Documentation Technique

SUPRSS – Outil de gestion de flux RSS

InfoFlux Pro

3 septembre 2025

Table des matières

Contexte du projet

1.1 Présentation générale

La société **InfoFlux Pro**, spécialisée dans les services numériques, souhaite diversifier son offre en proposant un outil interne et externe de veille informationnelle. L'entreprise a mandaté la réalisation d'une application nommée **SUPRSS**, destinée à la lecture et à la gestion de flux RSS.

Cet outil se veut une alternative complète, moderne et intuitive aux solutions déjà existantes telles que *Feedly*, *Inoreader* ou *NewsBlur*, souvent limitées dans leurs fonctionnalités ou soumises à des modèles payants. SUPRSS est pensé pour répondre aussi bien aux besoins individuels (employés cherchant à organiser leur veille) qu'aux besoins collaboratifs (équipes souhaitant mutualiser des sources et échanger autour des contenus).

Le projet doit comprendre à la fois la conception technique (API, base de données, interface web) et la mise en production (containérisation, déploiement via Docker).

1.2 Objectifs

Le développement de SUPRSS poursuit plusieurs objectifs :

1.2.1 Objectifs fonctionnels

- Permettre à chaque utilisateur de **créer un compte** (ou de se connecter via OAuth2).
- Offrir la possibilité de **s'abonner à des flux RSS**, de consulter leurs articles et de les organiser.
- Introduire la notion de **collections personnelles ou partagées**, afin que plusieurs utilisateurs puissent collaborer sur des ensembles de flux.
- Proposer des **outils de recherche et de filtrage** avancés (source, tags, favoris, lus/non lus).
- Mettre en place une **messagerie instantanée** et des **commentaires** sur les articles pour favoriser la discussion.
- Permettre l'**import/export** des flux RSS pour assurer la portabilité des données.

1.2.2 Objectifs techniques

- Concevoir une architecture en **trois briques distinctes** :

  - Une API REST centralisant la logique métier et la communication avec la base de données.
  - Un client web moderne servant uniquement d'interface utilisateur.
  - Une base de données relationnelle stockant utilisateurs, flux, articles et métadonnées.
- Assurer la **sécurité des données** (mots de passe hachés, gestion des permissions, gestion des secrets via variables d'environnement).
- Fournir un environnement de **déploiement automatisé** grâce à Docker et Docker Compose.
- Garantir une **scalabilité** et une **maintenabilité** du code via une architecture claire et modulaire.

1.3 Fonctionnalités principales

Les principales fonctionnalités attendues de l'application SUPRSS sont les suivantes :

- **Authentification et gestion des utilisateurs** :
  - Connexion via email/mot de passe.
  - Connexion via OAuth2 (Google, Microsoft, GitHub, etc.).
  - Gestion des préférences personnelles (mode sombre, taille du texte, etc.).
- **Collections de flux** :
  - Création de collections personnelles ou partagées.
  - Invitation d'autres membres et gestion des rôles (créateur, contributeur, lecteur).
  - Organisation et recherche d'articles au sein de chaque collection.
- **Gestion des flux RSS et des articles** :
  - Ajout d'un flux RSS avec ses métadonnées (titre, description, tags, fréquence de mise à jour).
  - Stockage en base de données des articles pour assurer leur pérennité.
  - Consultation des articles avec leurs attributs (titre, auteur, date, résumé, lien).
  - Marquage d'articles comme *lus/non lus* ou *favoris*.
- **Filtrage et recherche avancée** :
  - Filtrage par source, catégories, statut (lu/non lu) et favoris.
  - Recherche plein texte sur les articles.
- **Collaboration et échanges** :
  - Messagerie instantanée intégrée dans les collections partagées.
  - Commentaires associés à chaque article.
- **Interopérabilité et portabilité** :
  - Import de flux via fichiers OPML, JSON ou CSV.
  - Export des flux dans les mêmes formats.

Architecture du système

2.1 Vue d'ensemble

L'application **SUPRSS** repose sur une architecture classique en trois briques distinctes : un **serveur backend**, un **client web frontend**, et une **base de données relationnelle**, complétés par des composants auxiliaires pour la performance et la communication temps réel.

