# Rapport de projet : Système bancaire distribué avec microservices docker

Une image contenant texte, Police, symbole, capture d’écran

Le contenu généré par l’IA peut être incorrect.

Table des matières

[Rapport de projet : Système bancaire distribué avec microservices docker 1](#_Toc201487654)

[Contexte 3](#_Toc201487655)

[Planification du projet 3](#_Toc201487656)

[Infrastructure de développement 4](#_Toc201487657)

[Authentification et gestion des utilisateurs 4](#_Toc201487658)

[Gestion des comptes bancaires 5](#_Toc201487659)

[Application Frontend 8](#_Toc201487660)

[Architecture Docker 8](#_Toc201487661)

[Limites et sécurité 9](#_Toc201487662)

[Ce que j’ai appris 9](#_Toc201487663)

[Fonctionnalités détaillées 11](#_Toc201487664)

[Documentation des routes API 12](#_Toc201487665)

[Explication du fichier docker-compose.yml 12](#_Toc201487666)

[Performances de l’application frontend 13](#_Toc201487667)

[Tableau récapitulatif des services : 14](#_Toc201487668)

[Annexes et ressources 15](#_Toc201487669)

[Conclusion 15](#_Toc201487670)

## Contexte

Dans le cadre du module **DevCloud**, le projet avait pour objectif de concevoir une **architecture complète orientée microservices** pour simuler le fonctionnement d'une banque en ligne via des APIs DRF. L’enjeu principal consistait à **dissocier les responsabilités** en plusieurs services indépendants : **authentification**, **gestion des comptes**, **interface frontend** et **logging**.

Chaque service communique via des **API REST** développées avec **Django Rest Framework (DRF)**. Un **système de messagerie NATS** permet de **centraliser les logs** pour assurer un suivi efficace des événements critiques, tout en gérant également l’**envoi d’e-mails automatisés (SMTP & Mailjet)** lors d’actions sensibles telles que la connexion ou l’inscription d’un utilisateur.

Dans une optique d’apprentissage approfondi et pour répondre aux exigences de cette **Situation d’Apprentissage et d’Évaluation (SAE)**, j’ai opté pour une **virtualisation via Docker**, afin de garantir la **scalabilité**, l’**évolutivité** et la **portabilité** de l’ensemble de l’architecture dans des environnements complexes.

Enfin, le projet s’est conclu par la **rédaction d’un document explicatif** détaillant le processus de développement, les choix techniques, ainsi que l’architecture globale mise en place.

## 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, par conséquent je me suis consacré une séance afin de mettre en place le diagramme ci-dessous.  
  
Une image contenant texte, capture d’écran, diagramme, ligne

Le contenu généré par l’IA peut être incorrect.  
  
Pour structurer le développement de l’application, je me suis appuyé sur la méthode des 3QOCP ainsi que sur la matrice d’Eisenhower, afin d’identifier les différentes sous-étapes et de les organiser chronologiquement.

En ce qui concerne le **développement continu**, j’ai adopté une approche **AGILE**, consistant à concevoir d’abord une base d’API solide, puis à y intégrer progressivement de nouvelles fonctionnalités.

Dans cette dynamique, j’ai commencé par définir les **fonctionnalités primaires** telles que l’authentification (inscription, connexion, gestion des tokens), avant d’ajouter des fonctions **secondaires** comme l’envoi d’e-mails ou le système de journalisation (logging).

## Infrastructure de développement

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

Une image contenant capture d’écran, texte, diagramme, Plan

Le contenu généré par l’IA peut être incorrect.

On peut voir que les APIs communiquent entre elles et que chacun des services dépend du service de base de données.

## Authentification et gestion des utilisateurs

L’objectif de cette API était de diviser la logique utilisateur 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 basique permettant de tracer la connexion et l’utilisation du compte utilisateur sur le site et de sécuriser l’accès aux routes distantes.

Cette API est également centrale dans la gestion des authentification des autres APIs car son endpoint /validate-token/ permet d’authentifier, identifier une action utilisateur sur l’application.  
  
Par défaut lors de la création de l’application il est nécessaire d’avoir un super-utilisateur pour pouvoir créer des agents bancaires.

