pour faciliter la gestion des bases de données. Prisma offre une couche d'abstraction qui simplifie les interactions avec la base de données, tout en garantissant une sécurité accrue grâce à la prévention des attaques par injection SQL. Prisma prend également en charge TypeScript, ce qui a rendu l'intégration avec Nest. En résumé, le choix de Nest avec TypeScript et l'ORM Prisma pour notre projet de MSPR s'est avéré judicieux.

Ces technologies combinées nous ont offert une expérience de développement agréable, en mettant l'accent sur la qualité du code, la modularité et la facilité de gestion de la base de données. Nous sommes convaincus que ce choix nous permettra de produire une application robuste et performante, tout en développant nos compétences en tant que développeurs.

1.2) Choix de l'infrastructure réseau de l'API



Pour l'architecture réseau de notre API, nous avons choisi d'utiliser un VPS fourni par OVH. Ce choix nous permet d'avoir un contrôle total sur notre infrastructure et de l'adapter selon nos besoins.

Notre API est divisée en deux conteneurs Docker, qui sont gérés à l'aide de Docker Compose. Le premier conteneur est configuré avec Nginx, un serveur web réputé pour sa stabilité et ses performances. Nginx joue le rôle de proxy inverse, permettant de gérer les requêtes entrantes et de les rediriger vers notre application backend.

Le deuxième conteneur est basé sur Node.js et exécute notre application API développée avec Nest framework. Node.js est choisi pour sa rapidité et sa capacité à gérer un grand nombre de requêtes simultanées.

Par ailleurs, nous avons mis en place un troisième conteneur externe avec Docker Compose pour héberger notre base de données. Nous avons opté pour MariaDB, une solution de gestion de base de données relationnelle fiable et compatible avec MySQL.

En séparant le conteneur de la base de données, nous nous assurons que celle-ci reste active même en cas de réinitialisation des autres conteneurs.

Pour la mise à jour de notre API, nous avons mis en place une intégration continue (CI) qui se connecte en SSH via GitHub. Lorsque des modifications sont apportées au code source de notre API et poussées vers la branche principale de notre dépôt GitHub, la CI se déclenche automatiquement.

Elle récupère les dernières modifications du code, lance les tests automatiques et, si tout est réussi, met à jour notre API en déployant les conteneurs Docker sur notre VPS.

Cette approche de déploiement automatisé nous permet de garantir que notre API est constamment mise à jour avec les dernières modifications du code, tout en assurant une disponibilité continue grâce à la gestion séparée de la base de données. Ainsi, nous évitons les problèmes potentiels lors du démarrage de tous les conteneurs après une réinitialisation.

En résumé, notre architecture réseau de l'API repose sur un VPS OVH, des conteneurs Docker pour l'API avec Nginx et Node.js, un conteneur externe pour la base de données MariaDB et une intégration continue pour la mise à jour automatique de l'API via GitHub. Cette configuration nous permet d'avoir une infrastructure fiable, évolutive et facilement gérable pour notre API.

1.3) Choix de la technologie pour l'application mobile



Nous avons opté pour le framework Kivy en Python afin de développer une application mobile multiplateforme.

Notre choix s'est basé sur plusieurs raisons clés. Tout d'abord, nous avons privilégié l'utilisation de Python en raison de sa popularité, de sa lisibilité et de sa simplicité. Ce langage de programmation offre également une vaste gamme de bibliothèques et de frameworks qui facilitent le développement d'applications.

En ce qui concerne Kivy, il s'agit d'un framework open source spécialement conçu pour créer des interfaces utilisateur attrayantes dans le contexte de développement d'applications multiplateformes. Grâce à sa flexibilité, nous avons pu concevoir des interfaces graphiques riches et interactives pour notre application. L'un de nos objectifs principaux était de développer une application mobile capable de fonctionner sur plusieurs plateformes, notamment Android et iOS.

Kivy nous a permis de créer une application unique qui peut être déployée sur différentes plateformes, ce qui nous a fait gagner un temps précieux en évitant la nécessité de réécrire le code pour chaque plateforme. Un autre aspect important de notre application est l'intégration de fonctionnalités de réalité augmentée. Kivy propose des outils et des fonctionnalités qui facilitent l'incorporation de la réalité augmentée, nous permettant ainsi de créer des expériences interactives et immersives pour nos utilisateurs.

Enfin, notre application doit pouvoir lire et décoder des codes QR. Kivy offre des bibliothèques et des modules qui nous permettent de mettre en œuvre cette fonctionnalité essentielle.

1.4) Choix des outils collaboratif et agile



Pour faciliter notre collaboration, nous avons pris soin de sélectionner des outils collaboratifs appropriés à nos besoins.

Tout d'abord, nous avons opté pour Git et GitHub, une combinaison puissante pour le contrôle de version de notre code source.

Cette solution nous permet de travailler simultanément sur le projet, de fusionner facilement nos modifications et de revenir à des versions précédentes si nécessaire.

Exemple de notre repository Kawa sur github :

A screenshot of a GitHub repository page. At the top, it shows the main branch, 2 branches, 0 tags, and buttons for Go to file, Add file, and Code. Below this is a list of commits:

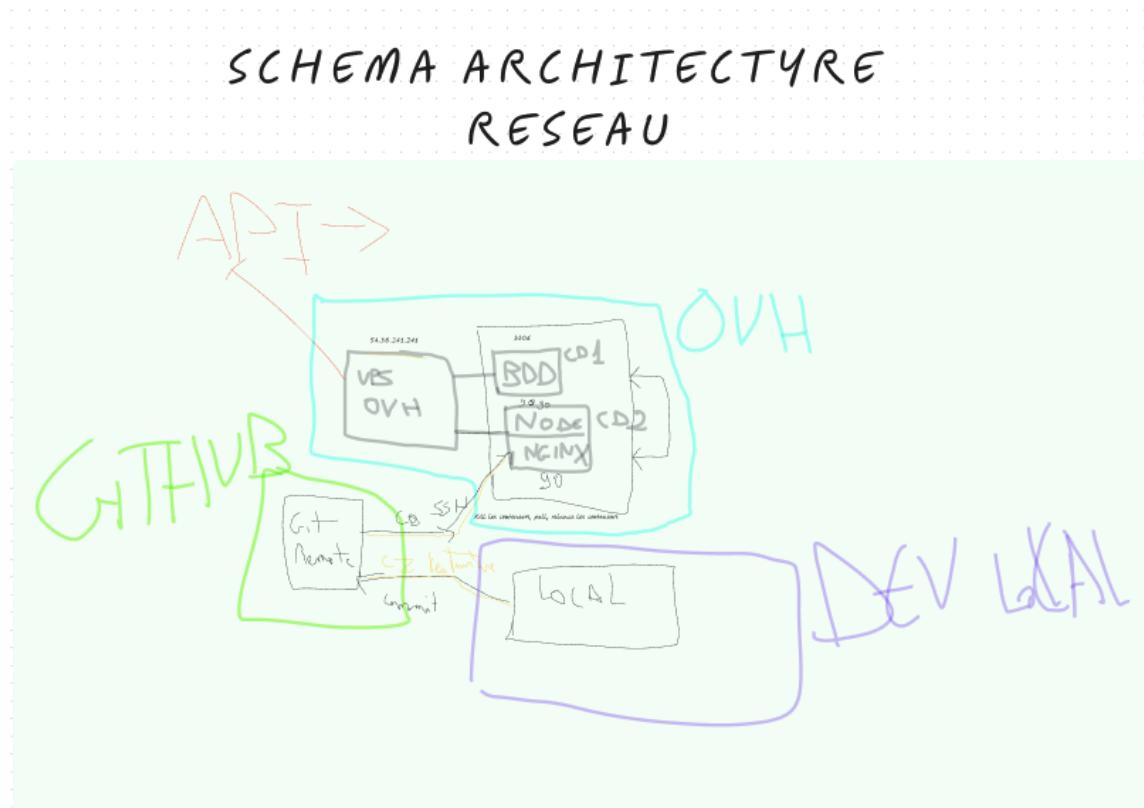
Commit	Message	Author	Date	Commits
	feat : add jwt to bearer auth system	vincent55312	yesterday	55 commits
	.github/workflows		5 days ago	
	prisma		5 days ago	
	src		yesterday	
	test		2 months ago	
	.env.template		5 days ago	

Nous avons utilisé deux branches, une branche **dev** et une branche **main** pour la production. Cette pratique nous permet de pouvoir faire des pull request et de merger le code en cours de développement dans main quand le code nous semble adéquat. De plus, nous nous sommes efforcé d'employer des bon noms de commits, selon conventionalcommits.org où tous nos commits commencent par feat : / fix : / chore :

Avoir un politique de nommage de commit, permet de mieux s'y retrouver dans le versionning et de pouvoir rollback le projet à une version qui nous intéresse. Par exemple pour trouver un bug.

En ce qui concerne la conception de l'interface utilisateur, nous avons choisi Figma. Cet outil de conception basé sur le cloud nous permet de créer et de partager des maquettes interactives en temps réel. Grâce à ses fonctionnalités de collaboration, nous pouvons facilement collaborer sur le design, recueillir des commentaires et apporter des modifications en temps réel, ce qui accélère notre processus de conception.

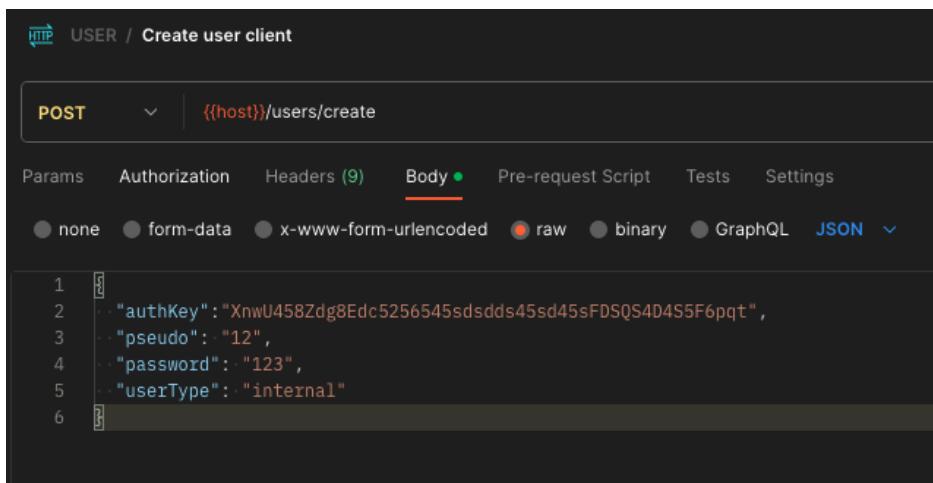
Exemple d'une ébauche que nous avons fait sous figma pour mettre nos idées en commun:



En ce qui concerne les appels API et l'uniformisation des requêtes, pour le développement, nous avons choisi Postman. Cet outil nous permet de tester et de valider nos API de manière efficace.

