

DB DragonBank

Application Bancaire Conteneurisée

Rapport de Projet — Virtualisation et Conteneurisation

Auteurs : Aylane Sehl · Jenson Val · Séri-Khane Yolou
Formation : BUT Informatique 2^{ème} année
Technologies : Docker · Python · Flask · PostgreSQL
Année : 2025 / 2026

Table des matières

1	Introduction	3
1.1	Objectif de la SAE	3
1.2	Fonctionnalités	3
1.3	Stack technique	3
1.4	Choix techniques justifiés	3
1.4.1	Pourquoi PostgreSQL ?	3
1.4.2	Pourquoi Flask et pas Django ?	3
1.4.3	Pourquoi deux conteneurs Flask distincts ?	4
2	Schéma de Base de Données	5
2.1	Présentation générale	5
2.2	Schéma relationnel	5
2.3	Description des tables	6
2.4	Types ENUM et contraintes	6
2.5	Index de performance	6
2.6	Triggers	6
3	Architecture Docker	8
3.1	Vue d'ensemble	8
3.2	Fonctionnement des réseaux Docker	8
3.3	Gestion des dépendances de démarrage	8
3.4	Docker Compose	9
3.5	Dockerfiles	9
3.6	Volumes et persistance	10
4	Application Backend	11
4.1	Architecture en couches	11
4.2	Endpoints de l'API REST	11
4.3	Le pattern DAO	11
4.4	Gestion des erreurs et atomicité	12
5	Service des Intérêts	13
5.1	Configuration	13
5.2	Algorithme de calcul	13
6	Sécurité	14
6.1	Authentification JWT	14
6.2	Tableau des mesures de sécurité	14
6.3	Logs d'audit et conformité	14
6.4	Docker Secrets	15
7	Guide de Démarrage	16
7.1	Lancement	16
7.2	Comptes de test	16
7.3	Commandes utiles	16
8	Conclusion	17

Résumé du projet — À lire en premier

En une phrase : qu'avons-nous construit ?

DragonBank est une application bancaire complète qui tourne dans des **conteneurs Docker** — des environnements virtuels isolés — qui communiquent entre eux de manière sécurisée, sans installation manuelle sur la machine hôte.

Comment ça fonctionne concrètement ?

1. **Le navigateur parle au Frontend** (conteneur **frontend**, port 8080) — les pages HTML, formulaires et boutons. Aucun accès direct à la base de données.
2. **Le Frontend parle au Backend** (conteneur **backend**, port 5000) — le cerveau : il vérifie l'identité via un token JWT, valide les données et applique les règles métier (ex : un seul Livret A par client).
3. **Le Backend parle à la Base de données** (conteneur **db**, port 5432) — le coffre-fort : stockage persistant de tous les utilisateurs, comptes et virements.
4. **Le service Intérêts** (conteneur **interests**) tourne en arrière-plan et recalcule automatiquement les intérêts toutes les **15 minutes**.

Pourquoi Docker ?

Isolation	Si un service plante, les autres continuent de tourner.
Reproductibilité	Fonctionne de la même façon sur n'importe quelle machine.
Sécurité	Le frontend n'a aucun accès direct à la base de données.
Réalisme	Architecture microservices utilisée en production.

Bonus implémentés

Bonus	Explication
Docker Secrets	Mot de passe DB stocké dans un fichier secret monté en mémoire, invisible depuis l'extérieur.
4 ^{ème} conteneur	Service intérêts qui crédite automatiquement les comptes épargne toutes les 15 min.
Healthchecks	Chaque conteneur se vérifie lui-même. Docker attend qu'un service soit «sain» avant de démarrer le suivant.
Simulateur	Page calculant la croissance d'une épargne sur 1 à 40 ans (intérêts composés).
Export CSV	Téléchargement de l'historique des transactions au format Excel.

Pour tester : `docker compose up -build` puis `http://localhost:8080`
Compte 1 : `test-verificationddd@gmail.com` mot de passe : `Test123!`
Compte 2 : `test-verification3@gmail.com` mot de passe : `Azerty09`

Chapitre 1

Introduction

1.1 Objectif de la SAE

DragonBank est une application bancaire développée dans le cadre d'une SAE portant sur la virtualisation et la conteneurisation. L'objectif est de concevoir, développer et déployer une application multi-conteneurs avec Docker.