- **Backend (API)** : développé en **Node.js** avec **NestJS**, écrit en **TypeScript**.
  - **Prisma** comme ORM pour interagir avec **PostgreSQL**.
  - **Redis** pour le cache, la gestion des sessions et la file d'attente.
  - **BullMQ** pour la planification et l'exécution de tâches asynchrones, notamment la récupération périodique des flux RSS.
  - **Socket.IO** pour la communication temps réel (chat, notifications).
  - **Passport.js** pour l'authentification (JWT et OAuth2 avec Google).
  - **bcrypt** pour le hachage des mots de passe.
  - **rss-parser** pour l'analyse des flux RSS et Atom.
  - **xml2js**, **csv-parse**, **csv-stringify** pour les fonctions d'import/export (formats OPML, CSV, JSON).
  - **dayjs** pour la gestion et la manipulation des dates.
- **Frontend (Client web)** : développé avec **Next.js** (framework React) et **TypeScript**.
  - **TailwindCSS** pour la mise en forme et le design responsive.
  - **shadcn/ui** pour des composants réutilisables et cohérents.
  - **lucide-react** pour les icônes.
  - **Framer Motion** pour les animations.
  - **Recharts** pour la visualisation graphique des données.
- **Base de données** : un **PostgreSQL** relationnel pour stocker de manière persistante les utilisateurs, flux, articles, collections, commentaires, et messages. Les migrations et le schéma sont gérés via Prisma.
- **Infrastructure et déploiement** : l'application est conteneurisée via **Docker** et orchestrée par **docker-compose**, regroupant les services suivants :
  - API NestJS.
  - Frontend Next.js.
  - PostgreSQL.
  - Redis.

2.2 Choix d'architecture

Les choix techniques réalisés répondent à des impératifs de performance, de maintenabilité et de scalabilité :

- **Séparation stricte des responsabilités** : le frontend ne contient aucune logique métier, il sert uniquement d'interface et communique avec l'API. L'API centralise toute la logique applicative et la gestion des données.
- **NestJS (backend)** : choisi pour sa structure modulaire, son support natif de TypeScript et sa capacité à intégrer facilement des modules tiers (authentification, WebSockets, ORM).
- **Prisma + PostgreSQL** : combinaison offrant un ORM moderne, type-safe et efficace pour interagir avec une base relationnelle robuste et largement utilisée.
- **Redis + BullMQ** : utilisation d'un système de cache et de file de messages pour optimiser la récupération des flux RSS et gérer des tâches planifiées en arrière-plan.
- **Socket.IO** : nécessaire pour la messagerie instantanée et la mise à jour temps réel des collections partagées.
- **OAuth2 avec Passport.js** : respect des standards modernes d'authentification et intégration simple avec Google.
- **Next.js (frontend)** : framework moderne offrant SSR/SSG, une intégration native avec React et une très bonne optimisation pour la performance et le SEO.
- **TailwindCSS + shadcn/ui** : choix d'une stack UI moderne, rapide à développer, et garantissant une cohérence visuelle.
- **Containérisation via Docker** : simplifie le déploiement, l'isolation des services et la reproductibilité de l'environnement, tout en respectant les contraintes de mise en production.

Cette architecture, modulaire et conteneurisée, assure à SUPRSS une bonne évolutivité et une maintenance facilitée. Elle répond également aux attentes du projet : centraliser la logique sur le serveur, utiliser un client web léger, et assurer une mise en production fiable via Docker.

