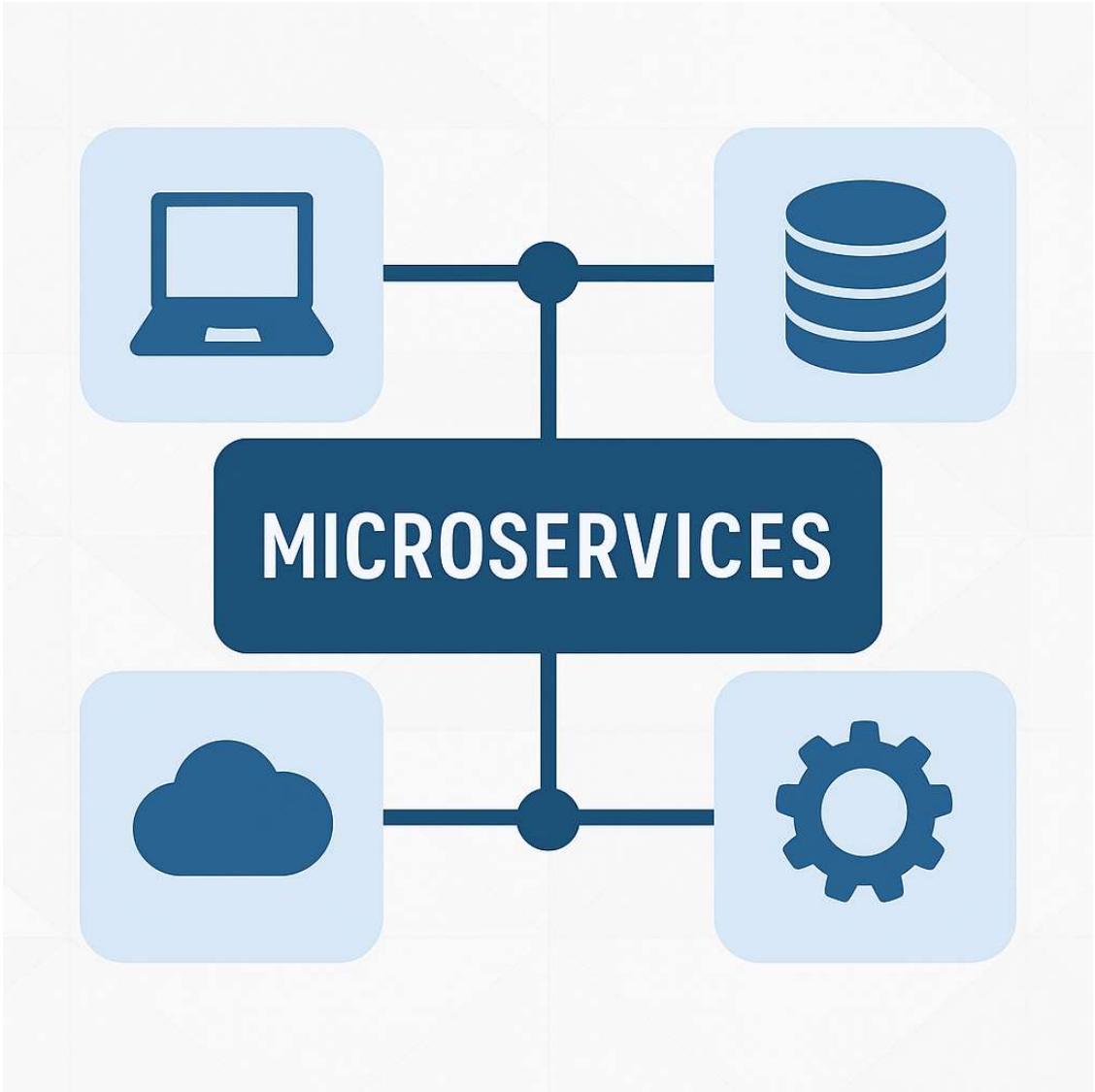
MAURER
Loïc
RT22 – Devcloud FI

Rapport de projet : Système bancaire distribué avec microservices docker



MAURER
Loïc
RT22 – Devcloud FI

Table des matières

Rapport de projet : Système bancaire distribué avec microservices docker 1
Contexte 3
Planification du projet 3
Infrastructure de développement 4
Authentification et gestion des utilisateurs 4
Gestion des comptes bancaires 5
Logging centralisé avec NATS 6
Application Frontend 7
Architecture Docker 8
Limites et sécurité 8
Ce que j'ai appris 8
Fonctionnalités détaillées 8
Documentation des routes API 10
Explication du fichier docker-compose.yml 11
Annexes et ressources 12
Conclusion 12

