**Logiciel de Gestion de Stock & Caisse (Web + Desktop)**

**1. Présentation Générale**

Développer une application complète de **gestion de stock** avec **caisse enregistreuse intégrée**, fonctionnant à la fois en mode **web (hébergé sur serveur)** et en mode **desktop (offline sur PC)** avec un **code unique**.

**Objectifs principaux**

- **Web** : Accès en ligne pour la gestion centralisée (administrateurs).
- **Desktop** : Application Windows autonome pour les points de vente, fonctionnant même sans connexion internet.
- **Synchronisation automatique** dès que la connexion est rétablie.
- **Deux rôles utilisateurs** : Admin (gestion complète) et Vendeur (utilisation caisse + consultation stock).
- **Technologies imposées** : Next.js, Prisma, PostgreSQL (serveur), PGlite (local), Electron, Drizzle ORM (optionnel), NextAuth.js.

---

**2. Rôles Utilisateurs et Permissions**

**Matrice des permissions**

| Fonctionnalité | Admin | Vendeur |
|---|---|---|
| Connexion | ✅ | ✅ |
| Tableau de bord (vue synthétique) | ✅ | ✅ (limitée) |
| **Gestion des produits** | | |
| - Consultation liste | ✅ | ✅ |
| - Ajout / Modification / Suppression | ✅ | ❌ |
| - Import/Export CSV | ✅ | ❌ |

| Fonctionnalité | Admin | Vendeur |
|---|---|---|
| **Gestion des stocks** | | |
| - Entrées (achats, retours) | ✅ | ❌ |
| - Sorties manuelles (casse, perte) | ✅ | ❌ |
| - Alertes stock | ✅ | ✅ (lecture) |
| **Caisse** | | |
| - Effectuer une vente | ✅ | ✅ |
| - Remboursement / Annulation | ✅ | ✅ (si autorisé) |
| - Impression ticket | ✅ | ✅ |
| - Historique des ventes | ✅ | ✅ (ses propres) |
| **Rapports** | ✅ | ❌ |
| **Gestion des utilisateurs** | ✅ | ❌ |
| **Paramètres (configuration)** | ✅ | ❌ |

---

## 3. Modules Fonctionnels

### 3.1 Module Authentification

- Page de connexion unique (email + mot de passe).
- Utilisation de NextAuth.js (Credentials provider) avec JWT.
- Redirection automatique selon le rôle après login.

- En mode desktop : authentification locale possible (hors-ligne) avec vérification périodique en ligne.

**3.2 Module Gestion de Stock (Admin uniquement)**

**Tableau de bord stock**

- **Indicateurs** : Nombre total de produits, valeur du stock (prix d'achat et vente), nombre de produits en rupture, en dessous du seuil.
- **Graphiques** : Évolution des entrées/sorties sur 30 jours, rotation des stocks.
- **Alertes** : Liste des produits critiques avec seuil dépassé.

**Gestion des produits (CRUD)**

- **Champs** :
    - id (UUID)
    - codeBarre (string, unique, nullable)
    - reference (string, unique)
    - nom (string)
    - description (text)
    - categorieId (foreign key vers Category)
    - prixAchat (decimal)
    - prixVente (decimal)
    - tva (decimal, ex: 20.0)
    - seuilAlerte (integer)
    - stockActuel (integer, calculé dynamiquement)
    - imageUrl (string, optionnelle)
    - variations (JSON, pour tailles/couleurs)
    - actif (boolean)
- **Fonctionnalités** :
    - Ajout / modification / suppression (soft delete possible)
    - Duplication d'un produit
    - Import/Export CSV/Excel
    - Gestion des catégories (arborescente)

**Mouvements de stock**

- **Entrées** :
    - Formulaire d'ajout : sélection produit, quantité, prix d'achat (optionnel), fournisseur, commentaire.
    - Import de commande fournisseur (fichier CSV avec code produit et quantité).
    - Historique des entrées avec date, utilisateur, fournisseur.
- **Sorties** :
    - Sorties pour vente (automatiques via caisse)
    - Sorties manuelles : motif (casse, perte, don, inventaire), quantité, commentaire.
    - Historique des sorties.
- **Journal des mouvements** : Tableau listant tous les mouvements (entrées/sorties) avec filtres (type, date, produit, utilisateur).

**3.3 Module Caisse (Vendeur + Admin)**

**Interface de vente**

- **Recherche produit** :
    - Scan de code-barres (via appareil photo ou scanner USB)
    - Recherche textuelle (nom, référence) avec autocomplétion
    - Affichage du stock disponible en temps réel
- **Panier** :
    - Liste des articles ajoutés avec quantité modifiable, prix unitaire, total ligne
    - Suppression d'article
    - Application de remises (pourcentage ou montant fixe) par article ou sur total
    - Sous-total, TVA par taux, total TTC
- **Validation** :
    - Sélection du mode de paiement (espèces, carte, chèque, virement)
    - Paiement mixte possible (ex: 20€ espèces + 30€ carte)
    - Calcul de la monnaie à rendre (pour espèces)