1.2 Fonctionnalités

Type	Fonctionnalités
Obligatoires	Inscription · Connexion · Comptes (courant, Livret A, Assurance Vie) · Bénéficiaires · Virements (interne, entre personnes, externe) · Historique · Interface web
Bonus	Docker Secrets · 4 ^{ème} conteneur intérêts · Healthchecks · Simulateur d'épargne · Export CSV · Recherche temps réel

1.3 Stack technique

Conteneur	Image de base	Rôle
db	postgres:16-alpine	Base de données PostgreSQL
backend	python:3.12-slim	API REST Flask (port 5000)
frontend	python:3.12-slim	Interface web Flask (port 8080)
interests	python:3.12-slim	Calcul périodique des intérêts

1.4 Choix techniques justifiés

1.4.1 Pourquoi PostgreSQL ?

PostgreSQL a été choisi pour ses fonctionnalités avancées indispensables à une application bancaire : les types `ENUM` pour contraindre les valeurs autorisées (types de compte, statuts de transaction), le type `DECIMAL(15,2)` pour les montants financiers sans erreur d'arrondi, les contraintes `CHECK` pour garantir qu'un solde ne peut pas être négatif, et les transactions `ACID` qui assurent qu'un virement est soit entièrement effectué, soit entièrement annulé.

De plus, PostgreSQL supporte nativement le type `UUID` pour les clés primaires, ce qui évite les collisions entre identifiants et rend impossible de deviner un ID par simple incrémentation.

1.4.2 Pourquoi Flask et pas Django ?

Flask a été choisi pour sa légèreté et sa flexibilité. Pour une API REST, Django apporterait trop de fonctionnalités inutiles (ORM, admin, formulaires). Flask permet de définir précisément ce dont on a besoin : des routes, un mécanisme d'authentification JWT, et une connexion directe à PostgreSQL via `psycopg2`. Cette approche donne un contrôle total sur les requêtes SQL, ce qui est important pour garantir les performances et la sécurité d'une application financière.

1.4.3 Pourquoi deux conteneurs Flask distincts ?

Séparer le frontend et le backend dans deux conteneurs différents présente plusieurs avantages. D'abord, cela respecte le principe de séparation des responsabilités : le frontend ne connaît que les règles d'affichage, le backend ne connaît que la logique métier. Ensuite, cela renforce la sécurité : le frontend n'a aucun accès direct à la base de données, il ne peut interagir avec elle qu'en passant par les endpoints de l'API qui valident chaque requête. Enfin, dans un contexte réel, cela permettrait de scaler indépendamment les deux services selon la charge.

Chapitre 2

Schéma de Base de Données

2.1 Présentation générale

La base dragonbank contient **7 tables**, **2 types ENUM** et **9 index**. Elle est initialisée automatiquement par le fichier `db/init.sql` au premier démarrage du conteneur PostgreSQL.

Diagramme de classes complet : Le schéma de base de données au format PlantUML (avec types, clés, contraintes et multiplicités) est disponible dans le dossier `schema/` du projet.