Python manage.py createsuperuser

## Gestion des comptes bancaires

Une image contenant texte, capture d’écran, logiciel, Système d’exploitation

Le contenu généré par l’IA peut être incorrect.Le service `fonct\_service` est chargé de la logique bancaire (logique métier) : création de comptes bancaires, virements internes et externes, gestion des RIB entre comptes bancaires. 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)

Une fois l’utilisateur authentifié et son token valide, le client va donc demander son dépôt vers un compte qu’il aura au préalable validé par un agent. Cette opération donc présente dans sa base de données sur laquelle l’agent à la main et peut refuser ou non cette action.  
  
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 NATS sur les topics (voir suite rapport) et les insère dans une base de données dédiée. 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).

Une image contenant capture d’écran, diagramme, texte, Plan

Le contenu généré par l’IA peut être incorrect.

*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 associé. Ce fichier permet donc l’envoi de log sur le bon canal NATS (publisher simple) ; Enfin le service frontend appellera l’Endpoint de l’api LOGGING pour récupérer les logs de la DB (GET : /logs/).

*Exemple d’un fichier logging.py 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"]  
  
### Les niveaux de logs :   
  
    LEVEL\_CHOICES = [

        ('INFO', 'Info'),

        ('WARNING', 'Warning'),

        ('ERROR', 'Error'),

    ]

### Quelques types d’opérations :

        ('SYSTEME', 'Système'),

        ('OPERATION', 'Opération bancaire'),

        ('UTILISATEUR', 'Utilisateur'),

        ('COMPTE', 'Compte bancaire'),

Application Frontend  
Une fois les APIs développées et testées avec une batterie de test sur POSTMAN (=solution d’interrogation d’API) ; J’ai mis à disposition une collection avec quelques appels d’api dans le repository : *\sae\_bank\_leading-main\Documents\Livrables\* *postman\_collection.json* Il est possible de les importer dans POSTMAN si nécessaire.

J’ai donc réalisé le projet Django en utilisant les APIs. Pour ce faire, j’ai utilisé une logique d’interrogation des APIs côté serveur via le module python `request` et non pas par des fetch de Javascript Vanillia.

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 endpoints 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` ce qui permet l’interrogation des conteneurs par leurs nom de service entre eux.

Une configuration permet également d’initialiser les bases automatiquement au lancement via des commandes `makemigrations`, `migrate` pour chaque service et pour l’API d’authentification `createsuperuser` pour générer un agent bancaire avec les identifiants **admin:adminpassword**

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 (Secure Sockets Layer) avec la création d’une CA locale et de ses certificats pour chaque API.

- **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 avec la mise en place de loadbalancer HAProxy par exemple ou encore traeffik pour gérer des fortes charges.

**- 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 basique est utilisé. Ou ajouter une fonctionnalité de péremption de token afin d’éviter le partage et l’usurpation d’identité.

**- Création d’un reverse Proxy.** Un reverse proxy aurait pu être utilisé afin de masquer les ports des micro-services. Il aurait fallu n’exposer que le port 8001 (application frontend) aux utilisateurs sur le port 80.

## 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.
* **Utilisation de la méthodologie AGILE** : Développement continu et ajout de différentes couches de fonctionnalités
* **Utilisation de la matrice d’Eisenhower**: Pour définir l’importance et la planification GANTT
* **Utilisation d’un modèle CUSTOM pour les utilisateurs Django**: Utilisation de la notion d’héritage pour créer un modèle à partir de la classe AbstractUser de Django

## 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 :

STEP 1 : Première couche AGILE, fonctions primaires

STEP 2 : Couche secondaire, fonctions secondaires

* Authentification (auth\_service)
* Inscription, connexion et déconnexion via API REST sécurisée --- STEP 1
* 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 --- STEP 2
* 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) --- STEP 1
* Validation manuelle des comptes par un agent ou rejet
* Virements internes (entre comptes d’un même utilisateur) avec validation d’un agent
* Retrait / Transfert / Ajout sur le compte bancaire avec validation d’un agent
* Visualisation des comptes clients pour l’agent
* Virements externes via RIB avec validation d’un agent
* Historique des transactions disponibles --- STEP 2
* Création d’un script de publisher NATS pour les logs
* Filtrage des logs en fonction du niveau désiré
* Logging et supervision (logging\_service) (STEP 2)
* 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
* Création des endpoints pour récupérer tous les logs sans filtre
* 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**](https://github.com/LOIC-only-one/sae_bank_leading/blob/main/Documents/auth_api_documentation.md)