Nous pouvons avec, facilement créer des requêtes HTTP et tester les réponses de l'API.

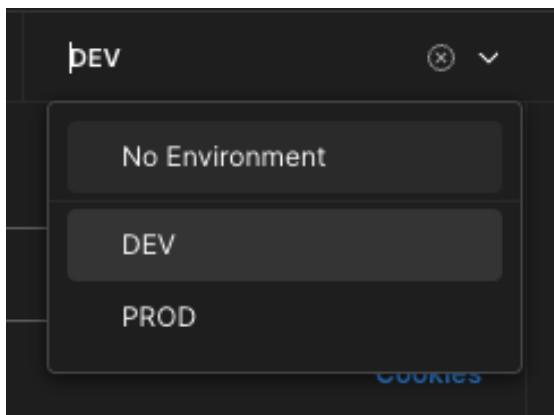
Exemple utilisation Postman :



Sous le logiciel Postman, avec la création d'une collection "kawa" dédiée, nous avons créé plusieurs environnements pour travailler aisément.

Par exemple, un environnement PROD destiné à communiquer avec notre serveur OVH et un environnement DEV pour le développement en local.

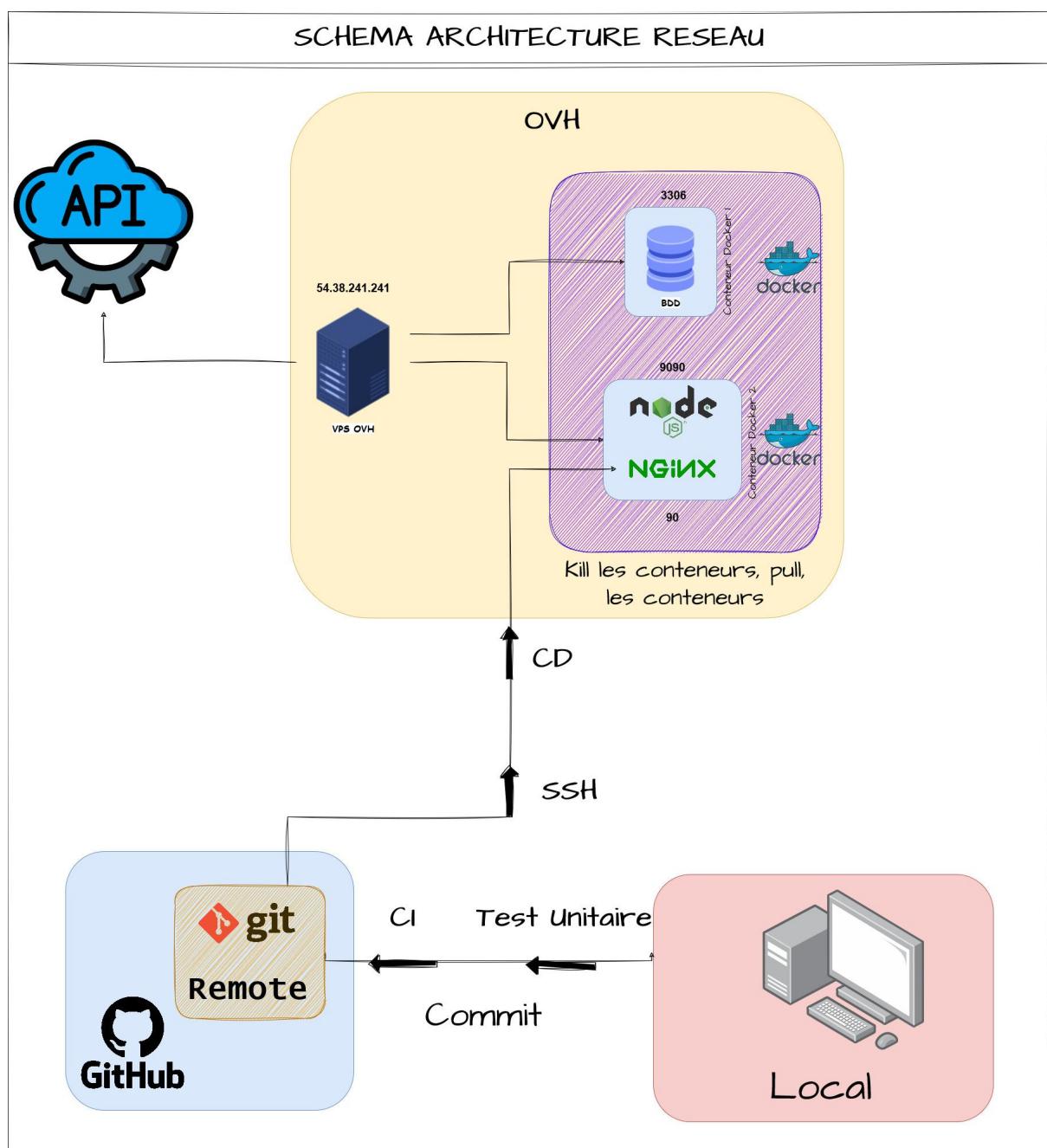
Les types d'environnement différents vont permettre de modifier la valeur {{host}} lors des call API et de facilement changer d'environnement de travail pour tester et utiliser l'API.



Grâce à ces choix d'outils collaboratifs, nous avons pu optimiser notre travail d'équipe, améliorer notre productivité et assurer une communication fluide et efficace tout au long de notre projet de MSPR.

II) Architecture réseau

2.1) Schéma de l'architecture réseau



2.2) Utilisation d'un VPS OVH

Nous avons choisi d'utiliser un VPS OVH au lieu d'une solution cloud comme AWS ou Azure pour notre projet de groupe MSPR.

Nous avons opté pour le VPS OVH car il était plus abordable et répondait à nos besoins en tant qu'étudiants. Nous avons également apprécié le contrôle total que nous avions sur notre environnement de déploiement, ce qui nous a permis de personnaliser notre serveur selon nos besoins spécifiques. L'utilisation de Docker a simplifié le déploiement sur le VPS OVH, et la réputation d'OVH en tant que fournisseur fiable nous a également rassurés. Le VPS OVH était rentable, offrait une flexibilité et une facilité de déploiement, ce qui en faisait un choix idéal pour notre projet de cours avec une petite infrastructure.

VALUE

5,80 €
5,33 €
HT/mois
soit 6,40 € TTC/mois

Engagement 12 mois (-8%) ▾

Commander

1 vCore
2 Go
40 Go SSD NVMe
250 Mbit/s illimité*

[Comparer →](#)

Si nous avions voulu avoir une infrastructure plus importante, évolutive, pouvant tenir la charge pour de nombreux utilisateurs, nous serions partis sur une infrastructure server-less de type amazon ou azure.

Voici le VPS sur notre dashboard OVH. Nous avons choisi une localisation du VPS en France, proche de Gravelines, pour avoir une latence faible pour la zone européenne.

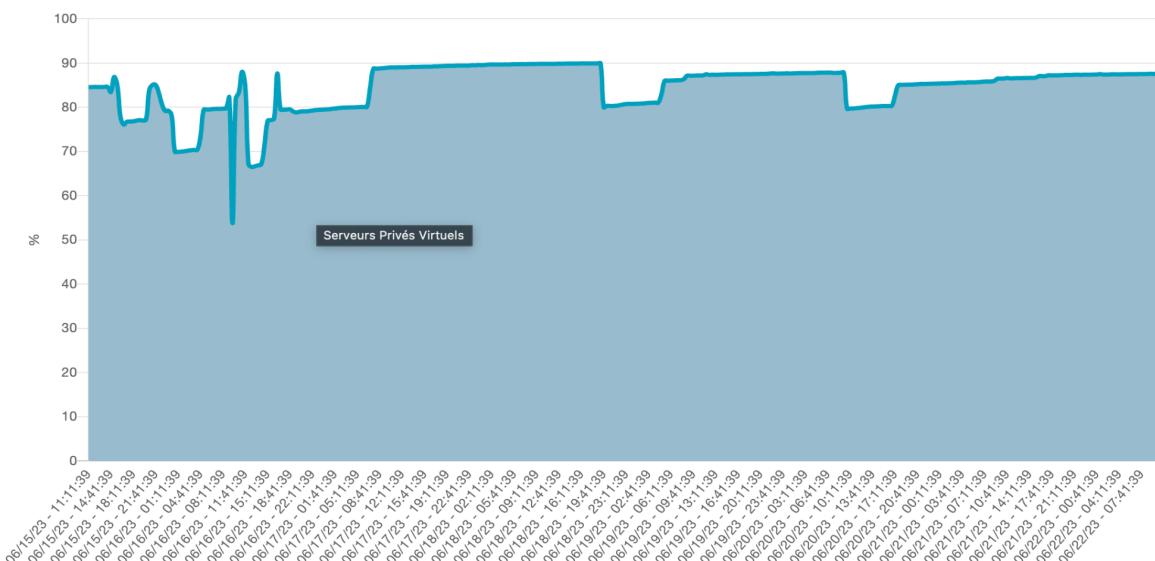
Le système d'exploitation du VPS est un debian avec docker déjà installé pour faciliter le déploiement et n'avoir peu de configuration à effectué sur le serveur.

Votre VPS		Votre configuration		IP	
Statut	Actif	Modèle	VPS vps2020-starter-1-2-20	IPv4	54.38.241.241
Nom	hedgebot-app	vCores	1	IPv6	2001:41d0:305:2100::7e78
Boot	LOCAL	Ajouter des vCores en passant à la gamme supérieure		Gateway	2001:41d0:305:2100::1
OS / Distribution	Debian 10 - Docker	Mémoire	2 Go	DNS secondaire	Aucun domaine configuré
Zone	Region OpenStack: os-gra6	Ajouter de la mémoire en évoluant vers la gamme supérieure		Stockage	20 Go
Localisation	FR Gravelines (GRA) - France	Stockage	20 Go	Augmenter le stockage en évoluant vers la gamme supérieure	

L'avantage aussi avec OVH, est qu'il nous permet d'avoir un monitoring dédié. Un redémarrage automatique du VPS en cas de problème, un sécurité anti-ddos, ect..

Un exemple du monitoring de la RAM :

Utilisation de la RAM



On remarque que nous sommes à 80% en moyenne d'utilisation RAM, ce qui fait que notre VPS est adéquat avec les ressources demandées par notre application.