Fichier : `schema/schemabd.png`

2.2 Schéma relationnel

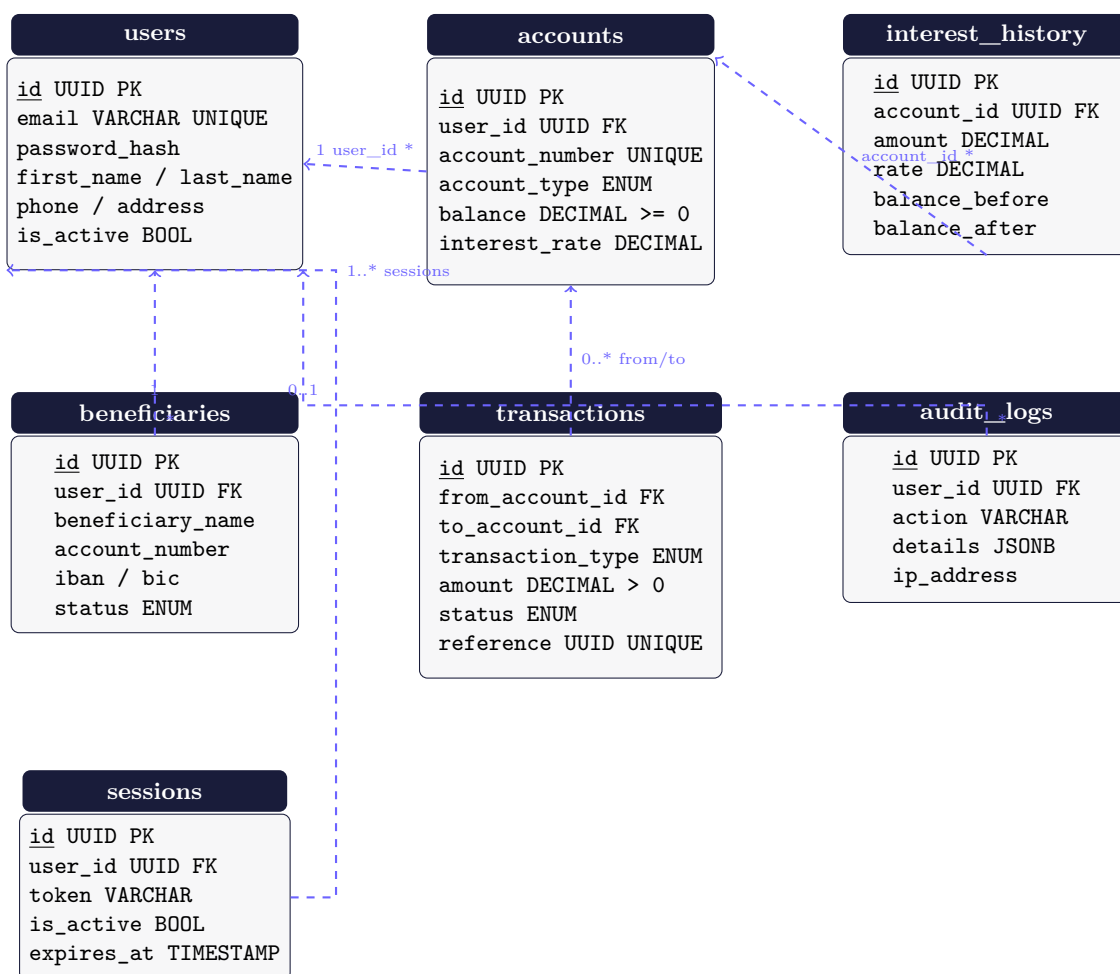


FIGURE 2.1 – Schéma relationnel de la base de données DragonBank

2.3 Description des tables

Table	Description
users	Utilisateurs : email unique, hash bcrypt, profil complet
accounts	Comptes bancaires : type ENUM, solde DECIMAL(15,2), taux d'intérêt
transactions	Opérations : montant, type, statut, référence UUID unique
beneficiaries	Bénéficiaires : nom, banque, IBAN, BIC. Unicité (user_id, account_number)
interest_history	Historique des calculs d'intérêts (solde avant/après, taux appliqué)
sessions	Sessions actives avec IP et User-Agent (sécurité)
audit_logs	Journal de toutes les actions sensibles (RGPD, DSP2)

2.4 Types ENUM et contraintes

Listing 2.1 – Types ENUM et contraintes métier

```

1 CREATE TYPE account_type AS ENUM (
2     'courant', 'livret_a', 'assurance_vie'
3 );
4 CREATE TYPE transaction_type AS ENUM (
5     'virement_interne', 'virement_entre_personnes',
6     'virement_externes', 'depot', 'retrait', 'interets'
7 );
8 CREATE TYPE transaction_status AS ENUM (
9     'pending', 'completed', 'failed', 'cancelled'
10 );
11 -- Contraintes metier
12 CONSTRAINT positive_balance CHECK (balance >= 0)      -- solde jamais negatif
13 CONSTRAINT positive_amount CHECK (amount > 0)          -- montant toujours
14                                     positif
15 UNIQUE(user_id, account_number)                        -- pas de doublon
16                                     beneficiaire