2.3 Diagramme d'architecture

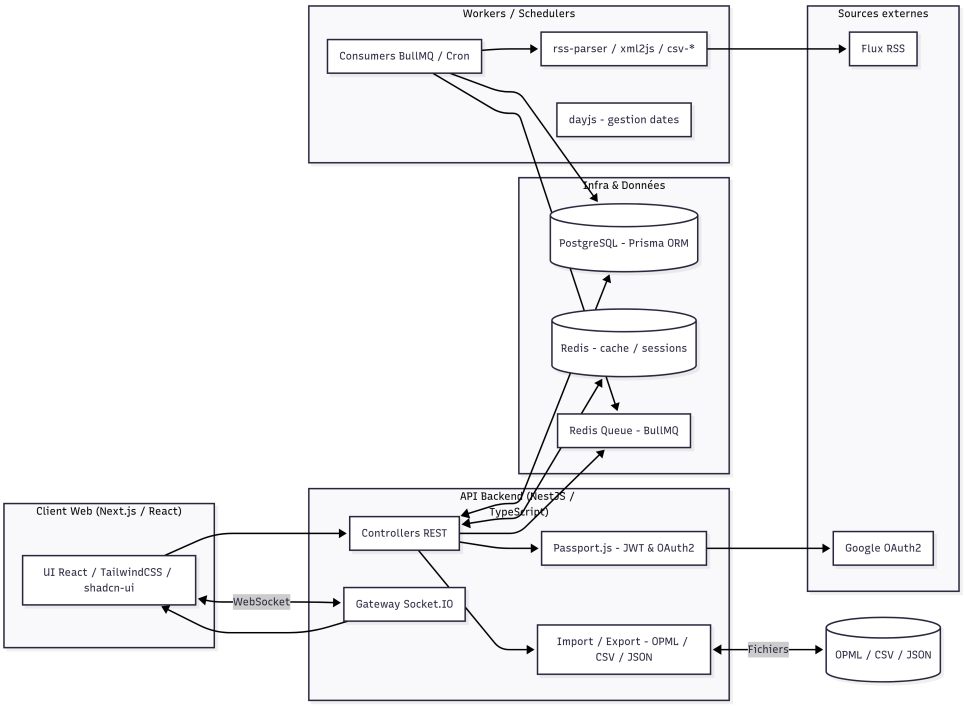

FIGURE 2.1 – Architecture générale de SUPRSS

Base de données

3.1 Schéma relationnel

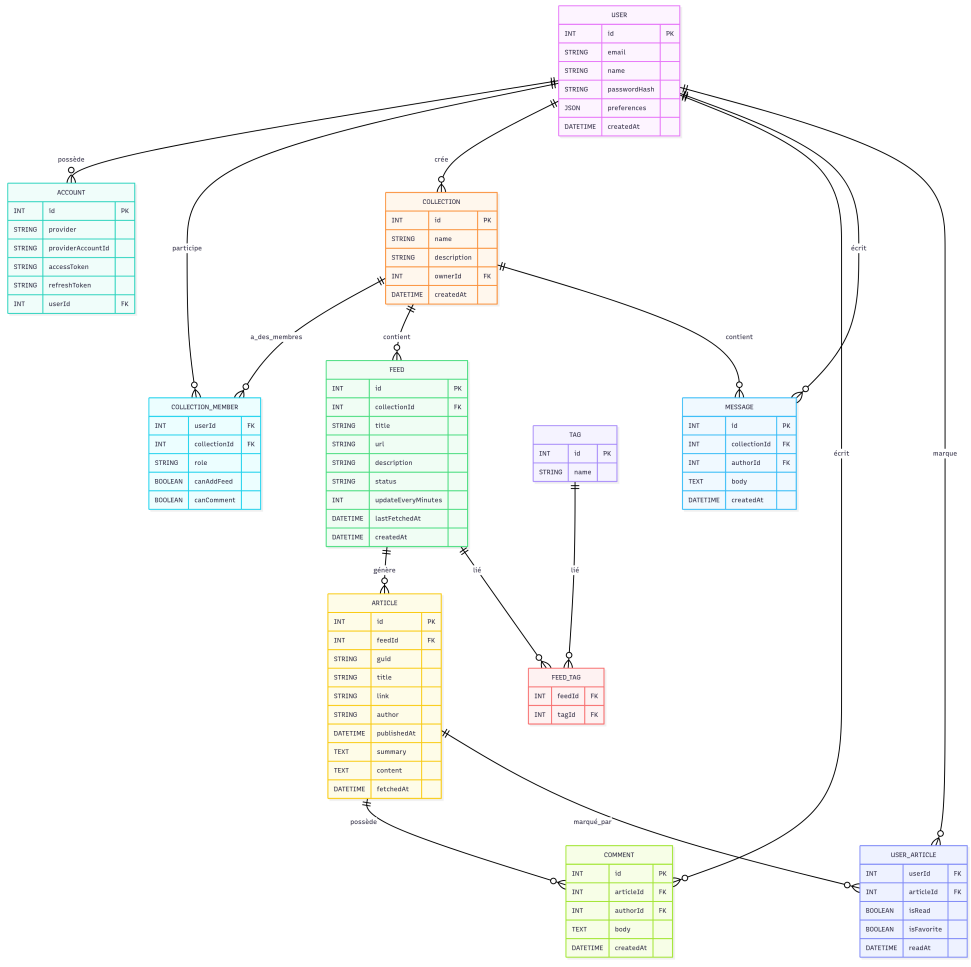

FIGURE 3.1 – Schéma de la base de données

3.2 Justification des choix

Le choix de **PostgreSQL** comme système de gestion de base de données relationnelle repose sur plusieurs critères techniques et fonctionnels :

— **Robustesse et fiabilité** : PostgreSQL est reconnu comme l'un des SGBDR open source les plus stables et les plus utilisés dans le milieu professionnel.
— **Support des relations complexes** : les modèles de SUPRSS nécessitent de nombreuses relations entre entités (utilisateurs, collections, flux, articles, tags). Une base relationnelle est parfaitement adaptée à ce besoin.
— **Richesse fonctionnelle** : PostgreSQL offre des fonctionnalités avancées (transactions ACID, contraintes d'intégrité, support des clés étrangères, gestion fine des index) qui garantissent la cohérence des données.
— **Compatibilité avec Prisma** : l'ORM choisi pour le projet est **Prisma**, dont l'intégration avec PostgreSQL est mature et efficace. Cela permet de bénéficier d'un schéma fortement typé en TypeScript et de migrations automatisées.
— **Extension JSONB** : PostgreSQL permet également de stocker des structures semi-structurées (préférences utilisateur, métadonnées RSS) dans des colonnes de type JSON, ce qui apporte de la souplesse sans compromettre le modèle relationnel.

L'utilisation d'une base relationnelle telle que PostgreSQL, associée à Prisma, garantit donc à la fois une structure solide pour gérer les relations entre entités et la flexibilité nécessaire pour intégrer des données plus dynamiques.

3.3 Gestion des données

