Rapport Technique Détaillé - CryptoTracker

1. Vue d'Ensemble

CryptoTracker est une plateforme web d'échange et de suivi de cryptomonnaies pair-à-pair (P2P). Elle permet aux utilisateurs de s'inscrire, de consulter les cours du marché en temps réel, de gérer un portefeuille de transaction, et de publier des offres d'achat/vente. Une interface d'administration permet la gestion complète des utilisateurs.

L'application est construite sur une architecture **Monorepo** séparant clairement le Backend (API) du Frontend (SPA).

---

2. Architecture Backend

2.1 Stack & Environnement

- **Serveur** : Express.js (v5.2.1) sur Node.js.
- **Base de Données** : SQLite (fichier local `database.sqlite`).
- **ORM** : Sequelize pour le mapping relationnel objet.
- **Sécurité** : `bcrypt` (hachage), `jsonwebtoken` (JWT), `cookie-parser`.

2.2 Modélisation des Données (Schémas Sequelize)

1. User (`models/User.js`)

Représente un acteur du système.

- **Champs** :
- `username` (String, unique)
- `email` (String, unique, validé)
- `password` (String, haché bcrypt)
- `role` (Enum: `item` (défaut), `admin`)
- `favoriteCoin` (String, ex: "BTC")
- `profile` (String, chemin vers l'image avatar)
- `socials` (JSON ou champs séparés: facebook, twitter, linkedin, instagram)

2. Offer (`models/Offer.js`)

Représente une offre de vente publiée sur le marché.

- **Champs** :
- `coin` (String, ex: "ETH")
- `amount` (Decimal, quantité totale)
- `price_usd` (Decimal, prix unitaire)
- `type` (Enum: `sell`)

- `status` (Enum: `open`, `closed`)
- `remaining` (Decimal, quantité restante à vendre)
- **Relations** : Appartient à un `User` (alias `seller`).

3. Transaction (`models/Transaction.js`)

Historique immuable des opérations.

- **Champs** :
- `coin`, `amount`, `price_usd`
- `type` (Enum: `buy`, `sell`)
- `date` (Timestamp)
- **Relations** : Liée à un `User`.

2.3 API RESTfult & Endpoints

Authentication (`/api/auth`)

- `POST /register` : Création de compte. Hachage du mot de passe (salt rounds: 10).
- `POST /login` : Authentification. Génère un JWT (durée 1 jour) stocké dans un cookie **HTTP-Only** (`token`).
- `POST /logout` : Suppression du cookie d'authentification.

Gestion Utilisateur (`/api/user`)

- **Middlewares appliqués** : `verifyToken` (authentification requise).
- `GET /me` : Renvoie les infos de l'utilisateur connecté via le décodage du token.
- `PUT /me` : Mise à jour du profil (bio, réseaux sociaux, mot de passe).
- `DELETE /me` : Suppression de son propre compte.
- `POST /avatar` : Upload d'image de profil (via middleware `multer`), stockée dans `/uploads` ou `/public`.

Administration (`/api/user` - Admin Only)

- **Middlewares appliqués** : `verifyToken`, `isAdmin`.
- `GET /` : Liste tous les utilisateurs (exclut les mots de passe).
- `PUT /:id` : Modifie n'importe quel utilisateur (rôle, infos).
- `DELETE /:id` : Bannissement/Suppression d'un utilisateur.

Offres & Marché (`/api/offers`)

- `GET /` : Liste publique des offres avec statut `open`, incluant le nom du vendeur (`User.username`).
- `POST /` : Création d'une nouvelle offre (Nécessite `verifyToken`). Initialise `remaining` égal à `amount`.

Transactions (`/api/transaction` - *Supposé*)

- `GET /` : Historique personnel.
- `POST /` : Enregistrement d'un achat/vente (mise à jour potentielle du solde ou des offres).

2.4 Sécurité & Middlewares

- **Protection CSRF/XSS** : Utilisation de cookies `httpOnly` et `SameSite: Lax` pour le stockage du JWT. Empêche l'accès au token via JavaScript côté client.
- **Auth Middleware (`middleware/auth.js`)** :

1. Extrait le token du cookie.

2. Vérifie la signature via `process.env.JWT_SECRET`.
3. Attache l'objet décodé à `req.user`.
4. Bloque la requête si invalide (401 Unauthorized).

---

3. Architecture Frontend

3.1 Stack Technique

- **Core** : React 19 + Vite.
- **Langage** : JavaScript (ES Modules).
- **Style** : Tailwind CSS v3 (Utility-first).
- **State Management** : Local state (`useState`) + Prop drilling. L'état utilisateur (`user`) est géré dans `App.jsx` et passé aux enfants.

3.2 Structure de l'Application (`App.jsx`)

L'application est une SPA gérée par `react-router-dom`.

- **Chargement initial** : Au montage, un `useEffect` appelle `/api/user/me` pour vérifier la session persistante (cookie).

Routage & Protection

- **Routes Publiques** :
- `/` (Home)
- `/login`, `/register` (Redirigent vers `/` si déjà connecté)
- **Routes Protégées (User)** :
- `/profile` : Gestion du compte.
- `/transactions` : Historique.
- `/trade` : Interface de trading.
- *Mécanisme* : Si `user` est null, redirection vers `/login`.
- **Routes Protégées (Admin)** :
- `/dashboard` : Panneau de gestion global.
- *Mécanisme* : Vérifie `user.role === 'admin'`.

3.3 Intégration API (`api/axios.js`)

- Instance Axios configurée avec `baseURL: "/api"`.
- `withCredentials: true` : Indispensable pour l'envoi et la réception automatique des cookies de session cross-origin (si applicable) ou même domaine.

3.4 Internationalisation (i18n)

- Utilise `react-i18next`.
- Détection de langue navigateur + option utilisateur.
- Support RTL (Right-to-Left) dynamique pour l'arabe (`document.body.dir` mis à jour dans `App.jsx`).

---

4. Flux de Données & Interactions

Flow d'Authentification

1. Client envoie `email/password` à `/api/auth/login`.
2. Serveur valide, génère JWT, set cookie `token`.

3. Client reçoit "OK", mais ne manipule pas le token directement.
4. Client recharge l'état utilisateur via `/api/user/me`.
5. `App.jsx` met à jour l'état `user` et débloque les routes protégées.

Flow de création d'Offre

1. Utilisateur remplit le formulaire sur `/trade` (Coin, Montant, Prix).
2. POST vers `/api/offers`.
3. Backend crée l'entrée en BDD avec `sellerId = req.user.id`.
4. L'offre apparaît immédiatement dans le listing public.

---

5. Recommandations & Améliorations Futures

1. **Validation des Données** : Ajouter une validation stricte (ex: Joi ou Zod) sur les entrées API pour éviter les injections ou données corrompues.
2. **Gestion d'État Globale** : Remplacer le prop drilling de `user` par un Context React (`AuthContext`) pour simplifier `App.jsx`.
3. **Gestion des Erreurs** : Mettre en place un système global de notifications (Toasts) pour les erreurs API, actuellement logguées uniquement en console.
4. **Sécurité Production** : Passer `secure: true` pour les cookies en production (HTTPS requis).
   Ce rapport couvre l'intégralité de l'état actuel du développement du projet CryptoTracker.