```

2.5 Index de performance

Listing 2.2 – Index pour éviter les full table scans

```

1 CREATE INDEX idx_accounts_user_id ON accounts(user_id);
2 CREATE INDEX idx_transactions_from ON transactions(from_account_id);
3 CREATE INDEX idx_transactions_to ON transactions(to_account_id);
4 CREATE INDEX idx_transactions_created ON transactions(created_at);
5 CREATE INDEX idx_beneficiaries_user_id ON beneficiaries(user_id);
6 CREATE INDEX idx_sessions_user_id ON sessions(user_id);
7 CREATE INDEX idx_sessions_token ON sessions(token);
8 CREATE INDEX idx_audit_user_id ON audit_logs(user_id);

```

2.6 Triggers

Listing 2.3 – Mise à jour automatique du champ updated_at

```

1 CREATE OR REPLACE FUNCTION update_updated_at()
2 RETURNS TRIGGER AS $$
3 BEGIN
4     NEW.updated_at = NOW();
5     RETURN NEW;
6 END;
7 $$ LANGUAGE plpgsql;
8
9 CREATE TRIGGER update_users_timestamp
10 BEFORE UPDATE ON users

```

```
11     FOR EACH ROW EXECUTE FUNCTION update_updated_at();
12
13 CREATE TRIGGER update_accounts_timestamp
14 BEFORE UPDATE ON accounts
15     FOR EACH ROW EXECUTE FUNCTION update_updated_at();
```


Chapitre 3

Architecture Docker

3.1 Vue d'ensemble

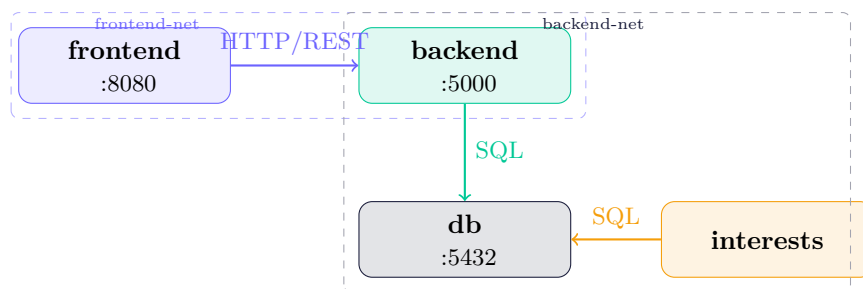


FIGURE 3.1 – Architecture réseau — 2 réseaux Bridge isolés

Le **frontend** n'a aucun accès direct à la base de données. Il passe obligatoirement par l'API backend qui valide chaque requête avec un token JWT.

3.2 Fonctionnement des réseaux Docker

Docker permet de créer des réseaux virtuels isolés entre conteneurs. DragonBank utilise deux réseaux de type **bridge** :

- **dragonbank-frontend-net** : relie uniquement le frontend et le backend. Le frontend peut appeler l'API, mais ne sait pas que la base de données existe.
- **dragonbank-backend-net** : relie le backend, la base de données et le service des intérêts. Ces trois services peuvent communiquer entre eux, mais le frontend n'y a pas accès.

Cette séparation est une mesure de sécurité concrète : même si un attaquant parvenait à compromettre le conteneur frontend, il ne pourrait pas atteindre directement la base de données car ils ne partagent pas le même réseau Docker.

3.3 Gestion des dépendances de démarrage

Une problématique courante avec Docker Compose est l'ordre de démarrage des services. Sans précaution, le backend peut tenter de se connecter à la base de données avant que celle-ci soit prête, ce qui provoque une erreur au démarrage.

DragonBank résout ce problème grâce à la directive `condition: service_healthy` combinée aux healthchecks :

1. La base de données démarre et s'initialise (création des tables via `init.sql`).
2. Docker vérifie toutes les 10 secondes si `pg_isready` répond.