La gestion des données dans SUPRSS a été conçue pour assurer à la fois la **pérennité des informations**, la **sécurité des utilisateurs** et la **cohérence du système**.

Stockage des articles RSS

— Chaque flux RSS ajouté à une collection est régulièrement interrogé grâce à un système de tâches planifiées (BullMQ + Redis).
— Les articles récupérés sont **persistés en base de données**, même si le flux d'origine disparaît ou modifie son historique.
— Chaque article stocke : le titre, le lien original, l'auteur (si disponible), la date de publication, un résumé, ainsi qu'un horodatage de récupération.
— Un utilisateur peut associer un statut individuel à chaque article (*lu / non lu*, *favori*) via la table de liaison `USER_ARTICLE`.

Sécurité des données sensibles

— **Mots de passe** : jamais stockés en clair. Ils sont systématiquement hachés avec l'algorithme **bcrypt** avant d'être enregistrés dans la table `USER`.
— **Authentification** : repose sur des **JWT signés** (avec une clé secrète et une durée de validité). Aucun token n'est conservé en clair dans la base.
— **OAuth2 (Google)** : lorsqu'un utilisateur se connecte via Google, les informations de compte externe sont stockées dans la table `ACCOUNT`, liées à l'utilisateur. Les tokens d'accès ne sont pas réutilisés pour la connexion mais uniquement pour lier les comptes.
— **Permissions et rôles** : la table `COLLECTION_MEMBER` définit les droits des utilisateurs dans les collections partagées (ajout de flux, lecture, commentaire, gestion).

Autres données

— Les **préférences utilisateur** (mode sombre, taille de police, etc.) sont stockées dans un champ JSON, offrant de la flexibilité.
— Les **commentaires** et **messages instantanés** sont reliés à la fois aux utilisateurs et aux collections ou articles, garantissant la traçabilité et la cohérence des échanges.