2.3) Utilisation de docker sur le VPS

Sur notre serveur fonctionnant en production, après une connexion en SSH, nous pouvons voir tous les conteneurs utilisés et en cours.

```

~ (0.427s)
ssh debian@54.38.241.241

debian@vps-0bb9531f:~ (0.114s)

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.

debian@vps-0bb9531f:~ (0.089s)
docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
9ebb22b3f27c kawa_nest "docker-entrypoint.s..." 2 days ago Up 2 days 0.0.0.0:9090->9090/tcp, :::9090->9090/tcp kawa_nest_1
c7483c4b6940 kawa_nginx "/docker-entrypoint.s..." 2 days ago Up 2 days 0.0.0.0:90->90/tcp, 80/tcp, 0.0.0.0:444->444/tcp, :::444->444/tcp kawa_nginx_1
b0f305e94d3c mariadb "docker-entrypoint.s..." 6 days ago Up 6 days 0.0.0.0:3306->3306/tcp, :::3306->3306/tcp kawa_db_mariadb_1

```

On remarque que nous utilisons 3 conteneurs différents:

- Nginx
- Node js
- Maria db

Nous avons organisé nos conteneurs en deux projets distincts à l'aide de docker-compose. Le premier projet comprend l'API avec les conteneurs Nginx et Node.js, tandis que le second projet est dédié à la base de données. Nous avons pris la décision de ne pas regrouper les trois conteneurs dans le même docker-compose.

Cette décision a été motivée par notre volonté d'avoir une base de données permanente, évitant ainsi les redémarrages fréquents. En séparant la base de données dans un projet distinct, nous assurons également une connexion stable, indépendamment du montage des conteneurs. Ainsi, même si l'ordre de montage des conteneurs est perturbé, notre projet reste fonctionnel, évitant les erreurs de plantage.

Cette approche nous permet d'optimiser la stabilité et la fiabilité de notre application, en garantissant une base de données constamment accessible et en évitant les problèmes potentiels liés au démarrage des conteneurs.

Exemple d'un lancement des conteneurs de l'API avec un docker-compose up --build :

```
debian@vps-0bb9531f:~/kawa git:(main)
docker-compose up --build
Creating network "kawa_default" with the default driver
Building nginx
Step 1/2 : FROM nginx:latest
--> eb4a57159180
Step 2/2 : COPY nginx.conf /etc/nginx/nginx.conf
--> Using cache
--> ea1c42c89c80
Successfully built ea1c42c89c80
Successfully tagged kawa_nginx:latest
Building nest
Step 1/7 : FROM node:14-alpine
--> 0dac3dc27b1a
Step 2/7 : WORKDIR .
--> Using cache
--> 75f154cff6d2
Step 3/7 : COPY .
--> Using cache
--> 7239e69765fc
Step 4/7 : RUN npm install
--> Using cache
--> fbbdb4ec0bbcc
Step 5/7 : RUN npm run build
--> Using cache
--> dab00f79aa6f
Step 6/7 : EXPOSE 9090
--> Using cache
--> 3095603f197c
Step 7/7 : CMD ["npm", "run", "start:prod"]
--> Using cache
--> 8e2af6b1a634
Successfully built 8e2af6b1a634
Successfully tagged kawa_nest:latest
Creating kawa_nest_1 ... done
Creating kawa_nginx_1 ... done
Attaching to kawa_nginx_1, kawa_nest_1
nginx_1 | /docker-entrypoint.sh: /docker-entrypoint.d/ is not empty, will attempt to perform configuration
nginx_1 | /docker-entrypoint.sh: Looking for shell scripts in /docker-entrypoint.d/
nginx_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/10-listen-on-ipv6-by-default.sh
nginx_1 | 10-listen-on-ipv6-by-default.sh: info: Getting the checksum of /etc/nginx/conf.d/default.conf
nginx_1 | 10-listen-on-ipv6-by-default.sh: info: Enabled listen on IPv6 in /etc/nginx/conf.d/default.conf
nginx_1 | /docker-entrypoint.sh: Sourcing /docker-entrypoint.d/15-local-resolvers.envsh
nginx_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/20-envsubst-on-templates.sh
nginx_1 | /docker-entrypoint.sh: Launching /docker-entrypoint.d/30-tune-worker-processes.sh
nginx_1 | /docker-entrypoint.sh: Configuration complete; ready for start up
nest_1
nest_1 | > kawa@0.0.5 start:prod /
nest_1 | > node dist/src/main
nest_1
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [NestFactory] Starting Nest application..
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [InstanceLoader] JwtModule dependencies initialized +29ms
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [InstanceLoader] AppModule dependencies initialized +1ms
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [RoutesResolver] UserController {/users}: +24ms
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [RouterExplorer] Mapped {/users/create, POST} route +8ms
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [RouterExplorer] Mapped {/users/login, POST} route +2ms
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [RoutesResolver] CheckController {/}: +1ms
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [RouterExplorer] Mapped {/, GET} route +1ms
nest_1 | [Nest] 24 - 06/22/2023, 8:33:59 AM      LOG [NestApplication] Nest application successfully started +153ms
```

La commande va build les images docker et exécuter la commande du dockerfile. Ensuite nous voyons que l'API kawa est en marche avec les différentes routes accessibles.

Voici le code du docker-compose permettant de lancer le serveur nginx associé au serveur node js:

```

1 version: '3'
2
3 > services:
4 >   nginx:
5     build:
6       context: .
7       dockerfile: Dockerfile-nginx
8     ports:
9       - "90:90"
10      - "444:444"
11     volumes:
12       - ./nginx.conf:/etc/nginx/nginx.conf:ro
13       - /etc/nginx/conf.d
14       - /var/log/nginx
15     env_file:
16       - .env
17 >   nest:
18     build:
19       context: .
20       dockerfile: Dockerfile-nest
21     ports:
22       - "9090:9090"
23     env_file:
24       - .env
25     environment:
26       - DATABASE_URL=${DATABASE_URL}
27
28   volumes:
29     mariadb_data:

```

Il y a deux services nommés nginx et nest. Chaque service possède un dockerfile où est définie l'image de build du conteneur. Les ports sont également spécifiés ainsi que le fichier d'environnement.

Voici l'image de notre conteneur nest :

```

1 > FROM node:14-alpine
2 WORKDIR .
3 COPY . .
4
5 RUN npm install
6 RUN npm run build
7
8 EXPOSE 9090
9
10 CMD ["npm", "run", "start:prod"]

```

2.4) CD : Déploiement continue

Nous avons mis en place une livraison continue (CD) sur GitHub pour notre projet, en utilisant une connexion SSH vers le serveur.

Cette configuration CD nous permet d'automatiser le déploiement de notre application à chaque fois qu'il y a une mise à jour du code source sur la branche principale. Nous utilisons une connexion SSH sécurisée pour transférer les fichiers et exécuter les commandes nécessaires sur le serveur. (Utilisations de secrets keys stocké de manière sécurisé en tant que variable d'environnement dans le repository github)

Cette approche de livraison continue nous permet de garantir des déploiements rapides et cohérents, en minimisant les erreurs manuelles et en assurant une livraison efficace de notre application sur le VPS ovh.

Voici le code de notre CD:

```

1   name: CD deploy
2
3   on:
4     push:
5       branches: [ main ]
6
7   jobs:
8     build:
9       runs-on: ubuntu-latest
10
11    steps:
12      - name: Deploy using ssh
13        uses: appleboy/ssh-action@master
14        with:
15          host: ${{ secrets.HOST }}
16          username: ${{ secrets.USERNAME }}
17          key: ${{ secrets.SSHKEY }}
18          port: 22
19          script: |
20            cd kawa
21            docker-compose down -v
22            git pull
23            docker-compose up --build -d --force-recreate

```

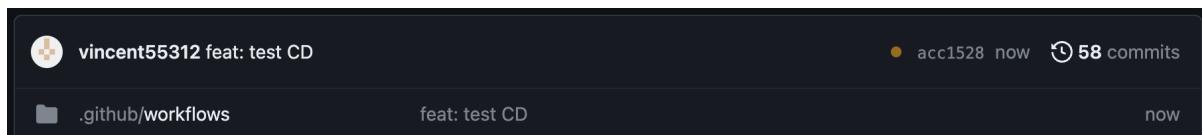
On remarque qu'il utilise une connexion SSH vers notre serveur, se connecte avec les secrets keys et utilise le script suivant :

```
cd kawa
docker-compose down -v
git pull
docker-compose up --build -d --force-recreate
```

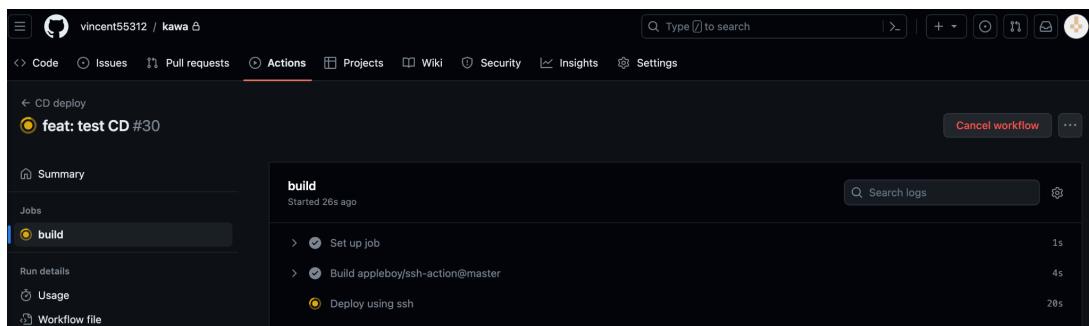
Ce script permet de se placer dans le dossier du projet, kill les conteneurs ainsi que les volumes créés, ensuite git pull le projet, soit récupérer les dernières modifications puis, recréer tous les conteneurs en background.