3. Seulement quand la DB est *healthy*, le backend démarre.
4. Seulement quand le backend répond sur `/api/health`, le frontend démarre.
5. Le service des intérêts démarre dès que la DB est prête, indépendamment du backend.

3.4 Docker Compose

Listing 3.1 – Structure du docker-compose.yml

```

1 services:
2   db:
3     build: ./db
4     environment:
5       POSTGRES_PASSWORD_FILE: /run/secrets/db_password
6     secrets: [db_password]
7     volumes:
8       - postgres_data:/var/lib/postgresql/data
9     healthcheck:
10      test: ["CMD-SHELL", "pg_isready -U dragonadmin -d dragonbank"]
11      interval: 10s retries: 5 start_period: 30s
12
13   backend:
14     depends_on:
15       db: {condition: service_healthy} # attend que la DB reponde
16     networks: [dragonbank-backend-net, dragonbank-frontend-net]
17
18   frontend:
19     depends_on:
20       backend: {condition: service_healthy}
21     ports: ["8080:8080"]
22
23   interests:
24     environment:
25       INTERVAL_SECONDS: 900 # calcul toutes les 15 minutes
26     depends_on:
27       db: {condition: service_healthy}
28
29 volumes:
30   postgres_data:
31
32 secrets:
33   db_password:
34     file: ./secrets/db_password.txt

```

3.5 Dockerfiles

Listing 3.2 – Dockerfile backend (même structure pour le frontend)

```

1 FROM python:3.12-slim
2 ENV PYTHONPATH=/app
3 WORKDIR /app
4 RUN apt-get update && apt-get install -y --no-install-recommends \
5     gcc libpq-dev curl && rm -rf /var/lib/apt/lists/*
6 COPY requirements.txt .
7 RUN pip install --no-cache-dir -r requirements.txt
8 COPY . .
9 RUN adduser --disabled-password --gecos '' appuser && \
10     chown -R appuser:appuser /app
11 USER appuser # utilisateur non-root (securite)
12 EXPOSE 5000
13 HEALTHCHECK --interval=15s --timeout=5s --retries=3 \
14     CMD curl -f http://localhost:5000/api/health || exit 1
15 CMD ["python", "app.py"]

```

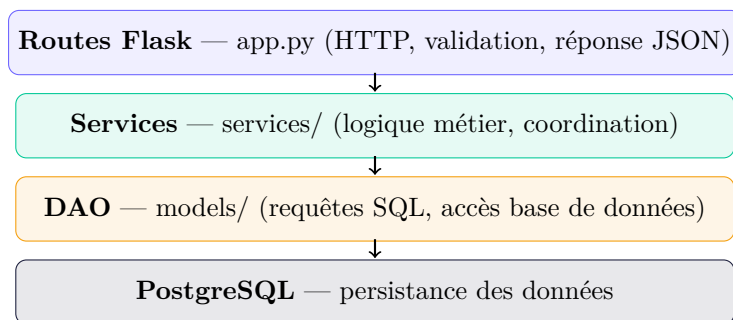
3.6 Volumes et persistance

Sans volume Docker, les données seraient perdues à chaque arrêt. Le volume nommé `postgres_data` survit aux redémarrages. Le fichier `init.sql` n'est exécuté qu'une seule fois, à la création initiale du volume.

Chapitre 4

Application Backend

4.1 Architecture en couches



4.2 Endpoints de l'API REST

Méthode	Route	Description
POST	/api/auth/register	Inscription + compte courant ouvert automatiquement
POST	/api/auth/login	Connexion, retourne token JWT
GET/PUT	/api/user/profile	Profil utilisateur
GET/POST	/api/accounts	Liste / ouverture de compte
GET/POST/DELETE	/api/beneficiaries	Gestion des bénéficiaires
POST	/api/transfers/internal	Virement entre ses propres comptes
POST	/api/transfers/person	Virement vers un bénéficiaire
POST	/api/transfers/external	Virement externe (simulé)
GET	/api/transactions	Historique avec filtres
GET	/api/transactions/export	Export CSV
GET	/api/stats	Statistiques tableau de bord
POST	/api/simulator	Simulation de croissance épargne

4.3 Le pattern DAO

Le pattern DAO (*Data Access Object*) est une technique de conception logicielle qui consiste à séparer la logique d'accès aux données du reste du code. Dans DragonBank, chaque table de la base de données correspond à une classe DAO dédiée.