En résumé, la base de données de SUPRSS combine une structure relationnelle robuste avec des mécanismes de sécurité éprouvés, afin de garantir à la fois la **confidentialité des utilisateurs**, la **stabilité des flux RSS archivés** et une **expérience collaborative fluide**.

Diagramme de cas d'utilisation

4.1 Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation permet de représenter les interactions possibles entre les différents types d'utilisateurs et le système **SUPRSS**.

On distingue trois rôles principaux :

— **Propriétaire** : créateur d'une collection, il dispose de l'ensemble des droits (gestion des collections et des flux, organisation et lecture des articles, collaboration).

— **Membre** : utilisateur invité à rejoindre une collection. Il peut gérer les flux, lire et organiser les articles, ainsi que participer aux échanges collaboratifs.

— **Lecteur** : utilisateur disposant d'un rôle restreint. Il peut uniquement consulter et organiser les articles, ainsi que collaborer via la messagerie et les commentaires, mais ne peut pas gérer la collection ni les flux.

Les cas d'utilisation sont regroupés en grandes catégories :

— **Authentification et compte** : inscription, connexion (email/mot de passe ou OAuth2), gestion du profil et des préférences.

— **Collections et flux RSS** : création et gestion des collections, ajout et paramétrage des flux, import/export.

— **Articles** : consultation, recherche, filtrage, marquage (lu/non lu, favoris).

— **Collaboration** : commentaires, messagerie instantanée et notifications.

— **Automatisation** : synchronisation périodique des flux RSS et persistance des articles par le système.