* 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**](https://github.com/LOIC-only-one/sae_bank_leading/blob/main/Documents/fonct_api_documentation.md)

* 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**](https://github.com/LOIC-only-one/sae_bank_leading/blob/main/Documents/logging_api_documentation.md)

## 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, par conséquent 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 dans un fichier.env ; Cependant, pour des raisons de simplicité j’ai redéfini directement les variables dans le docker compose

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

## Performances de l’application frontend

Comme expliqué dans la section amélioration dans le cadre d’un déploiement en production il est nécessaire de rendre scalable son infrastructure, c’est pour quoi j’ai utilisé APACHE BENCHMARK afin de déterminer la puissance maximal que l’application peut gérer

*Une image contenant texte, capture d’écran, Police

Le contenu généré par l’IA peut être incorrect.*

Comme nous pouvons le voir :

Une image contenant texte, capture d’écran, Police, noir

Le contenu généré par l’IA peut être incorrect.

| **Élément** | **Signification** |
| --- | --- |
| **Requests per second** : 793.88 [#/sec] | Le serveur peut traiter ~794 requêtes HTTP par seconde. Très bon score pour un service local. |
| **Time per request (mean)** : 12.596 ms | En moyenne, chaque requête met environ **12,6 ms** pour recevoir une réponse (à 10 en parallèle). |
| **Transfer rate** : 3161.56 [Kbytes/sec] | Le débit de réponse du serveur est de ~3 Mo/sec, ce qui montre qu’il peut bien supporter les charges de base. |

## Tableau récapitulatif des services :

| Service | Port | Rôle | Techno | DB utilisée |
| --- | --- | --- | --- | --- |
| auth\_service | 8000 | Authentification, rôles, tokens | Django DRF | auth\_service\_db |
| fonct\_service | 8002 | Logiciel bancaire (comptes, opérations) | Django DRF | fonct\_service\_db |
| logging\_service | 8003 | Centralisation des logs | Django DRF + NATS | logging\_service\_db |
| frontend\_project | 8001 | Interface utilisateur | Django + TailwindCSS | - |
| natslogging | 4222 | Message broker pour logs | NATS Server | - |
| mysqlbank | 3306 | Base de données commune (MariaDB/MySQL) | MySQL | dépend des services |

## Annexes et ressources

[Authentification avec tokens DRF](https://medium.com/django-unleashed/token-based-authentication-and-authorization-in-django-rest-framework-user-and-permissions-347c7cc472e9%20)

[Construire une API d'authentification securisée avec Django](https://dev.to/mayowakalejaiye/mastering-django-building-a-secure-user-authentication-api-from-scratch-4ma3)

[La serialization - DRF](https://www.django-rest-framework.org/tutorial/1-serialization/)

[Les contrôles de santé Docker](https://medium.com/@saklani1408/configuring-healthcheck-in-docker-compose-3fa6439ee280%20)

[Verifier connexion MYSQL - Stackoverflow](https://stackoverflow.com/questions/42567475/docker-compose-check-if-mysql-connection-is-ready)

[Django Template Language Doc Basic](https://docs.djangoproject.com/en/5.2/ref/templates/language/%20)

[Django Template Language Doc (filter)](https://docs.djangoproject.com/en/5.2/ref/templates/builtins/%23ref-templates-builtins-filters)

[Les décorateurs en python](https://python.doctor/page-decorateurs-decorator-python-cours-debutants)

## Conclusion

Ce projet m’a permis de concevoir une architecture distribuée complète et fonctionnelle, répondant à des besoins réalistes en matière de banque en ligne. J’ai pu assimiler les fondements de l’approche microservices, la communication interservices sécurisée, le découplage via la messagerie, ainsi que le déploiement à l’aide de Docker.  
Il constitue une base solide sur laquelle je pourrai construire d'autres projets plus ambitieux à l'avenir.

J’ai trouvé ce projet très enrichissant et formateur. Je remercie l’enseignant pour la qualité du sujet proposé.