Par exemple, `CompteDAO` regroupe toutes les opérations SQL liées aux comptes bancaires : lister les comptes d'un utilisateur, en ouvrir un nouveau, débiter ou créditer un solde. Aucune requête SQL liée aux comptes n'apparaît dans `app.py` — elles sont toutes encapsulées dans la classe.

Ce découpage apporte plusieurs avantages concrets :

- **Lisibilité** : une route Flask fait 15 à 20 lignes au lieu de 80. Elle se lit comme un algorithme : valider, vérifier, exécuter, répondre.

- **Réutilisabilité** : la méthode `compte_dao.debiter()` est appelée par les trois types de virement sans dupliquer de code.
- **Maintenabilité** : si la structure de la table `accounts` change, on modifie uniquement `models/compte.py`.

4.4 Gestion des erreurs et atomicité

Toutes les routes utilisent le pattern `try/except/finally`. En cas d'erreur lors d'un virement, `conn.rollback()` annule **toutes** les modifications — on ne peut jamais débiter sans créditer.

Listing 4.1 – Atomicité d'un virement et gestion des erreurs

```
1 conn = creer_connexion()
2 try:
3     montant = valider_montant(donnees['amount']) # Decimal, pas float
4     compte_dao.debiter(conn, id_source, montant) # -X euros
5     compte_dao.crediter(conn, id_destination, montant) # +X euros
6     transaction_dao.enregistrer(conn, ...)
7     conn.commit() # tout valide en une seule transaction SQL
8     return jsonify({'message': 'Virement effectue'}), 200
9 except ValueError as e:
10    return jsonify({'error': str(e)}), 400
11 except Exception as e:
12    conn.rollback() # echec : AUCUNE modification appliquee
13    return jsonify({'error': 'Erreur interne'}), 500
14 finally:
15    conn.close() # toujours ferme la connexion
```

Chapitre 5

Service des Intérêts

Ce 4^{ème} conteneur est un microservice indépendant calculant les intérêts sur les comptes épargne à intervalle régulier.

5.1 Configuration

Listing 5.1 – Configuration dans interests/app.py

```
1 TAUX_LIVRET_A      = Decimal(os.environ.get('INTEREST_RATE_LIVRET_A', '0.03'))
2 TAUX_ASSURANCE_VIE = Decimal(os.environ.get('INTEREST_RATE_ASSURANCE_VIE', '0.02'))
3
4 # Duree en secondes entre deux cycles de calcul des interets.
5 # La valeur 900 correspond a 15 minutes.
6 INTERVALLE_SECONDES = int(os.environ.get('INTERVAL_SECONDS', '900'))
```

5.2 Algorithme de calcul

Listing 5.2 – Calcul avec arrondi bancaire ROUND_HALF_UP

```
1 for compte in comptes_eligibles:
2     solde      = Decimal(str(compte['balance']))
3     taux       = Decimal(str(compte['interest_rate']))
4     interets   = (solde * taux).quantize(
5         Decimal('0.01'), rounding=ROUND_HALF_UP
6     )
7     # Mise a jour atomique : solde + historique + transaction
8     UPDATE accounts SET balance = solde + interets WHERE id = ...
9     INSERT INTO interest_history (amount, rate, balance_before, ...)
10    INSERT INTO transactions (type='interets', amount=interets, ...)
11 conn.commit()    # tout ou rien
```

Chapitre 6

Sécurité

6.1 Authentification JWT

Après la connexion, le serveur génère un token JWT signé. Chaque route protégée est décorée par `@token_requis` qui vérifie la signature avant d'autoriser l'accès.