Voici un exemple du processus de déploiement automatique mis en place :

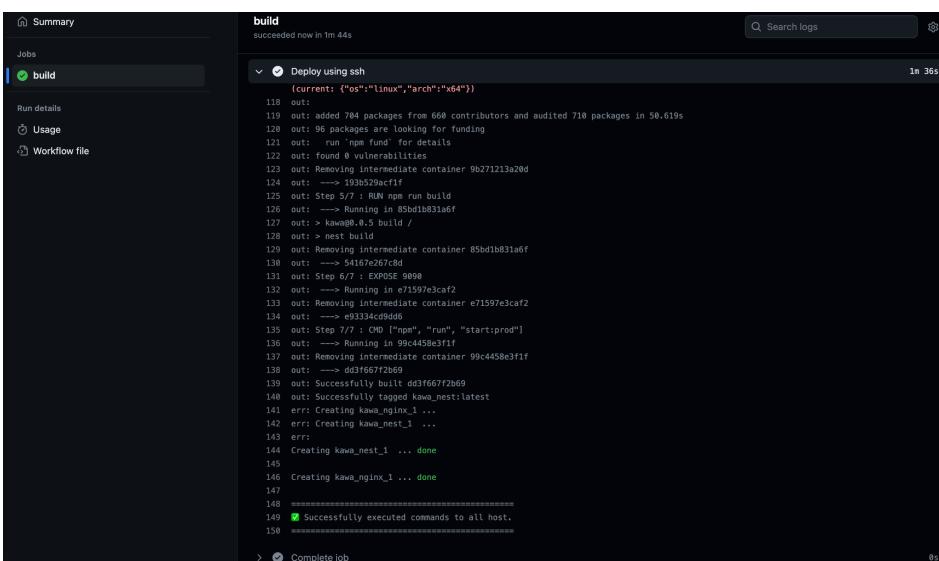
Commit :



Déclenchement de la CD sur github :



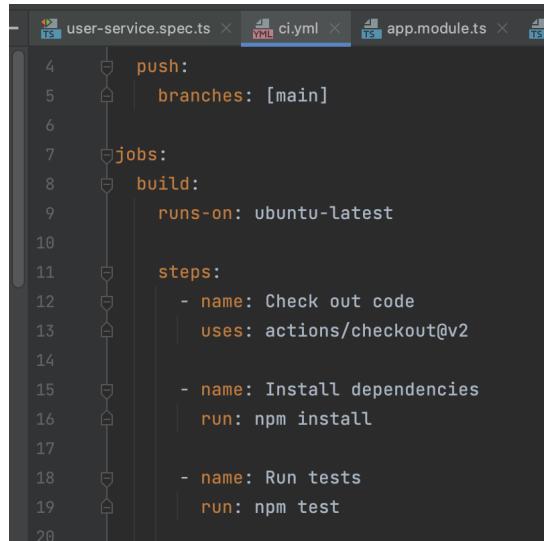
Exécution du script bash et re-création des conteneurs docker :



Le déploiement automatique a été un succès, l'api a été mise à jour correctement.

2.5) CI : Intégration continue

Nous avons également mis en place une CI, une intégration continue des tests unitaires. Voici par exemple les steps permettant de faire run les tests unitaires à l'intérieur de la CI sur github :

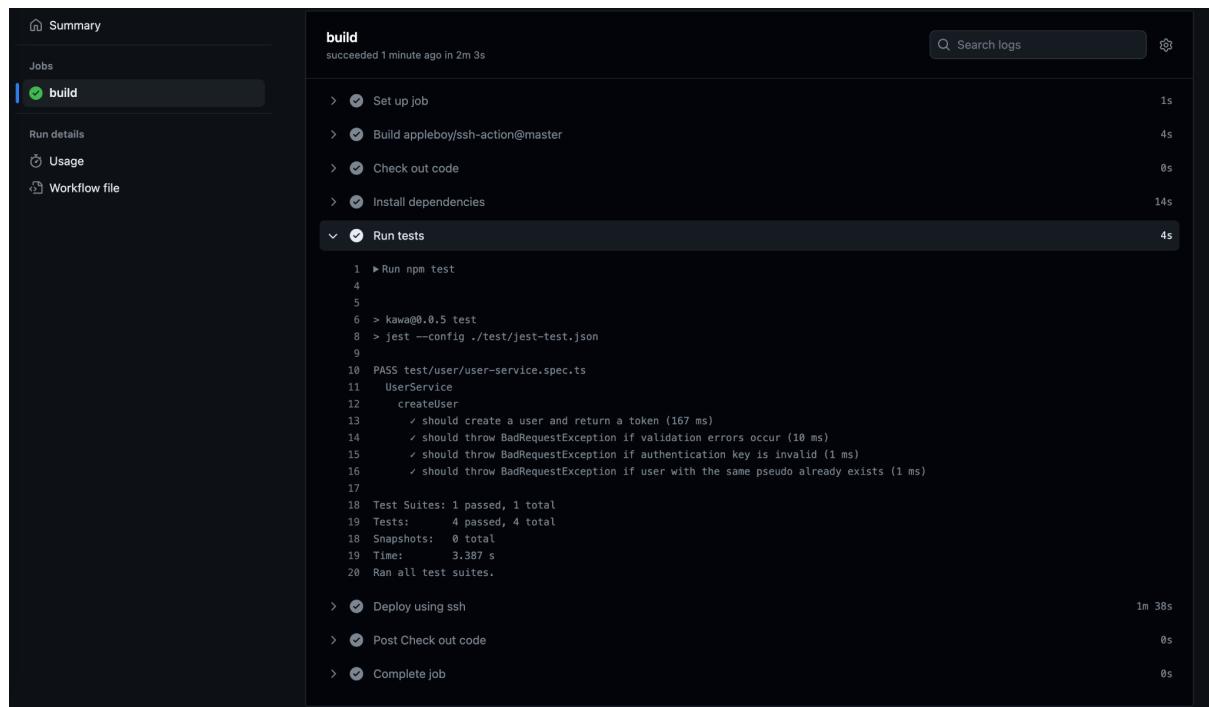


```

4 push:
5   branches: [main]
6
7   jobs:
8     build:
9       runs-on: ubuntu-latest
10
11     steps:
12       - name: Check out code
13         uses: actions/checkout@v2
14
15       - name: Install dependencies
16         run: npm install
17
18       - name: Run tests
19         run: npm test
20

```

Voici l'exemple d'un run des tests dans la CI:



Step	Description	Time
Set up job		1s
Build appleboy/ssh-action@master		4s
Check out code		0s
Install dependencies		14s
Run tests	Run npm test kawa@0.0.5 test jest --config ./test/jest-test.json Test Suites: 1 passed, 1 total Tests: 4 passed, 4 total Snapshots: 0 total Time: 3.387 s Ran all test suites.	4s
Deploy using ssh		1m 38s
Post Check out code		0s
Complete job		0s

III) Architecture du code technique / application API

3.1) Organisation du code de l'application du dossier principal

Voici l'organisation du dossier SRC du code de l'API. On retrouve une organisation avec /src qui va avoir tout le code fonctionnel relatif à l'application.

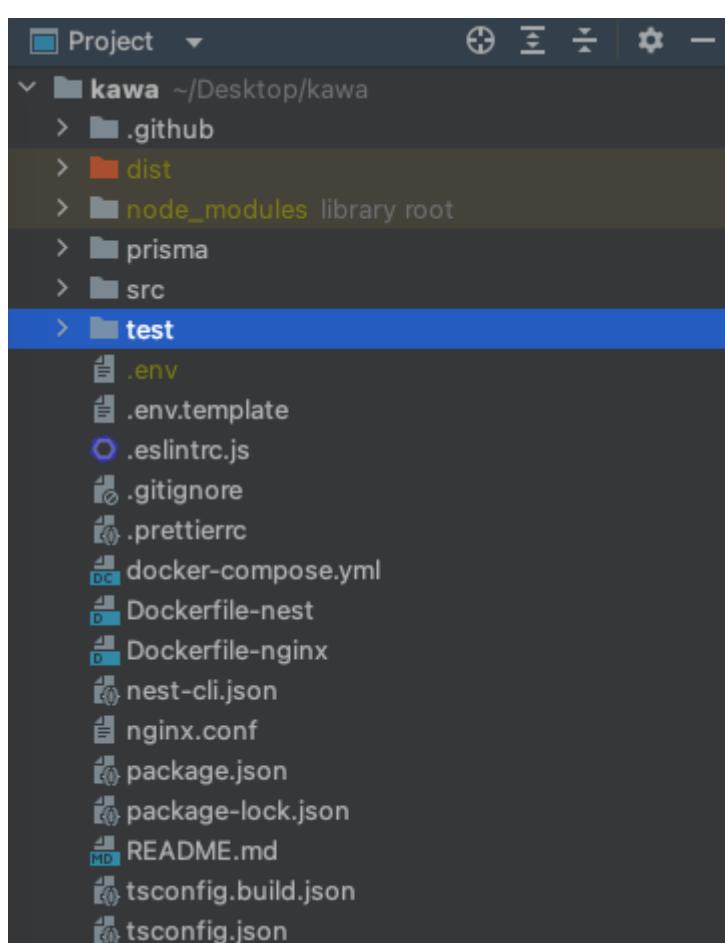
Le répertoire **/prisma** pour la configuration de l'ORM.

Le répertoire de **/test** relatif aux tests end to end de l'API.

Le répertoire **/.github** relatif pour la CI CD.

Le répertoire **/dist** pour le build compilé de l'application.

Le répertoire **/nodes_modules** pour dépendances de l'application.



Organisation du dossier src :

/controllers pour les contrôleurs du projets

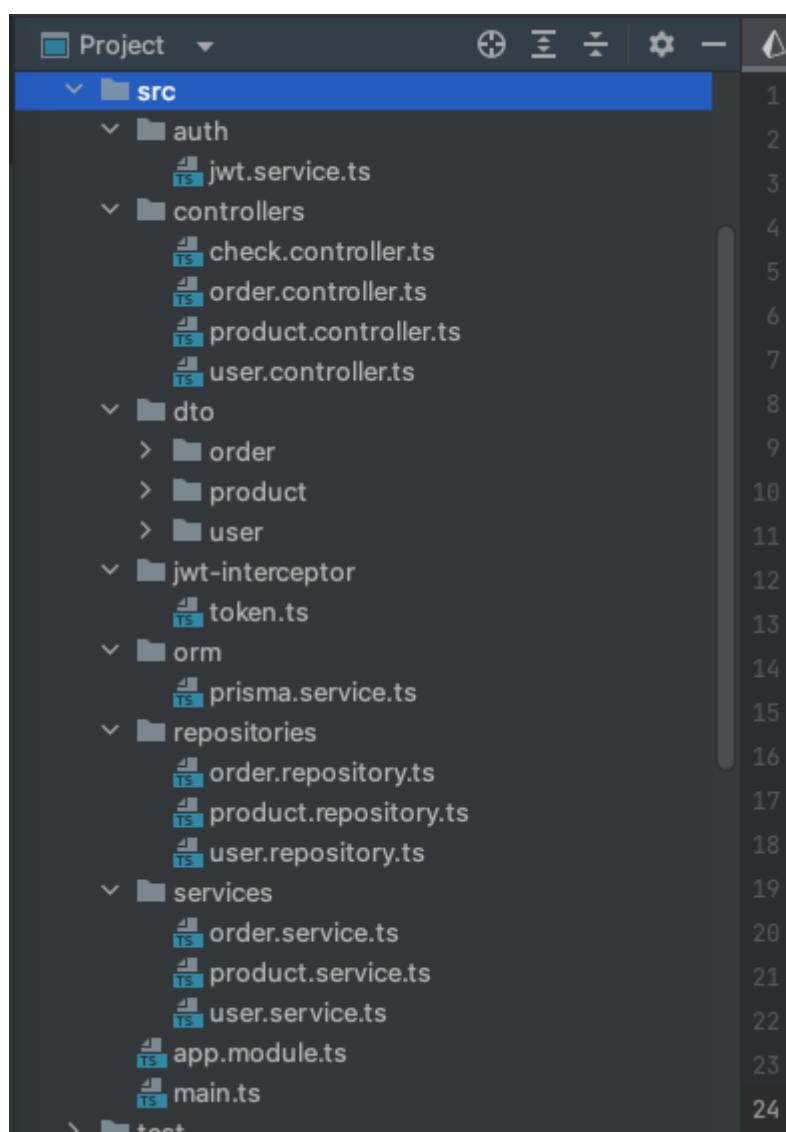
/dto pour tous les dto (data -transfert- object), une surcouche des entités prisma

/auth relatif à l'authentification et la sécurisation de l'api

/orm relatif au prisma service pour faire des appels à la base de donnée

/repositories pour les interfaces et récupérer les données à l'orm

/services : relatif au code métier



3.2) Exemple d'un endpoint de l'API

Voici un exemple montrant le code utilisé pour procéder à un login de l'utilisateur.