MAURER
Loïc
RT22 – Devcloud FI

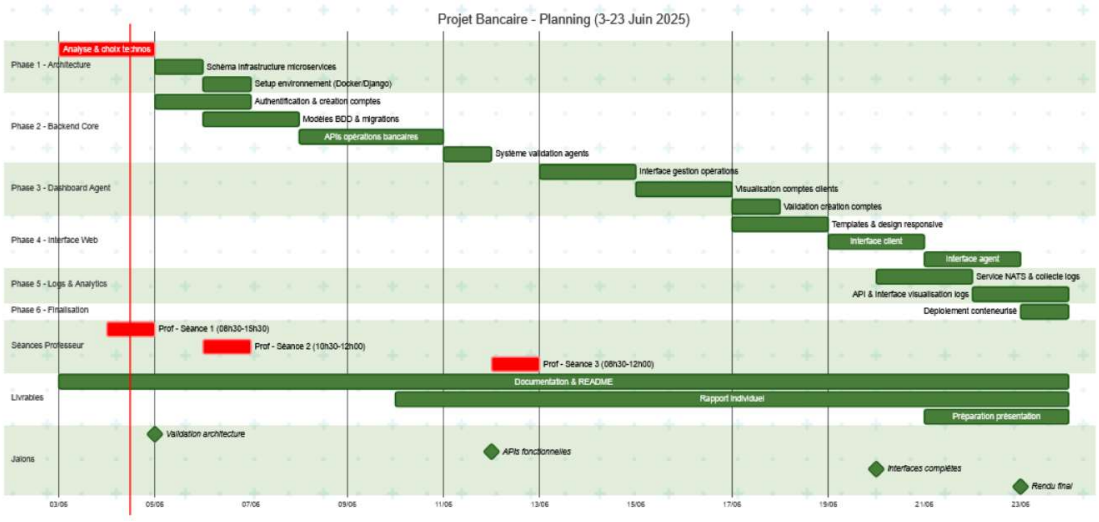
Contexte

Dans le cadre du module DevCloud, le projet visait à concevoir une architecture complète orientée micro-services pour simuler une banque en ligne. L'objectif principal était de séparer les responsabilités en plusieurs services indépendants : l'authentification, la gestion des comptes, le frontend et le logging. Chaque service communique via des API REST Framework Django Rest Framework (DRF), et un système de message NATS assure la centralisation des logs pour un suivi efficace des événements critiques ainsi que l'envoi de mail lors des étapes critiques utilisateurs (login/register). Afin de tenir compte des apprentissages critiques de cette Situation d'évaluation et d'apprentissage (SAE), je me suis tourné vers une virtualisation sous Docker pour sa scalabilité et son évolutivité et sa facilité de déploiement dans des environnements complexes.

Pour finir à la fin de ce projet nous devions fournir un document explicatif de notre projet sur le processus de développement de notre application.

Planification du projet

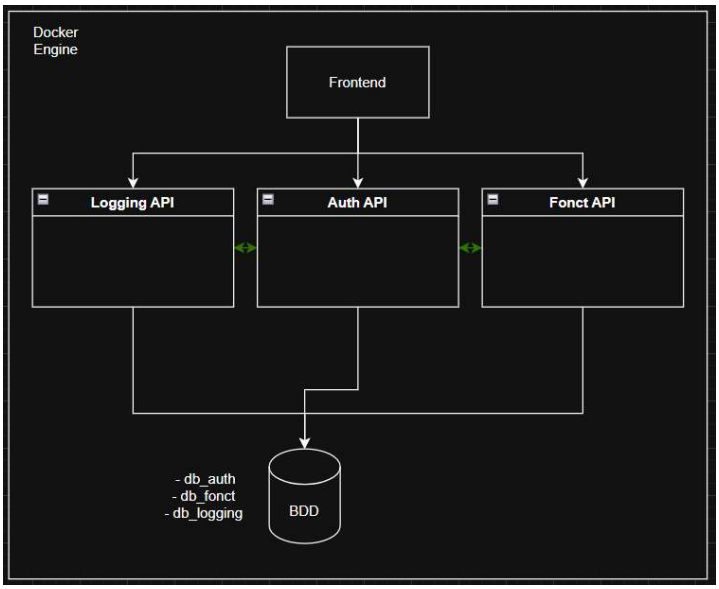
La toute première étape de ce projet a été de diviser le travail en sous fonctionnalités à l'aide d'un diagramme de Gantt, de ce fait je me suis consacré une séance afin de mettre en place le diagramme ci-dessous.