Listing 6.1 – Génération et vérification du token JWT

```
1 # Generation apres connexion reussie
2 charge_utile = {
3     'user_id': str(id_utilisateur),
4     'exp':     datetime.now(utc) + timedelta(hours=24),
5     'jti':     str(uuid.uuid4()) # ID unique anti-rejeu
6 }
7 token = jwt.encode(charge_utile, SECRET_KEY, algorithm='HS256')
8
9 # Protection d'une route
10 @app.route('/api/accounts')
11 @token_requis # verifie le token avant d'exécuter la route
12 def obtenir_comptes(id_utilisateur_courant):
13     ...
```

6.2 Tableau des mesures de sécurité

Mesure	Implémentation
Mots de passe	Hachage bcrypt coût 12 — jamais stockés en clair
Timing attack	Vérification bcrypt fictive si email inconnu (temps uniforme)
Injections SQL	Requêtes paramétrées (%s) — jamais de concaténation
Docker Secrets	Mot de passe DB dans <code>/run/secrets/</code> (invisible dans <code>docker inspect</code>)
Utilisateur	USER <code>appuser</code> non-root dans chaque Dockerfile
Réseaux	Frontend sans accès direct à la DB
Audit	Table <code>audit_logs</code> : toutes les actions sensibles tracées avec IP

6.3 Logs d'audit et conformité

Chaque action sensible effectuée dans DragonBank est enregistrée dans la table `audit_logs` avec l'identifiant de l'utilisateur, le code de l'action, les détails en JSONB et l'adresse IP d'origine. Les actions tracées incluent notamment : `REGISTER`, `LOGIN`, `TRANSFER_INTERNAL`, `TRANSFER_PERSON`, `TRANSFER_EXTERNAL`, `OPEN_ACCOUNT`, `ADD_BENEFICIARY`, `DELETE_BENEFICIARY`.

Cette traçabilité répond à deux exigences réglementaires importantes :

- **RGPD** : obligation de traçabilité des accès aux données personnelles.
- **DSP2** : obligation de journalisation des opérations de paiement pour permettre les investigations en cas de fraude.

6.4 Docker Secrets

Listing 6.2 – Déclaration et utilisation des secrets

```
1 secrets:
2   db_password:
3     file: ./secrets/db_password.txt    # non commite dans Git
4
5 services:
6   db:
7     secrets: [db_password]
8     environment:
9       POSTGRES_PASSWORD_FILE: /run/secrets/db_password
```


Chapitre 7

Guide de Démarrage

7.1 Lancement

```
1 # Construire et lancer tous les conteneurs
2 docker compose up --build
3
4 # Interface web : http://localhost:8080
5 # API backend   : http://localhost:5000
```

7.2 Comptes de test

Email	Mot de passe
test-fromtest@u-pec.com	Password123
test-verification2@gmail.com	Test1234!

7.3 Commandes utiles

```
1 docker compose logs -f                                # logs en direct
2 docker compose ps                                     # etat + healthchecks
3 docker exec -it dragonbank-db psql \
4     -U dragonadmin -d dragonbank                      # acces PostgreSQL
5 docker compose down -v && docker compose up --build  # remise a zero
6 docker exec dragonbank-db pg_dump \
7     -U dragonadmin dragonbank > sauvegarde.sql        # sauvegarde DB
```

Chapitre 8

Conclusion

DragonBank réunit l'ensemble des exigences de la SAE : 4 conteneurs Docker orchestrés par Compose avec réseaux isolés, volumes persistants, healthchecks et gestion des secrets. Le code backend est organisé en couches (Routes / Services / DAO), l'authentification repose sur JWT et bcrypt, et toutes les actions sensibles sont auditées.

Les bonus sont tous implémentés : Docker Secrets, conteneur intérêts calculant les intérêts toutes les **15 minutes** (INTERVAL_SECONDS=900), healthchecks sur chaque service, simulateur d'épargne avec intérêts composés, export CSV et recherche en temps réel.

Points d'amélioration	Description
Reverse proxy	Ajouter Nginx en frontal pour HTTPS et load balancing
Cache Redis	Sessions et tokens en mémoire pour de meilleures performances
Tests automatisés	Tests unitaires pytest sur les DAO et les validators
CI/CD	Pipeline de déploiement automatique (GitHub Actions)