On retrouve le users controller, attendant un POST à l'endpoint : /users/login

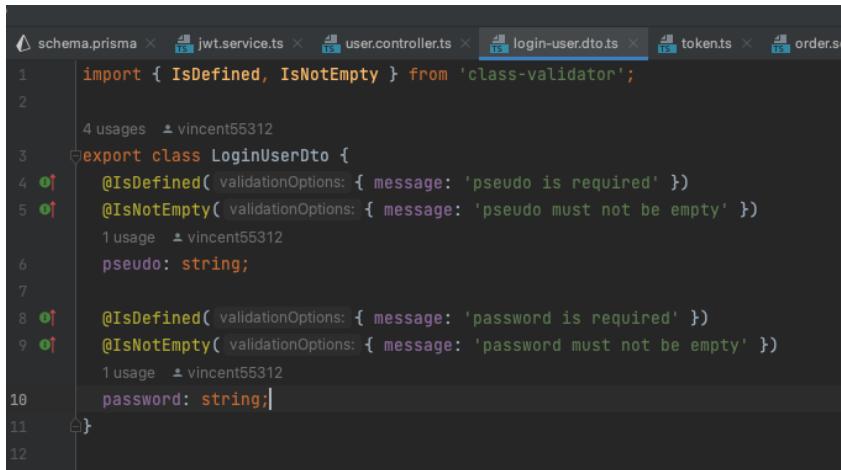
```
@Controller(prefix: 'users')
export class UserController {
    no usages vincent55312
    constructor(private readonly userService: UserService) {}

    no usages vincent55312
    @Post(path: '/create')
    async createUser(
        @Body() createUserDto: CreateUserDto,
    ): Promise<{ token: string }> {
        return await this.userService.createUser(createUserDto);
    }

    no usages vincent55312
    @Post(path: '/login')
    async loginUser(
        @Body() loginUserDto: LoginUserDto,
        @Token() bearerToken: string,
    ): Promise<{ token: string }> {
        return await this.userService.loginUser(loginUserDto, bearerToken);
    }
}
```

Pour **login** l'utilisateur, un DTO a été créé, permettant de récupérer les données du body de la **request POST** envoyé par le client. Ce DTO est créé avec l'aide de class-validator, un package permettant de directement envoyer des messages d'erreurs en fonction des données attendues par le client.

Un décorateur **@token** a également été créé afin de récupérer facilement le token JWT des authorization de la request.



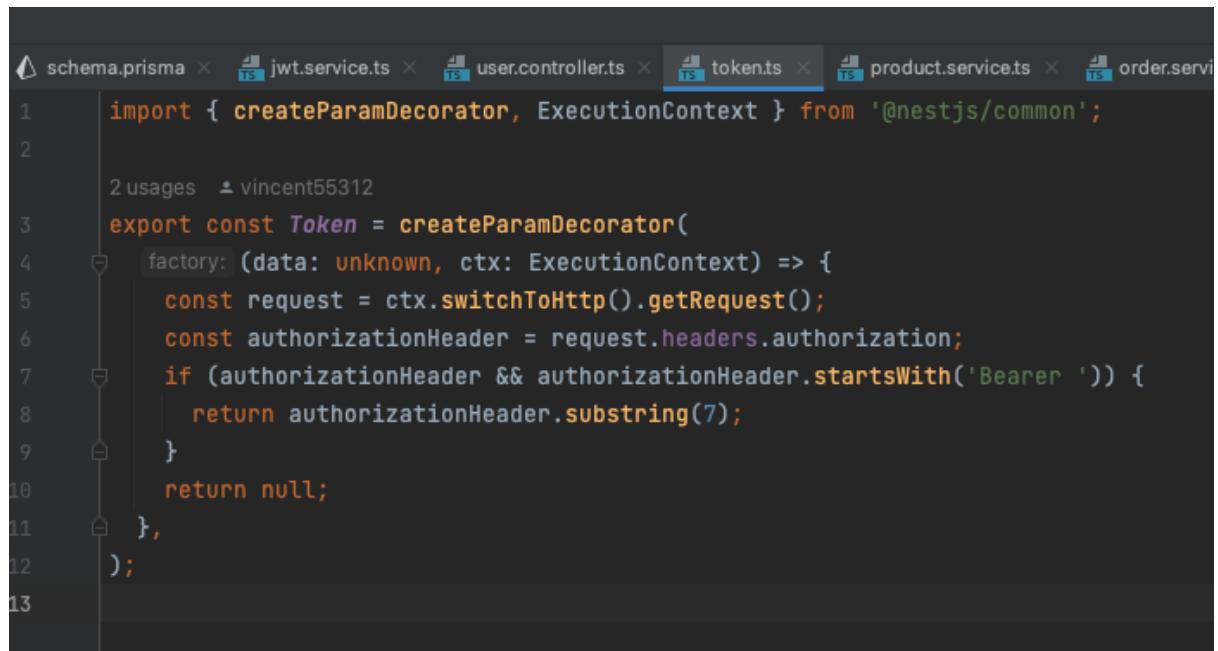
```

1 import { IsDefined, IsNotEmpty } from 'class-validator';
2
3 export class LoginUserDto {
4   @IsDefined({ message: 'pseudo is required' })
5   @IsNotEmpty({ message: 'pseudo must not be empty' })
6   pseudo: string;
7
8   @IsDefined({ message: 'password is required' })
9   @IsNotEmpty({ message: 'password must not be empty' })
10  password: string;
11 }
12

```

Ce code définit un décorateur appelé Token dans le framework NestJS. Un décorateur est une fonction spéciale qui permet d'ajouter des fonctionnalités supplémentaires à une classe ou à une méthode. Ce décorateur est utilisé pour extraire un jeton d'autorisation à partir d'une requête HTTP.

Il vérifie si l'en-tête d'autorisation de la requête commence par Bearer ". Si c'est le cas, il retourne le jeton d'autorisation. Sinon, il renvoie null.

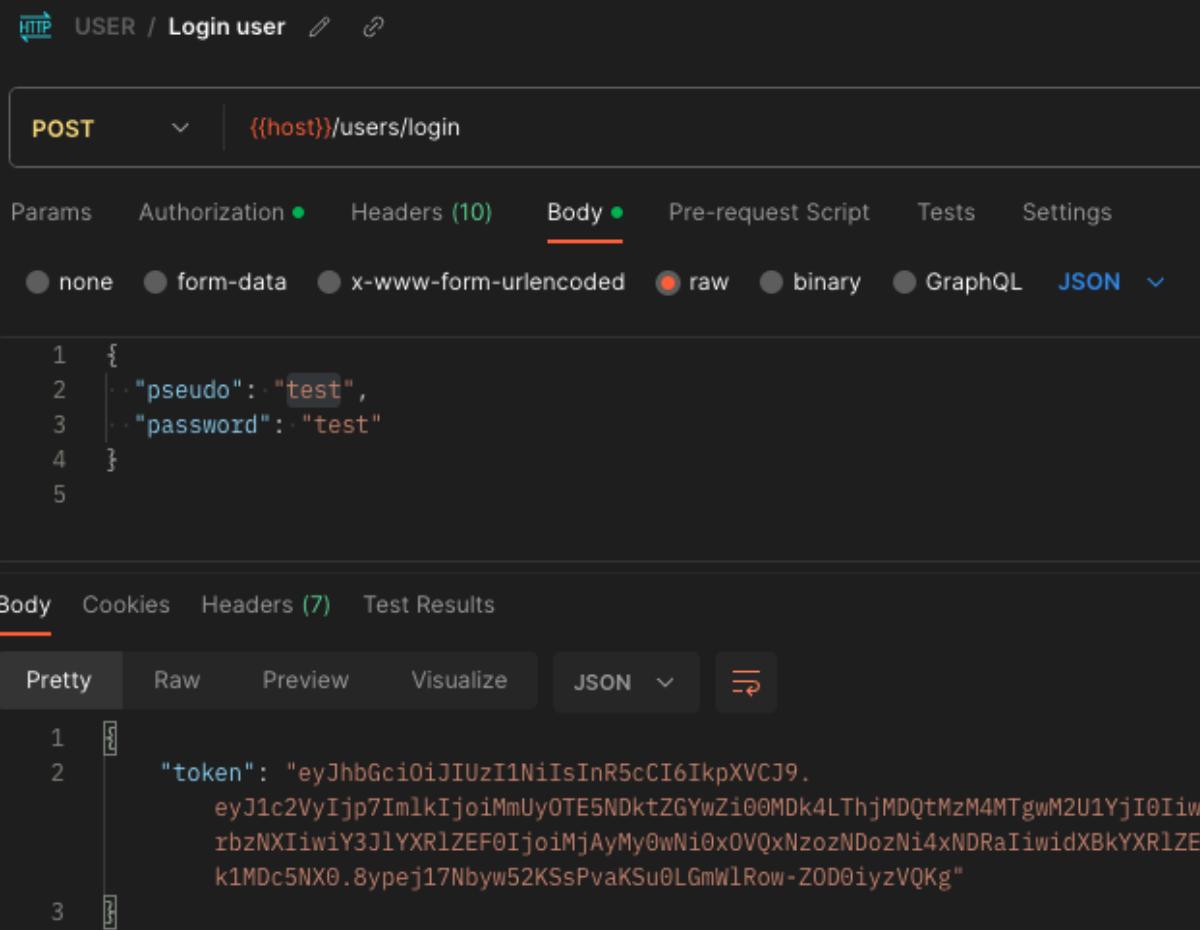


```

1 import { createParamDecorator, ExecutionContext } from '@nestjs/common';
2
3 export const Token = createParamDecorator(
4   factory: (data: unknown, ctx: ExecutionContext) => {
5     const request = ctx.switchToHttp().getRequest();
6     const authorizationHeader = request.headers.authorization;
7     if (authorizationHeader && authorizationHeader.startsWith('Bearer ')) {
8       return authorizationHeader.substring(7);
9     }
10    return null;
11  },
12);
13

```

Voici un exemple du body envoyé pour le login :