Je me suis tourné vers la méthode des **3QOCP, ainsi que l'utilisation de la matrice d'Eisenhower** pour déterminer les sous étapes ainsi que leurs chronologies dans le développement de l'application.

Infrastructure de développement

Cette partie est destinée à la mise en œuvre d'un schéma d'infrastructure complète final, ce schéma a été créer dans le but de **visualiser l'infrastructure de stockage des données dans les DB du NATS et des données UTILISATEURS**



On peut voir que les APIs communiquent entre elle et que chacun des services dépend de la base de données.

Authentification et gestion des utilisateurs

L'objectif de cette API était de diviser la logique utilisateurs en 3 types de membres (Superuser/Agent/Membre).

Le service `auth_service` gère les utilisateurs, leur connexion, l'inscription, la modification de profil et la réinitialisation de mot de passe. L'API REST de ce service permet également aux agents de superviser les comptes utilisateurs, avec un rôle `is_agent`. La sécurisation des mots de passe se fait via Django et une politique stricte de validation est mise en œuvre pour la réinitialisation.

J'ai également utilisé un module Django qui permet la génération d'un token basic permettant de tracer la connexion et l'utilisation de compte utilisateur sur le site et de sécuriser l'accès aux routes distantes.

```
"GET /login/ HTTP/1.1" 200 8153
"GET /favicon.ico HTTP/1.1" 404 9935
"POST /api/auth/login/ HTTP/1.1" 200 255
"POST /login/ HTTP/1.1" 302 0
"GET /api/auth/users/validate-token/ HTTP/1.1" 200 61
"GET /api/comptes/ HTTP/1.1" 200 2
"GET /api/auth/users/validate-token/ HTTP/1.1" 200 61
"GET /logs/ HTTP/1.1" 200 21663
"GET /dashboard/ HTTP/1.1" 200 9447
```

Par défaut lors de la création de l'application il est nécessaire d'avoir un superutilisateur pour pouvoir créer des agents bancaires.

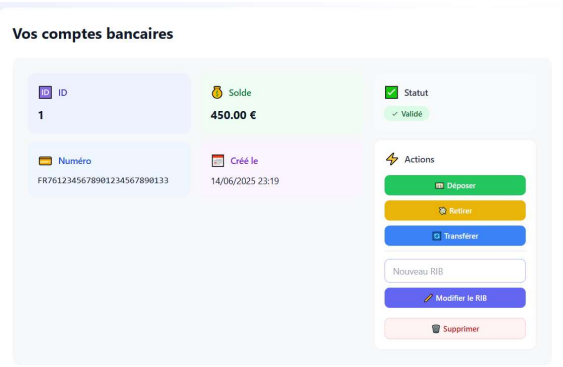
```
Python manage.py createsuperuser
```

MAURER
Loïc
RT22 – Devcloud FI

Gestion des comptes bancaires

Le service `fonct_service` est chargé de la logique bancaire : création de comptes bancaires, virements internes et externes, gestion des RIB entre compte bancaire. Les agents doivent valider manuellement les demandes d'ouverture de comptes bancaires et toutes autres opérations bancaires. Chaque opération déclenche un envoi de log via NATS pour garantir la traçabilité des échanges.

Il est donc nécessaire d'avoir un agent bancaire minimum pour pouvoir faire fonctionner la logique bancaire, sans agent bancaire il est impossible de créer des opérations bancaires



Afin de garantir la sécurité des transactions et des appels entre APIs il a fallu mettre en place un Wrapper permettant de vérifier la validité d'un token en envoyant à l'Api Auth le token qui vérifie la validité du token et de la correspondance utilisateur dans la base de données.

