Architecture Technique — Jira Issue Extractor

Vue d'ensemble

Jira Issue Extractor est une application web multi-conteneurs construite sur une architecture microservices légère. Elle repose sur un backend FastAPI (Python 3.11), un frontend React/Vite, une base PostgreSQL, un broker de messages Redis et un stockage objet MinIO. Un sixième conteneur dédié gère l'exécution des planifications périodiques.

Conteneurs et rôles

`frontend` — Interface utilisateur

- **Technologie** : nginx (alpine) + React 18 / Vite / Tailwind CSS
- **Rôle** : Sert l'application single-page (SPA) et proxifie toutes les requêtes `/api/*` vers le backend.

- **Port exposé** : 80 (configurable via `FRONTEND_PORT`)
- **Artefact** : Le build Vite (`dist/`) est inclus dans l'image via un build multi-stage.
- **Particularité** : Désactivation du buffering nginx pour les flux SSE (`proxy_buffering off`), timeouts de 3600 s pour les exports longs.

`backend` — API et moteur d'extraction

- **Technologie** : FastAPI + Uvicorn (Python 3.11)
- **Port interne** : 8080
- **Rôle principal** :
  - Exposer l'API REST (authentification, connexions Jira, lancement d'exports, historique, etc.)
  - Exécuter les extractions Jira en tâches asynchrones (`asyncio.create_task`)
  - Streamer les événements de progression en temps réel (SSE)
  - Servir les fichiers exportés pour la prévisualisation navigateur
- **Démarrage** : `python -m backend.main` → initialisation DB + migrations Alembic + démarrage Uvicorn
- **Scheduler intégré** : Si `JIRA_EXTRACTOR_ENABLE_SCHEDULER=true` et qu'aucun conteneur scheduler dédié n'est présent, la boucle de planification tourne dans le processus backend.

`scheduler` — Planificateur d'exports

- **Technologie** : Python 3.11 (même image que le backend)
- **Rôle** : Boucle infinie (tick 30 s) qui interroge la table `schedules`, lance les exports arrivés à échéance et met à jour `next_run_at`.
- **Communication** :
  - Lit/écrit dans PostgreSQL (accès direct)
  - Publie les événements de job sur Redis (même bus que le backend)
- **Démarrage** : `python -m backend.scheduler.main`
- **Isolation** : Séparé du backend pour permettre un scaling indépendant et éviter que les planifications bloquent les requêtes HTTP.

`postgres` — Base de données relationnelle

- **Image** : `postgres:16-alpine`
- **Rôle** : Source de vérité pour toutes les données persistantes : connexions, jobs, utilisateurs, tokens d'authentification, configuration, planifications, répertoires.
- **Migrations** : Alembic (lancé au démarrage du backend via `subprocess.run`)
- **Volume** : `postgres_data` (persistant)

`redis` — Bus de messages

- **Image** : `redis:7-alpine`
- **Rôle** : Broker pub/sub pour diffuser les événements SSE entre le backend et le scheduler.

- **Canal** : `job:{job_id}` — chaque job a son propre canal.
- **Mode dégradé** : Si Redis n'est pas configuré (`JIRA_EXTRACTOR_REDIS_URL` vide), le streaming SSE se fait via une `asyncio.Queue` en mémoire (fonctionnel en single-node uniquement).
- **Persistance** : Non — ephémère par design.

`minio` — Stockage objet

- **Image** : `minio/minio`
- **Rôle** : Stockage S3-compatible des exports (fichiers HTML, XML, JSON, CSV, ZIP) après leur création.
- **URI interne** : `minio:jobs/{job_id}/{export_name}/` (répertoire) ou `minio:jobs/{job_id}/{name}.zip` (archive)
- **Mode dégradé** : Si MinIO n'est pas configuré, les exports restent sur le volume local `exports_data`.
- **Accès** : API S3 sur le port 9000 ; console admin sur 9001.
- **Volume** : `minio_data` (persistant)

Flux de communication inter-services

1. Lancement d'un export

2. Streaming SSE

3. Planification

4. Prévisualisation d'un fichier exporté

Authentification et sécurité

Mécanisme	Implémentation
JWT (Bearer)	`python-jose` / HS256, clé `JWT_SECRET_KEY`, TTL configurable
Hachage des mots de passe	`bcrypt` (min. 12 tours)
Tokens email	UUID haché SHA-256 en base, TTL court (invite, reset)
Browse token	Usage unique, échangé contre cookie HttpOnly `browse_{job_id}`
RBAC	3 rôles : `admin` (toutes opérations), `editor` (exports, connexions), `visitor` (lecture seule)
OAuth2 Jira	Flux authorization code avec callback `/api/oauth2/callback`

Persistance et stockage

Donnée	Stockage	Format
Connexions Jira	PostgreSQL	Table `connections`
Jobs et historique	PostgreSQL	Table `jobs` (events en JSON)
Utilisateurs	PostgreSQL	Table `users`
Planifications	PostgreSQL	Table `schedules`
Fichiers exportés	MinIO (ou volume local)	Arborescence ou ZIP
Préférences utilisateur	PostgreSQL	JSON dans `users.preferences`
Config globale	PostgreSQL	Table `app_config` (clé-valeur JSON)

Modes de déploiement

Mode complet (6 conteneurs)

Production recommandée. Redis assure la communication SSE cross-container. MinIO assure la persistance des exports.

Mode simplifié (backend + frontend + postgres)

Sans Redis ni MinIO. Les exports restent sur le volume local, le SSE fonctionne en mémoire. Adapté à un usage mono-instance sans haute disponibilité.

Mode développement

Backend : `uvicorn backend.main:app --reload` (port 8080)

Frontend : `npm run dev` (port 5173, proxy vers 8080)

Base : PostgreSQL local ou SQLite (fallback automatique via `aiosqlite`)

Dépendances externes

Service	Usage	Obligatoire
Jira Cloud / Server	API REST pour les extractions	Oui

Service	Usage	Obligatoire
Serveur SMTP	Envoi des invitations et réinitialisations de mot de passe	Non
PostgreSQL	Base de données principale	Oui (ou SQLite en dev)
Redis	Bus d'événements SSE multi-conteneurs	Non
MinIO	Stockage objet des exports	Non