HTTP USER / Login user

POST {{host}}/users/login

Body

```

1 {
2   "pseudo": "test",
3   "password": "test"
4 }
5

```

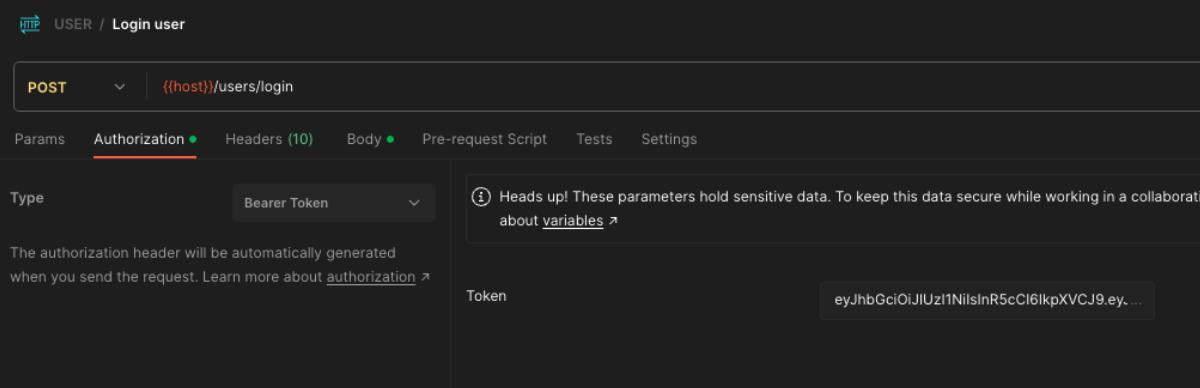
Pretty Raw Preview Visualize JSON

```

1 "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.
2   eyJ1c2VyIjp7ImlkIjoiMmUyOTE5NDktZGYwZi00MDk4LThjMDQtMzM4MTgwM2U1YjI0Iiw
3   rbzNXIiwiY3JlYXR1ZEFOIjoiMjAyMy0wNi0xOVQxNzozNDozNi4xNDRaIiwidXBkYXR1ZE
4   k1MDc5NX0.8ypej17Nbyw52KSsPvaKSu0LGmwlRow-ZD0iyzVQKg"

```

ainsi que l'authorization avec un bearer token (jwt token)



HTTP USER / Login user

POST {{host}}/users/login

Authorization

Type Bearer Token

The authorization header will be automatically generated when you send the request. Learn more about [authorization](#)

Token

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9eyJ...

Une fois le call à l'endpoint, il est renvoyé un token, avec les données de l'utilisateur dans le payload qui serviront au client de pouvoir récupérer les données et d'avoir un accès sécurisé à l'api. Nous reviendrons par la suite sur comment est sécurisé notre API avec l'utilisation des JWT token.

Et voici la fonction qu'appelle le controller, le userService avec la fonction loginUser :

```
1 usage  ✘ vincent55312
async loginUser(
  loginUserDto: LoginUserDto,
  bearerToken: string,
): Promise<{ token: string }> {
  const errors = await validate(loginUserDto);
  if (errors.length > 0) {
    throw new BadRequestException(errors);
  }

  this.jwtAuthService.verifyToken(bearerToken);

  const userExists = await this.prismaService.user.findUnique( args: {
    where: {
      pseudo: loginUserDto.pseudo,
    },
  });

  if (!userExists) {
    throw new BadRequestException( objectOrError: 'User no existing');
  }

  const passwordMatch = await compare(
    loginUserDto.password,
    userExists.password,
  );
  if (!passwordMatch) {
    throw new UnauthorizedException( objectOrError: 'Invalid credentials');
  }

  const token = this.jwtAuthService.generateToken(userExists);
  return { token };
}
```

Il y a un appel à la fonction validate qui va retourner les erreurs du DTO vu précédemment.

- Ensuite une vérification du JWT token.
- Une vérification si l'utilisateur existe.
- Une vérification si le hash du password en base de données correspond avec le mot de passe du client. Et ensuite, le token est généré et retourné.

3.2) Configuration et utilisation de Prisma ORM

Voici le fichier schema.prisma pour la configuration de l'ORM prisma.

```

1  datasource db {
2    provider = "mysql"
3    url      = env("DATABASE_URL")
4  }
5
6  generator client {
7    provider      = "prisma-client-js"
8    binaryTargets = ["native", "linux-musl-arm64-openssl-3.0.x"]
9  }
10
11 model User {
12   id        String @id @default(uuid())
13   pseudo    String @unique @db.VarChar(255)
14   password  String @db.VarChar(255)
15   createdAt DateTime @default(now())
16   updatedAt DateTime @default(now()) @updatedAt
17   userType UserType @default(client)
18   orders    Order[]
19 }
20
21 enum UserType {
22   seller
23   client
24   prospects
25   internal
26 }
27
28 model Product {
29   id        String @id @default(uuid())
30   name     String @db.VarChar(255)
31   description String? @db.VarChar(255)
32   price    Int @db.Int
33   stock    Int @db.Int
34   createdAt DateTime @default(now())
35   updatedAt DateTime @default(now()) @updatedAt
}

```

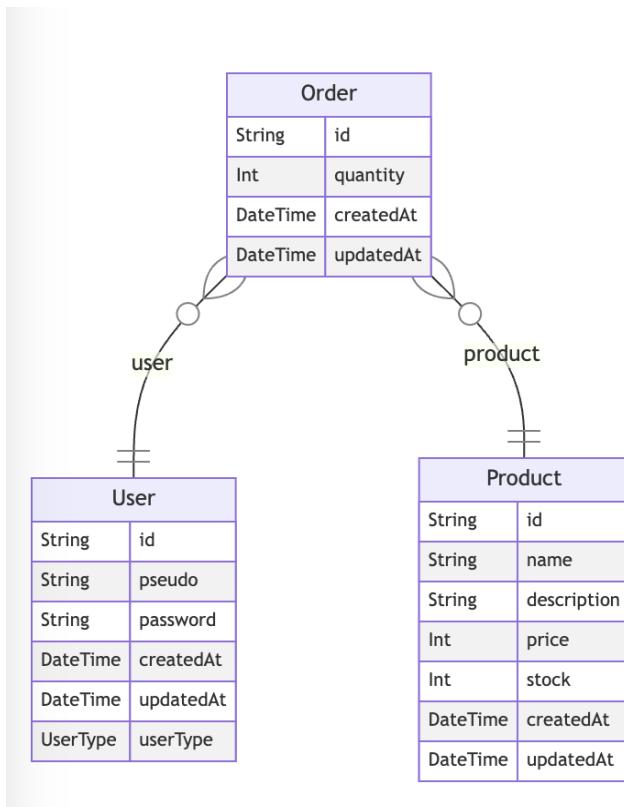
On y retrouve dedans les model, soit les entités utilisées pour l'API.

Les modèles sont considérés comme des objets et constituerons ensemble, l'ensemble des tables de la base de données. L'ORM va gérer pour nous la création de la database, les tables, les jointures, clés publiques, étrangères.

Prisma ORM se configure à partir de l'url de la base de données utilisée. Dans notre cas, voici la variable d'environnement utilisée (dans le .env) :

DATABASE_URL=mysql://\$\$\$\$\$:\$***@54.38.241.241:3306/kawa_db?sslmode=prefer**

Voici le schéma de l'ensemble de nos modèles générés par prisma (utilisation d'un générateur de schéma via prisma.build)



Dans le dossier Prisma, on y retrouve également les migrations. Chaque migration correspond à un changement du **model** (schema.prisma). La migration va contenir le code SQL modifiant la database d'une version X à une version Y. L'avantage des migrations est de pouvoir avoir une base de données pouvant être rollback à des commits antérieurs.

The screenshot shows a code editor interface with a sidebar displaying project files and a main panel showing the content of a migration file. The sidebar includes files like .env, .gitignore, .prettierrc, .eslintrc.js, .env.template, test, src, and prisma/migrations/20230424092012_init/migration.sql. The main panel displays the following SQL code:

```

-- CreateTable
CREATE TABLE `User` (
    `id` VARCHAR(191) NOT NULL,
    `pseudo` VARCHAR(255) NOT NULL,
    `password` VARCHAR(255) NOT NULL,
    `token` VARCHAR(255) NULL,
    `createdAt` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),
    `updatedAt` DATETIME(3) NOT NULL DEFAULT CURRENT_TIMESTAMP(3),
    UNIQUE INDEX `User_pseudo_key`(`pseudo`),
    PRIMARY KEY (`id`)
) DEFAULT CHARACTER SET utf8mb4 COLLATE utf8mb4_unicode_ci;
  
```

Dans l'exemple d'au dessus, c'est la première migration réalisée via notre ORM. Le dossier de migration porte le nom : 20230424032012, correspond à la date où cette migration fut réalisée. Ensuite on y retrouve la table User qui est créée en SQL.

Pour mettre à jour la base de donnée distante, nous devons utiliser la commande : **prisma migrate dev**

Cette commande va permettre d'exécuter les migrations qui ne sont pas encore appliquées dans la base de données.

Par exemple, voici l'exécution de la commande avec aucun changement sur le *model prisma* :

```
vvuylsteker@mbpdevvuylsteker kawa % prisma migrate dev
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": MySQL database "kawa_db" at "54.38.241.241:3306"

Already in sync, no schema change or pending migration was found.

✓ Generated Prisma Client (4.15.0 | library) to ./node_modules/@prisma/client in 87ms
```

On remarque que notre ORM nous dit : Already in sync, no schema change or pending migration was found

Maintenant, procédons à un changement de notre fichier *schema.prisma* avec l'ajout d'un type test sur l'enum *UserType* :

```
enum UserType {
  seller
  client
  prospects
  internal
  test
}
```

On retape la commande : **prisma migrate dev**

Après exécution de cette commande, cette fois-ci, prisma ORM nous demande de saisir en input un texte, relatif à la modification effectuée sur le ***schema.prisma***

```

vvuylsteker@mbpdevvuylsteker kawa % prisma migrate dev
Environment variables loaded from .env
Prisma schema loaded from prisma/schema.prisma
Datasource "db": MySQL database "kawa_db" at "54.38.241.241:3306"

✓ Enter a name for the new migration: ... test
Applying migration `20230621131126_test`

The following migration(s) have been created and applied from new schema changes:

migrations/
└─ 20230621131126_test/
    └─ migration.sql

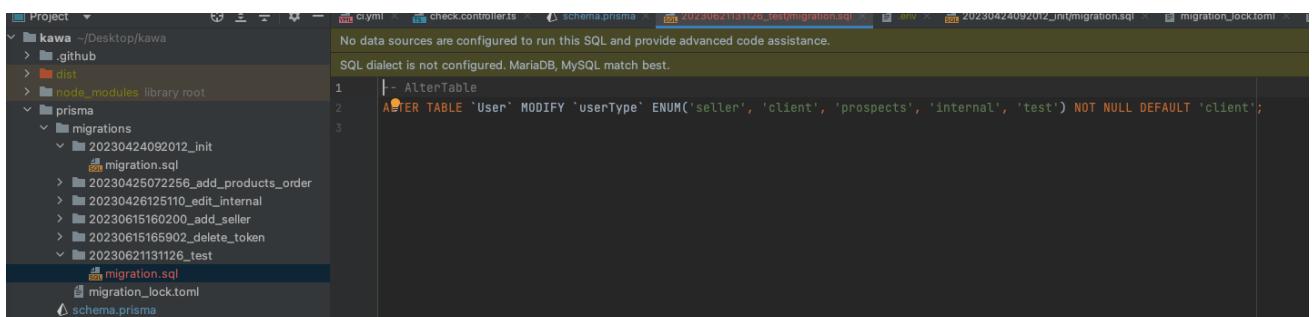
Your database is now in sync with your schema.

✓ Generated Prisma Client (4.15.0 | library) to ./node_modules/@prisma/client in 83ms