Exemple : Un utilisateur veut créer un dépôt d'argent, il est nécessaire de vérifier s'il est toujours authentifié et si son token est valide

Wrapper permettant la connexion distante entre deux APIs à l'aide du token (source : Medium)

```
class RemoteTokenAuthentication(BaseAuthentication):
    def authenticate(self, request):
        auth_header = request.headers.get('Authorization')
        if not auth_header or not auth_header.startswith('Token '):
            return None
        token = auth_header.split(' ')[1]
        url = 'http://authservice:8000/api/auth/users/validate-token/'
        headers = {'Authorization': f'Token {token}'}
        response = requests.get(url, headers=headers, timeout=5)
        data = response.json()
        user = User(id=data['user_id'], username=data['username'])
        user.roles = [data.get('role', '').lower()]

        return (user, None)
```

MAURER
Loïc
RT22 – Devcloud FI

Logging centralisé avec NATS

Un service `logging_service` reçoit tous les messages importants via un système de messagerie NATS. Un script `listener.py` écoute en permanence les messages et les insère dans une base de données dédiée (script présent dans l'api de LOGGING). Une API REST permet ensuite à l'utilisateur de consulter ses logs, tandis que les agents peuvent filtrer par niveau, type ou utilisateur (gérer côté frontend).

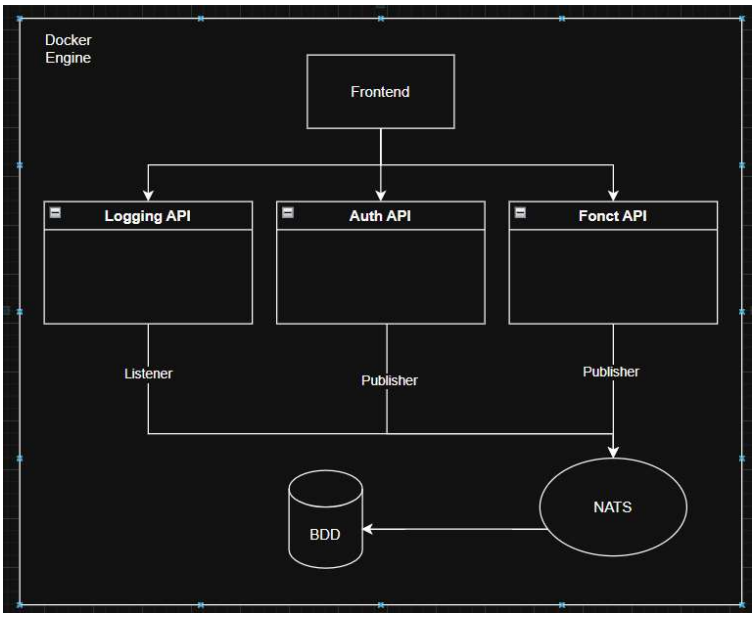


Schéma d'infrastructure NATS

Pour faire fonctionner correctement cette fonctionnalité il est nécessaire d'avoir le fichier `logging.py` dans chaque APIs désirant l'envoi de log sur le canal NATS. Ce fichier permet donc l'envoi de log sur le bon canal NATS ; Enfin le service frontend appellera l'endpoint de l'api LOGGING pour récupérer les logs de la DB.

Exemple d'un fichier logging présent dans une API :

```
import json
import nats
import asyncio
NATS_URL = "nats://natslogging:4222"
async def publish_log(topic, data):
    nc = await nats.connect(NATS_URL)
    await nc.publish(topic, json.dumps(data).encode())
    await nc.flush()
```

```
    await nc.close()


def send_log(topic, data):
    asyncio.run(publish_log(topic, data))
```

Afin de rendre les logs le plus explicite je me suis tourné vers la mise en place de différents types d'opérations / niveaux de sévérité et des canaux NATS différents :

```
topics = ["log.agents.info", "log.agents.error",
"log.agents.warning","log.membres.info", "log.membres.error",
"log.membres.warning"]
```

Application Frontend

Une fois les APIs développé et testé avec une batterie de test sur POSTMAN (=solution d'interrogation d'API). J'ai réalisé le projet django usant les APIs. Pour ce faire j'ai utilisé une logique d'interrogation côté serveur via le module python `request`. Pour ce qui est de la mise en place des pages HTML/CSS par manque de temps et étant seul tout en étant désireux d'avoir une esthétique, je me suis tourné vers le framework TailWindCSS que j'utilise dans de nombreux projets.