- Confirmation de vente → déduction immédiate du stock

**Gestion des tickets**

- Génération automatique d'un numéro de ticket unique (format : YYYYMMDD-XXXX)
- Impression sur imprimante thermique (via plugin Electron)
- Envoi par email au client (optionnel)
- Recherche d'un ticket par numéro ou par date

**Remboursements**

- Recherche du ticket original
- Sélection des articles à rembourser
- Calcul du montant remboursé
- Mise à jour du stock (réintégration)
- Enregistrement de la transaction de remboursement (lien avec le ticket original)

**3.4 Module Rapports (Admin)**

- **Chiffre d'affaires** : par jour, semaine, mois, année (graphique et tableau)
- **Top produits** : les plus vendus (quantité et montant)
- **Marge** : par produit, catégorie, global (calculée à partir du prix d'achat)
- **Rotation des stocks** : vitesse d'écoulement
- **Inventaire** : comparaison stock théorique vs physique après comptage
- **Export** : PDF, Excel, CSV

**3.5 Module Administration (Admin)**

- **Gestion des utilisateurs** : ajout, modification, suppression, changement de rôle.
- **Gestion des fournisseurs** : nom, contact, adresse, historique des commandes.
- **Paramètres généraux** : TVA par défaut, seuil d'alerte global, devise, informations légales (pour tickets).
- **Logs d'activité** : consultation des actions des utilisateurs.

---

**4. Architecture Technique Détaillée**

**4.1 Structure du projet (monorepo)**

text

```
mon-app/
├── packages/
│   ├── web/              # Application Next.js (pages, composants)
│   │   ├── pages/
│   │   ├── components/
│   │   ├── lib/          # Utilitaires, hooks
│   │   ├── styles/
│   │   └── public/
│   ├── desktop/          # Électron (main.js, preload.js)
│   │   ├── main.js
│   │   ├── preload.js
│   │   └── build/        # Fichiers de build (next export)
│   ├── shared/           # Code partagé
│   │   ├── types/        # Types TypeScript communs
│   │   ├── utils/
│   │   ├── db-abstract.ts    # Interface de base de données
│   │   └── sync/
│   └── sync-api/         # API de synchronisation (intégrée dans web)
├── prisma/               # Schéma Prisma (base serveur)
│   └── schema.prisma
├── drizzle/              # Schémas Drizzle (pour PGlite)
│   └── schema.ts
├── electron-builder.json     # Config build desktop
├── turbo.json            # (optionnel) pour monorepo
└── package.json
```

**4.2 Stack technologique**

| Environnement | Technologie |
| --- | --- |
| **Frontend (web & desktop)** | Next.js (React, TypeScript) |
| **Backend API (web)** | Next.js API routes |
| **Base de données serveur** | PostgreSQL + Prisma ORM |
| **Base de données locale (desktop)** | PGlite (PostgreSQL embarqué WASM) + Drizzle ORM |
| **Authentification** | NextAuth.js (Credentials) + JWT |
| **Desktop wrapper** | Electron |
| **UI Components** | Tailwind CSS + Headless UI / Radix UI |
| **State management** | Zustand ou Context API |
| **Synchronisation** | API REST + file d'attente locale |
| **Tests** | Jest, React Testing Library, Playwright |
| **Build & déploiement** | Vercel (web), electron-builder (desktop) |