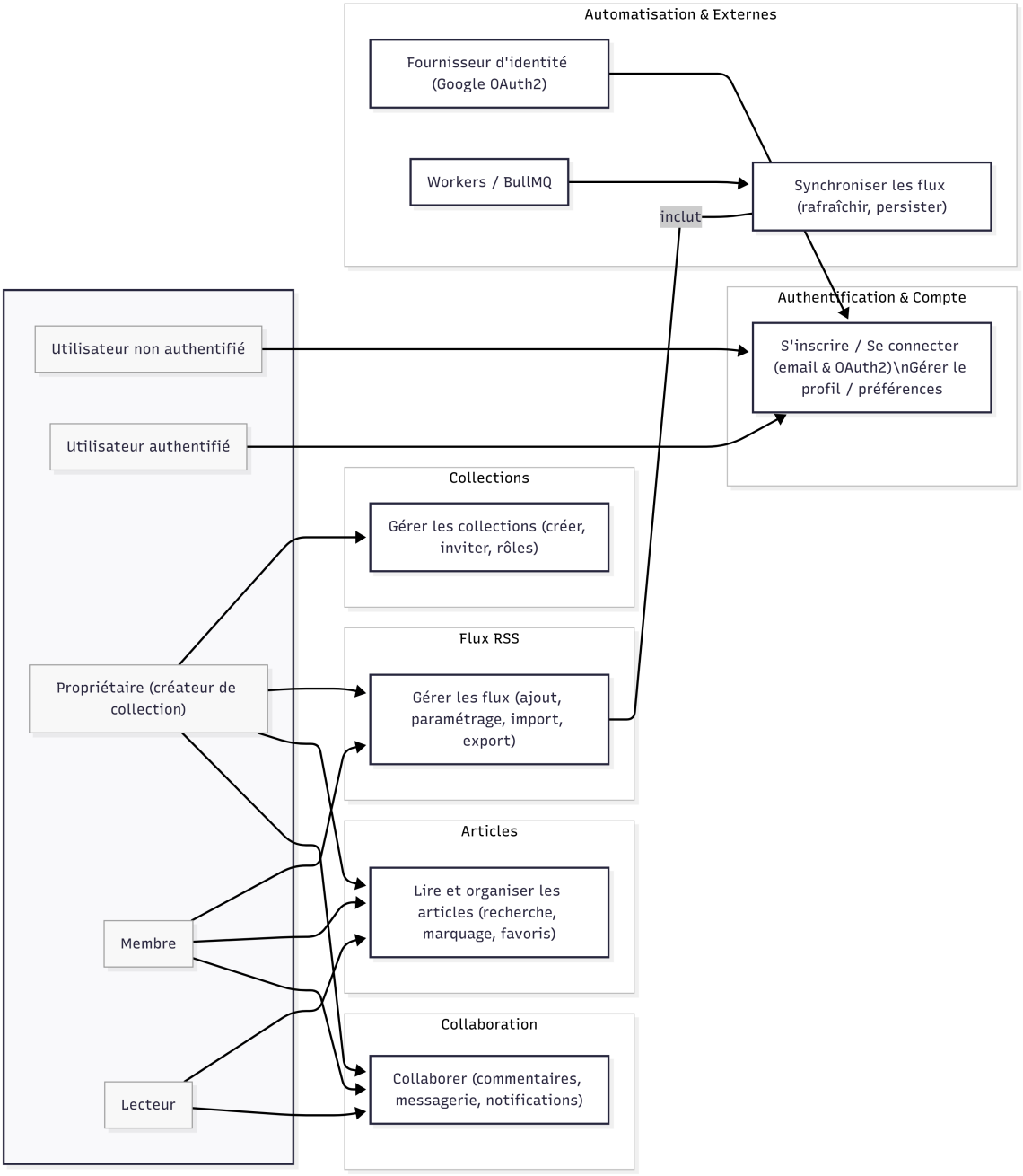

FIGURE 4.1 – Diagramme de cas d'utilisation de SUPRSS

Déploiement et Containérisation

5.1 Docker et docker-compose

L'application **SUPRSS** est intégralement conteneurisée afin de faciliter son installation et son déploiement, aussi bien en local que sur un serveur distant. Le fichier `docker-compose.yml` définit les différents services nécessaires au fonctionnement du système :

- **API (NestJS)** : service backend développé en Node.js/TypeScript. Il expose une API REST et gère l'ensemble de la logique métier (authentification, gestion des flux, collections, articles, commentaires).
- **Client web (Next.js)** : service frontend React. Il sert l'interface utilisateur moderne et communique exclusivement avec l'API.
- **PostgreSQL** : base de données relationnelle où sont stockés les utilisateurs, collections, flux RSS, articles, messages et commentaires.
- **Redis** : utilisé à la fois comme cache et comme gestionnaire de files de messages (BullMQ) pour le traitement asynchrone des flux RSS.

L'ensemble de ces services est orchestré par **docker-compose**, ce qui garantit un environnement reproductible et simplifie la mise en production.

5.2 Guide d'installation et de déploiement

L'installation et le déploiement de SUPRSS reposent sur Docker. Les étapes sont les suivantes :

1. **Cloner le dépôt Git** :

   ```
   git clone https://github.com/noamfvl/SUPRSS.git
   cd suprss
   ```

2. **Configurer les variables d'environnement** : Copier le fichier `.env.example` vers `.env` pour chaque service (API et client web), puis adapter les valeurs en fonction de votre environnement.
3. **Construire et lancer les conteneurs** :

   ```
   docker-compose up --build
   ```

   Cette commande télécharge les images nécessaires, construit les services et démarre l'ensemble de l'application.

4. **Accéder à l'application** :
   — Interface web (Next.js) : `http://localhost:5173`
   — API backend (NestJS) : `http://localhost:3000`

5.3 Environnements

Le projet utilise des fichiers `.env` pour stocker les variables d'environnement nécessaires au bon fonctionnement de chaque service. Ces fichiers ne doivent pas être partagés publiquement car ils contiennent des informations sensibles (mots de passe, clés secrètes).

Exemples de variables utilisées

— **API (NestJS)** :
  — `DATABASE_URL` : chaîne de connexion PostgreSQL.
  — `REDIS_URL` : connexion à Redis.
  — `JWT_SECRET` : clé secrète utilisée pour signer les tokens JWT.
  — `GOOGLE_CLIENT_ID`, `GOOGLE_CLIENT_SECRET` : identifiants OAuth2 pour l'authentification via Google.
— **Client web (Next.js)** :
  — `NEXT_PUBLIC_API_URL` : URL de l'API backend.
— **Base de données (PostgreSQL)** :
  — `POSTGRES_USER`, `POSTGRES_PASSWORD`, `POSTGRES_DB` : informations de connexion.