Pour la logique backend de l'application, il ne me suffisait que de d'interroger les bons Endpoint et d'en extraire les données nécessaires de manière sécurisée en passant constamment le token en `headers`

```
@token_required
def show_profile_view(request):
    headers = {'Authorization': f"Token {request.session.get('token')}"}
    response = requests.get(f"{AUTH_API_URL}/api/auth/profile/",
headers=headers)
    if response.status_code == 200:
        user_data = response.json(); request.session['user'] = user_data
        return render(request, 'frontend_app/ALL/profile.html', {'user':
user_data})
    else:
        messages.error(request, "Erreur lors de la récupération de votre
profil.")
        return redirect('dashboard')
```

Architecture Docker

L'ensemble du projet est orchestré avec Docker. Le fichier `docker-compose.yml` définit chaque service, ses dépendances, et ses ports. Chaque container est autonome et se connecte à un réseau Docker commun `backend`. Une configuration permet d'initialiser les bases automatiquement au lancement via des commandes `makemigrations`, `migrate` pour chaque service et `createsuperuser` en plus pour le service d'Auth-API qui créer un ADMIN capable de créer des agents bancaires.

```
Python manage.py makemigrations
python manage.py migrate
python manage.py createsuperuser # pour l'api d'authentification
```

Limites et sécurité

Même si chaque service est isolé, certaines limites ont été identifiées :
- **Aucun chiffrement n'est encore appliqué aux communications internes (NATS, APIs).** Il aurait fallu envoyer des messages chiffrés sur le canal NATS avec un système de clé privée/public afin de pouvoir garantir, confidentialité des messages. Pour ce qui est du protocole http, j'aurai pu mettre en place SSL avec la création d'une CA local et de ses certificats pour chaques APIs.

- **La gestion des erreurs peut être améliorée sur certains endpoints** (retour JSON, erreurs 400/500).

- **Le frontend repose fortement sur l'API Auth pour la sécurité.** Il aurait été presque préférable de réaliser la redondance de l'API Auth avec une solution tel que Docker Swarm ou K8s.

- Aucun jeton JWT ou session sécurisée HTTPS n'est encore mis en œuvre. JWT serait une solution plus adaptée pour ce qui est des échanges de token. Dans ma solution seul un token basic est utilisé. Ou ajouter une fonctionnalité de périmassions de token afin d'éviter le partage et l'usurpation d'identité.

Ce que j'ai appris

Ce projet m'a permis de renforcer mes compétences en :

- **Architecture microservices avec Django** : apprentissage et division des services dans différents conteneurs Docker, avec la réalisation de tests sur les conteneurs.

- **Gestion des APIs REST avec Django** : mise en pratique directe des ressources d'apprentissage du framework (DRF).
- **Les décorateurs en Python** : utilisation et implémentation de décorateurs pour éviter la redondance de code dans les vues et pour les vérifications de sécurité.

- **Utilisation avancée de Django Template Language** : lecture et mise en pratique de la documentation sur le langage de template pour manipuler et parser les données passées dans le contexte Django d'une vue.

- **Intégration et déploiement via Docker avec création de tests de santé et d'état des conteneurs (Healthcheck)** : mise en place d'une condition de déploiement des APIs, celles-ci ne se lançant qu'une fois la base de données prête à recevoir des connexions.

- **Utilisation de NATS pour la communication inter-services** : mise en place d'un service NATS (comme vu en TD) pour envoyer et recevoir des logs utilisateurs.

- **Sécurité des opérations et gestion des rôles, ainsi que l'utilisation de tokens pour les APIs** : mise en place d'un système de tokens afin de garantir la traçabilité des connexions et des actions.

- **Mise en place d'un service mail avec SMTP pour les actions critiques des utilisateurs** : utilisation dans les settings de l'interface SMTP de DRF pour envoyer des mails via un relais avant redistribution aux clients effectuant une action.

- **Analyse et traitement des logs bancaires avec utilisation de Chart.js pour la mise en place de différents graphiques** : création d'un tableau de bord visuel et esthétique à l'aide de Chart.js pour la visualisation des logs par les agents bancaires.

Fonctionnalités détaillées

Le projet intègre de nombreuses fonctionnalités organisées selon le rôle de l'utilisateur (client ou agent) et le microservice concerné. Voici un aperçu structuré :

- Authentification (auth_service)

  - Inscription, connexion et déconnexion via API REST sécurisée
  - Réinitialisation du mot de passe avec ancien mot de passe requis
  - Modification de profil utilisateur (nom, email, mot de passe)
  - Rôles `is_agent` permettant des privilèges supplémentaires
  - Création d'un script de publisher NATS pour les logs

  - Envoi de mail pour les étapes critiques d'inscription
  - Vérification du token et de l'utilisateur
  - Suppression, Rejet, Désactivation, Activation d'un utilisateur

- Gestion bancaire (fonct_service)