```

Pour le test, je l'appelle test. Ensuite, on remarque que la migration s'est bien appliquée au serveur avec le message : **Your database is now in sync with your schema.**

Voici la migration générée par PRISMA ORM durant le process :



```

Project ~ Desktop/kawa
  > node_modules library root
  > prisma
    > migrations
      > 20230424092012_init
        migration.sql
      > 20230426072256_add_products_order
      > 20230426125110_edit_internal
      > 20230615160200_add_seller
      > 20230615165902_delete_token
      > 20230621131126_test
        migration.sql
        migration_lock.toml
      schema.prisma

```

```

No data sources are configured to run this SQL and provide advanced code assistance.
SQL dialect is not configured. MariaDB, MySQL match best.

1   AlterTable
2     ALTER TABLE `User` MODIFY `userType` ENUM('seller', 'client', 'prospects', 'internal', 'test') NOT NULL DEFAULT 'client';
3

```

Vérifions maintenant sur le serveur de base de donnée si la migration s'est bien effectuée :

`userType enum('seller','client','prospects','internal','test')`

Notre base de données distante présente maintenant un nouvel attribut sur notre enum : test.

3.4) Tests unitaires avec jest

Grâce à la librairie jest, nous avons pu développer des tests unitaires fonctionnels et end to end pour tester l'ensemble de notre code.

Pour éviter d'inscrire des données dans la base de données pendant les tests, des mock ont été utilisés. Les mock permettent de désigner ce que vont retourner les fonctions du repository et du service. C'est une méthode relativement intéressante permettant de pouvoir tout tester.

Voici un exemple de test de la classe UserService avec l'utilisation de mock :

Après les instanciations des constructeurs et de ses variables attendues de retour via les mock, nous spécifions à jest avec les **expect** ce qu'il doit retourner et vérifier que la fonction du service de création d'utilisateur fonctionne correctement :



```

1 user-service.spec.ts × .env.template × user.service.ts × app.module.ts × create-user.dto.ts × login-u
28
29
30
31
32
33
34
35
36
37
38     prismaServiceMock.user.create.mockResolvedValue( value: {
39         id: 'userId',
40         pseudo: createUserDto.pseudo,
41         password: hashedPassword,
42         userType: createUserDto.userType,
43     });
44
45     const expectedToken = 'generatedToken';
46     jwtAuthServiceMock.generateToken.mockReturnValue(expectedToken);
47
48     // Act
49     const result = await userService.createUser(createUserDto);
50
51     // Assert
52     expect(result).toEqual( expected: { token: expectedToken } );
53     expect(prismaServiceMock.user.findUnique).toHaveBeenCalledWith( params: {
54         where: {
55             pseudo: createUserDto.pseudo,
56         },
57     });
58     expect(prismaServiceMock.user.create).toHaveBeenCalledWith( params: {
59         data: {
60             id: expect.any(String),
61             password: expect.any(String),
62             pseudo: createUserDto.pseudo,
63             userType: createUserDto.userType,
64         },
65     });
66     expect(jwtAuthServiceMock.generateToken).toHaveBeenCalledWith( params: {
67         id: 'userId',
68         pseudo: createUserDto.pseudo,
69         password: hashedPassword,
70         userType: createUserDto.userType,
71     });
72 });

```

Voici le run des tests unitaires pour le UserService :

```
PASS  test/user/user-service.spec.ts
UserService
  createUser
    ✓ should create a user and return a token (117 ms)
    ✓ should throw BadRequestException if validation errors occur (5 ms)
    ✓ should throw BadRequestException if authentication key is invalid
    ✓ should throw BadRequestException if user with the same pseudo already exists (1 ms)

Test Suites: 1 passed, 1 total
Tests:       4 passed, 4 total
Snapshots:  0 total
Time:        1.252 s, estimated 2 s
```

Tous les tests sont passés. Une autre approche assez en vogue en ce moment est le TDD, test driven development. C'est le fait de d'abord devoir écrire les tests unitaires avant le code fonctionnel en lui-même. Nous avons tenté au début du projet d'appliquer cette méthode, mais malgré le manque de connaissance de certaines personnes du groupe en test unitaire, nous avons vite dû abandonner cette approche et prioriser des tests après le développement du code.

IV Sécurisation de l'API et de l'APP

4.1) Variables d'environnements

Nous avons accordé une grande importance à la sécurisation de notre API dans le cadre de notre projet. Afin de garantir la protection des informations sensibles, nous avons pris des mesures spécifiques pour éviter leur présence directe dans le code source. Tout d'abord, nous avons utilisé un fichier .env et .env.template pour gérer les variables d'environnement de notre projet. Ces fichiers nous permettent de stocker en toute sécurité des informations sensibles telles que les clés d'API et les identifiants de base de données.

Nous avons également ajouté le fichier .env à notre fichier .gitignore, ce qui signifie qu'il n'est pas inclus dans le projet du code source et qu'il reste confidentiel. En suivant cette approche, les informations sensibles ne sont pas exposées dans le code source lui-même. Au lieu de cela, ces informations sont chargées dynamiquement à partir des variables d'environnement lors de l'exécution de l'application.

Cela renforce considérablement la sécurité de notre API en réduisant les risques potentiels de divulgation d'informations confidentielles. Par ailleurs, en ce qui concerne la connexion SSH utilisée dans notre configuration de livraison continue (CD), nous avons adopté une pratique sécurisée. Les informations de connexion SSH, telles que l'hôte, le nom d'utilisateur et la clé SSH, sont stockées en tant que secrets dans GitHub. Ces secrets sont cryptés et ne sont pas visibles dans le code source ou les journaux du projet, ce qui renforce encore la sécurité de notre infrastructure.

Exemple du .env.template présent sur github :

```

1  MYSQL_HOST=
2  MYSQL_USER=
3  MYSQL_PASSWORD=
4  MYSQL_PORT=
5  MYSQL_DB=
6
7  DATABASE_URL=mysql://${MYSQL_USER}:${MYSQL_PASSWORD}@${MYSQL_HOST}:${MYSQL_PORT}/${MYSQL_DB}?sslmode=prefer
8
9  AUTH_KEY=
10 JWT_SECRET_KEY=

```

Ce fichier vise à être modifié par la CI et d'ajouter les variables d'environnements stockées dans github afin de les garder secrets.

Voici le .gitignore permettant d'ignorer le .env afin que le .env ne soit pas ajouté sur le repository github en remote.

```

service.ts x .gitignore x .eslintrc.js x dc
11   yarn-error.log*
12   lerna-debug.log*
13
14   # OS
15   .DS_Store
16
17   # Tests
18   /coverage
19   /.nyc_output
20
21   # IDEs and editors
22   /.idea
23   .project
24   .classpath
25   /.c9/
26   *.launch
27   /.settings/
28   *.sublime-workspace
29
30   # IDE - VSCode
31   .vscode/*
32   !.vscode/settings.json
33   !.vscode/tasks.json
34   !.vscode/launch.json
35   !.vscode/extensions.json
36
37   .env

```

4.2) Github repository secrets

De plus voici un exemple du stockage de variables secrets pour la connexion SSH de notre CD:

Les variables de HOST/ SSHKEY/ USERNAME y sont stockées pour permettre à la CD de se connecter à notre VPS serveur de manière sécurisé.

Repository secrets		
<input type="checkbox"/> HOST	Updated last week	
<input type="checkbox"/> SSHKEY	Updated last week	
<input type="checkbox"/> USERNAME	Updated last week	

4.3) JWT Token

Nous avons choisi d'utiliser des JWT (JSON Web Tokens) en tant que bearer tokens pour l'authentification des requêtes dans notre API, dans le cadre de la sécurité de notre projet. L'utilisation des JWT offre plusieurs avantages en termes de sécurité et de gestion des sessions.

Les données sensibles, telles que les informations d'utilisateur ou d'autorisation, sont encodées dans le payload du JWT. Cela garantit que les données ne sont pas facilement lisibles et protège la confidentialité des informations transmises. Nous avons également mis en place une clé dans le fichier .env pour la génération et la vérification des JWT.

Cette clé est utilisée pour signer et vérifier l'authenticité des tokens, ce qui renforce la sécurité de notre API. Concernant la gestion des sessions, nous avons configuré une expiration de 8 heures pour les JWT. Cela signifie que chaque token créé sera valide pendant cette période et devra être renouvelé par la suite. Cette approche limite la durée de validité des tokens, réduisant ainsi le risque d'utilisation abusive d'un token volé ou compromis.

Pour stocker le token côté client, nous encourageons l'utilisation de mécanismes sécurisés, tels que le stockage local (localStorage) ou les cookies sécurisés avec des attributs appropriés (HTTP Only, Secure). Cela garantit que le token est stocké de manière sécurisée et n'est accessible que par notre application cliente.

À chaque requête vers notre API, nous vérifions la validité du token. Si le token est expiré, invalide ou non fourni, l'accès à l'API est refusé. Cette vérification renforce la sécurité de notre API en s'assurant que seules les requêtes avec un token valide peuvent accéder aux ressources protégées.

Enfin, nous avons également mis en place un mot de passe maître sécurisé pour la génération et la gestion des JWT. Ce mot de passe maître est stocké de manière confidentielle et permet de protéger la clé utilisée pour signer et vérifier les tokens. Cela renforce encore la sécurité globale de notre système d'authentification.

Voici la classe JwtAuthService permettant de générer un token pour une utilisation ou de le vérifier.

```

import { Injectable, UnauthorizedException } from '@nestjs/common';
import { JwtService } from '@nestjs/jwt';
import { User } from '@prisma/client';

4 usages  ↳ vincent55312
@Injectable()
export class JwtAuthService {
    no usages  ↳ vincent55312
    constructor(private readonly jwtService: JwtService) {}

    2 usages  ↳ vincent55312
    generateToken(user: User): string {
        const payload = { user };
        return this.jwtService.sign(payload);
    }

    1 usage  ↳ vincent55312
    verifyToken(token: string): User {
        try {
            const payload = this.jwtService.verify(token);
            return payload.user;
        } catch (error) {
            throw new UnauthorizedException({ objectOrError: 'Invalid token' });
        }
    }
}

```

Et le module inséré dans le framework nest avec la private key de chiffrement :
(JWT_SECRET_KEY)

```

@Module( metadata: {
    imports: [
        JwtModule.register({ options: {
            secret: process.env.JWT_SECRET_KEY,
            signOptions: { expiresIn: '8h' },
        } },
    ],
})

```

Voici un exemple du décodage du payload via un outil comme jwt.io

The screenshot shows the jwt.io interface. At the top, there's a navigation bar with the JUUT logo, Debugger, Libraries, Introduction, Ask, and a Crafted by auth0 by Okta badge. Below the navigation, there's a dropdown menu for Algorithm set to HS256.

Encoded: eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VyIjp7ImlkIjoiMmUyOTE5NDktZGYwZi00MDk4LThjMDQtMzM4MTgwM2U1YjI0IiwicHNldWRvIjoiMSIsInBhc3N3b3JKIjoiJDJiJDEwJDU2LldrWFYTU95YkVVV80akZULy5PNjdMeGc0dXdjsXRmXZQWFR4aVVmNmI2Zi9rbzNXIiwiY3J1YXR1ZEF0IjoiMjAyMy0wNi0x0VQxNzozNDozNi4xNDRaIiwidXBkYXR1ZEF0IjoiMjAyMy0wNi0x0VQxNzozNDozNi4xNDRaIiwidXN1clR5cGUiOjPbnRlcmt5hbCJ9LCJpYXQiOjE2ODcxOTYwNzYsImV4cCI6MTY4ODA2MDA3Nn0.k3q-nLP-mm0ccuOMNwmfujIOAo6MIRh9rjPQR-k6tA4

Decoded:

```

HEADER: ALGORITHM & TOKEN TYPE
{
  "alg": "HS256",
  "typ": "JWT"
}

PAYLOAD: DATA
{
  "user": {
    "id": "2e291949-df0f-4098-8c04-3381803e5b24",
    "pseudo": "1",
    "password": "$2b$10$S6.WkXQXM0ybEUW/4jFT/.067Lxg4uwcIs.uvPXTxiUf6b6f/ko3W",
    "createdAt": "2023-06-19T17:34:36.144Z",
    "updatedAt": "2023-06-19T17:34:36.144Z",
    "userType": "internal"
  },
  "iat": 1687196076,
  "exp": 1688060076
}

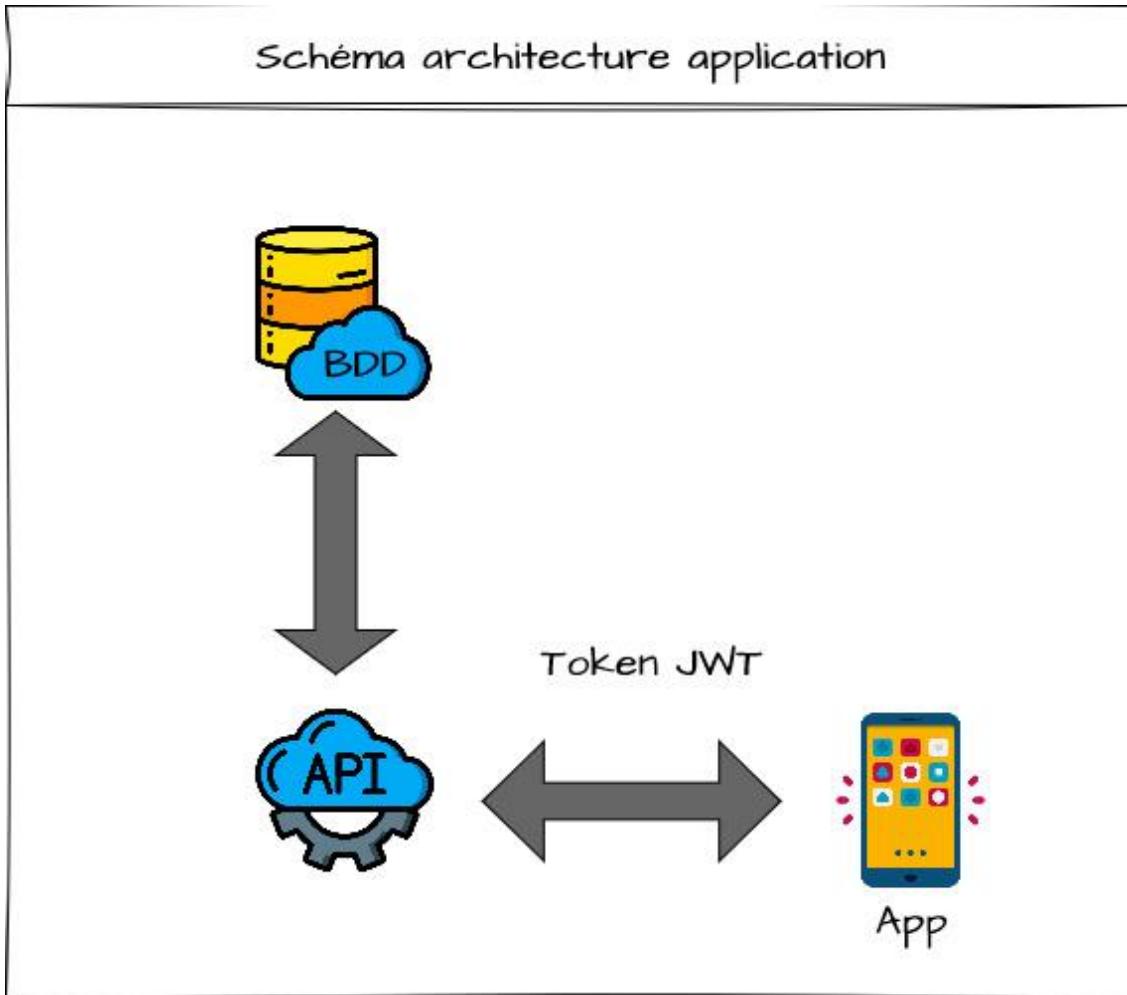
```

Le payload n'est pas chiffré en soi, tout le monde peut accéder aux données, ce n'est uniquement le serveur qui sera capable de vérifier l'authenticité du token avec la signature (en bleu) qui devra correspondre grâce à la private key stockée sur le serveur.

Ici dans notre exemple, on remarque les données de l'utilisateur stockées dans le payload où l'application va après pouvoir récupérer les données.

V- Architecture de l'Application

5.1 Schéma d'architecture de l'application



Dans notre application iOS et Android, nous utilisons le langage de programmation Python couplé avec le framework Kivy. L'application fait appel à une API qui communique avec notre base de données pour effectuer différentes fonctionnalités. Comme par exemple afficher la liste des produits,...

5.2 Écran de connexion

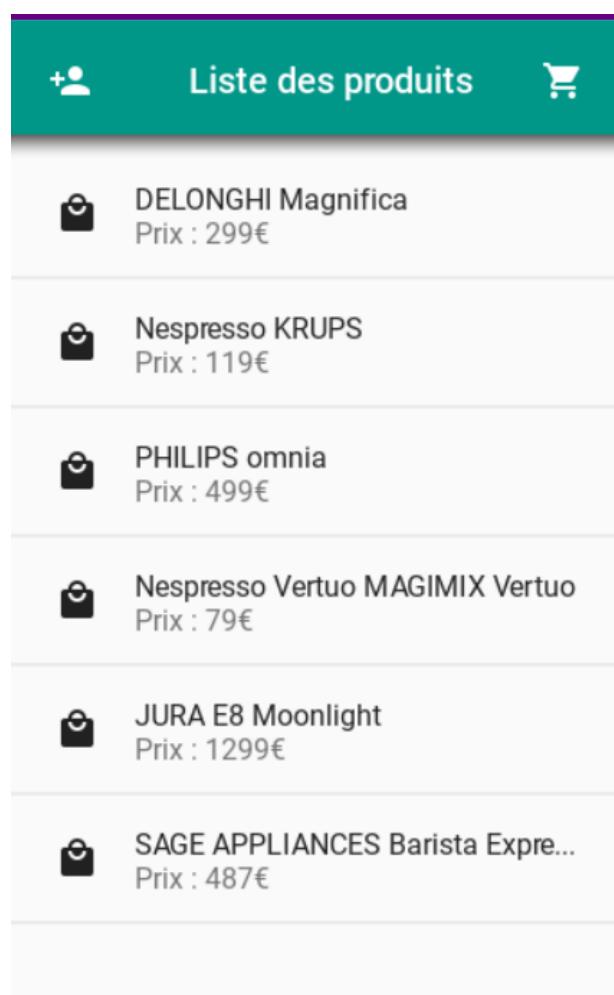
Tout d'abord, nous avons mis en place un système de connexion pour les revendeurs. Lors de la création de leur compte, un QRCode est généré et stocké dans la table **User** de notre base de données. Ce QRCode contient toutes les informations relatives à la personne, ce qui permet au revendeur de se connecter facilement en le scannant.



5.3 Affichage de la liste des produits

L'API permet d'afficher les produits à partir de la table **Product** de notre base de données. Chaque produit est accompagné d'une description détaillée, du prix, du modèle et d'une photo. Ainsi, les utilisateurs peuvent parcourir les produits disponibles dans l'application.

Si vous êtes Administrateur, vous aurez la possibilité d'ajouter un revendeur.



5.4 Détail du produit

L'application offre également la possibilité d'ajouter des produits au panier. Lorsque l'utilisateur sélectionne un produit, il peut l'ajouter à son panier pour l'achat ultérieur.

Une fonctionnalité intéressante que nous avons implémentée est la possibilité d'avoir un aperçu en 3D du produit en réalité augmentée. Cela permet aux utilisateurs de visualiser le produit dans leur environnement réel avant de prendre une décision d'achat.



En plus de ces fonctionnalités principales, nous avons d'autres fonctionnalités que nous détaillerons lors de la présentation. Ces fonctionnalités visent à améliorer l'expérience utilisateur et à rendre l'application plus conviviale et pratique pour les revendeurs et les clients.

VI - Livrable (liens)

serveur api:

<http://54.38.241.241:9090/>

kawa api :

<https://github.com/vincent55312/kawa>

kawa db api :

https://github.com/vincent55312/kawa_db

postman pour api :

<https://www.postman.com/martian-flare-229907/workspace/kawa/collection/2618414-7-eef1542e-016a-4204-bd39-16744a376cdb>

App Mobile:

<https://github.com/Manoarijaona/PayeTonKawa>