  - Création de compte bancaire (Suppression + Modification du RIB)
  - Validation manuelle des comptes par un agent ou rejet
  - Virements internes (entre comptes d'un même utilisateur) avec validation d'un agent
  - Virements externes via RIB avec validation d'un agent
  - Historique des transactions disponibles
  - Création d'un script de publisher NATS pour les logs
  - Filtrage des logs en fonction du niveau désiré
  - Retrait / Transfert / Ajout sur le compte bancaire avec validation d'un agent
  - Visualisation des comptes clients pour l'agent

- Logging et supervision (logging_service)

  - Tous les événements critiques sont envoyés à NATS
  - `listener.py` reçoit les messages et les insère dans une base de données
  - Les agents peuvent filtrer les logs par utilisateur, type d'action ou période

- Frontend intégré (frontend_project)

  - Interagit avec tous les services via API
  - Formulaires pour la connexion, la gestion de profil, l'ouverture de compte, les virements
  - Affichage des erreurs, validations, et notifications

Documentation des routes API

Vous pouvez trouver, sur le repository GitHub dans chaque fichier `documentation_api_x` Une documentation complète des routes des différentes APIs

- Extrait typique - Authentification :

  - POST /register/ → Inscription d'un utilisateur
  - POST /login/ → Retourne un token d'accès
  - POST /password/reset/ → Réinitialisation sécurisée avec ancien mot de passe
    Lien vers la documentation : Doc API Auth

- Extrait typique - Comptes bancaires :

  - GET /comptes/ → Liste des comptes d'un utilisateur connecté

- POST /comptes/create/ → Demande d'ouverture de compte
- POST /virement/ → Effectue un virement interne ou externe
  Lien vers la documentation : Doc API Fonct

- Extrait typique - Logs :

  - GET /logs/ → Liste des logs liés à l'utilisateur
  - GET /logs?user=5&type_action=VIREMENT&level=INFO → Filtres spécifiques (agent uniquement)
    Lien vers la documentation : Doc API Logging

Explication du fichier docker-compose.yml

Le fichier `docker-compose.yml` orchestre tous les services de l'architecture en voici l'exemple avec le service de l'api `authservice`:

```
  authservice:
    build: ./auth_service
    container_name: authservice
    command: >
      bash -c "
      until mysqladmin ping -h mysqlbank -ptoto --silent; do
        echo 'Waiting for MySQL...';
        sleep 2;
      done &&
      python manage.py makemigrations &&
      python manage.py migrate &&
      python manage.py createsuperuser --noinput --username admin --email
admin@example.com &&
      echo \"from django.contrib.auth import get_user_model; User =
get_user_model(); user = User.objects.get(username='admin');
user.set_password('adminpassword'); user.save()\" | python manage.py shell &&
      python manage.py runserver 0.0.0.0:8000"
    ports:
      - "8000:8000"
    volumes:
      - ./auth_service:/app
    depends_on:
```

```
      mysqlbank:
        condition: service_healthy
    networks:
      - backend
```

Le principal problème que j'ai pu rencontrer c'était l'ordre de démarrage des conteneurs, en effet il fallait créer le conteneur MYSQL et le démarrer avant celui des APIs, de ce fait j'ai mis en place une vérification permettant de suivre l'état de santé d'un conteneur, ` until mysqladmin ping -h mysqlbank -ptoto –silent` (trouvé sur stackoverflow) une fois la BDD prête mes APIs exécutent les commandes de migrations et de création d'un super utilisateur pouvant créer des agents.

Dans ma logique j'ai également essayé d'utiliser au maximum les variables d'environnements pour garantir une portabilité des constantes.

Et pour finir j'ai réalisé un réseau docker pour que les APIs puissent communiquer entre elles dans l'infrastructure.

Annexes et ressources

Authentification avec tokens DRF

Construire une API d'authentification securisée avec Django

La serialization - DRF

Les contrôles de santé Docker

Verifier connexion MYSQL - Stackoverflow

Django Template Language Doc Basic

Django Template Language Doc (filter)

Les décorateurs en python

Conclusion

Ce projet m'a permis de construire une architecture distribuée complète et fonctionnelle, répondant à des besoins réalistes en matière de banque en ligne. J'ai pu comprendre les fondements de l'approche microservices, la communication inter-service sécurisée, le découplage via les messages, et le déploiement avec Docker.
Il s'agit d'un socle solide sur lequel je pourrai bâtir d'autres projets plus ambitieux dans le futur.