**4.3 Base de données : Schéma détaillé**

**Tables serveur (PostgreSQL) – via Prisma**

prisma

```
model User {
 id          String    @id @default(uuid())
 email       String    @unique
 password    String    // hashé
 name        String?
 role        Role      @default(VENDEUR)
```

```
  storeId      String?   // pour multi-magasins futur
  createdAt    DateTime  @default(now())
  updatedAt    DateTime  @updatedAt
  sales        Sale[]
  movements    StockMovement[]
}

model Role {
  id   String @id @default(uuid())
  name  String @unique // "ADMIN", "VENDEUR"
  users User[]
}

model Category {
  id        String   @id @default(uuid())
  name      String   @unique
  description String?
  parentId   String?  @map("parent_id")
  parent     Category? @relation("CategoryToCategory", fields: [parentId], references: [id])
  children   Category[] @relation("CategoryToCategory")
  products   Product[]
  createdAt  DateTime  @default(now())
  updatedAt  DateTime  @updatedAt
}

model Product {
  id         String      @id @default(uuid())
  codeBarre    String?     @unique
```

```
  reference    String      @unique
  nom          String
  description  String?
  categorieId  String
  categorie    Category    @relation(fields: [categorieId], references: [id])
  prixAchat    Decimal     @map("prix_achat")
  prixVente    Decimal     @map("prix_vente")
  tva          Decimal     @default(20.0)
  seuilAlerte  Int         @default(5) @map("seuil_alerte")
  imageUrl     String?     @map("image_url")
  variations   Json?       // pour tailles/couleurs
  actif        Boolean     @default(true)
  createdAt    DateTime    @default(now())
  updatedAt    DateTime    @updatedAt
  stockMovements StockMovement[]
  saleItems    SaleItem[]
}

model StockMovement {
  id         String       @id @default(uuid())
  type       MovementType   // ENTRY, EXIT_SALE, EXIT_MANUAL
  quantity   Int
  productId  String
  product    Product      @relation(fields: [productId], references: [id])
  userId     String?
  user       User?        @relation(fields: [userId], references: [id])
  saleId     String?      // si lié à une vente
  sale       Sale?        @relation(fields: [saleId], references: [id])
```

```
  motif      String?       // pour sorties manuelles
  fournisseur String?      // pour entrées
  createdAt   DateTime      @default(now())
}

model Sale {
  id        String     @id @default(uuid())
  numero     String     @unique // format: YYYYMMDD-XXXX
  date       DateTime   @default(now())
  userId     String
  user       User       @relation(fields: [userId], references: [id])
  items      SaleItem[]
  payments   Payment[]
  totalHT    Decimal
  totalTVA   Decimal
  totalTTC   Decimal
  remise     Decimal?   @default(0)
  statut     SaleStatus  @default(COMPLETED) // COMPLETED, REFUNDED, CANCELLED
  refundOfId  String?    // si c'est un remboursement, lien vers la vente originale
  refundOf   Sale?      @relation("Refund", fields: [refundOfId], references: [id])
  refunds    Sale[]     @relation("Refund")
  createdAt   DateTime   @default(now())
}

model SaleItem {
  id        String   @id @default(uuid())
  saleId     String
  sale       Sale     @relation(fields: [saleId], references: [id])
```

```
  productId   String
  product    Product  @relation(fields: [productId], references: [id])
  quantity   Int
  prixUnitaire Decimal // au moment de la vente
  tva       Decimal
  remise     Decimal? @default(0)
}

model Payment {
  id        String     @id @default(uuid())
  saleId     String
  sale      Sale      @relation(fields: [saleId], references: [id])
  mode      PaymentMode // CASH, CARD, CHECK, TRANSFER
  montant    Decimal
}

enum MovementType {
  ENTRY
  EXIT_SALE
  EXIT_MANUAL
}

enum SaleStatus {
  COMPLETED
  REFUNDED
  CANCELLED
}
```

```
enum PaymentMode {
  CASH
  CARD
  CHECK
  TRANSFER
}
```

**Tables locales (PGlite) – via Drizzle ORM**

Les tables locales sont similaires, mais simplifiées et avec des champs supplémentaires pour la synchronisation.

- **Product** : identique mais sans les relations complexes (juste des IDs).
- **Category** : id, name.
- **Sale** : ajout de synced (boolean), syncError (text), lastSyncAttempt (datetime).
- **SaleItem** : id, saleId, productId, quantity, price, tva.
- **Payment** : id, saleId, mode, amount.
- **StockMovement** : id, type, productId, quantity, saleId (nullable), synced.
- **SyncQueue** : table dédiée pour les actions à synchroniser.

```
ts
// drizzle/schema.ts (exemple pour SyncQueue)
import { pgTable, uuid, text, timestamp, boolean, json } from "drizzle-orm/pg-core";

export const syncQueue = pgTable("sync_queue", {
  id: uuid("id").defaultRandom().primaryKey(),
  action: text("action").notNull(), // "CREATE_SALE", "UPDATE_PRODUCT", etc.
  data: json("data").notNull(),     // payload de l'action
  synced: boolean("synced").default(false),
  createdAt: timestamp("created_at").defaultNow(),
  updatedAt: timestamp("updated_at").defaultNow(),
});
```

### 4.4 API Endpoints (REST)

Tous les endpoints sont préfixés par /api/. Authentification via JWT (Bearer token).

**Authentification**

- POST /api/auth/login : { email, password } → { token, user }
- POST /api/auth/register (admin only)
- GET /api/auth/me : retourne l'utilisateur courant

**Produits**

- GET /api/products : liste paginée, filtres (catégorie, actif, recherche)
- GET /api/products/:id : détail
- POST /api/products : création (admin)
- PUT /api/products/:id : modification (admin)
- DELETE /api/products/:id : suppression (admin)
- POST /api/products/import : upload CSV

**Catégories**

- CRUD standard

**Mouvements de stock**

- GET /api/stock-movements : liste avec filtres
- POST /api/stock-movements/entry : ajout entrée (admin)
- POST /api/stock-movements/exit-manual : sortie manuelle (admin)

**Ventes**

- GET /api/sales : liste (admin toutes, vendeur les siennes)
- GET /api/sales/:id : détail
- POST /api/sales : création vente (accessible aux deux rôles)
- POST /api/sales/:id/refund : remboursement

**Rapports**

- GET /api/reports/turnover : params (from, to, groupBy)
- GET /api/reports/top-products : params (from, to, limit)
- GET /api/reports/margins : etc.

**Synchronisation (desktop)**

- POST /api/sync/push : reçoit un lot d'actions (ventes, mouvements) non synchronisées
    - Body : { actions: [{ action, data, localId }] }
    - Retour : { success: true, conflicts: [] } ou liste des conflits
- GET /api/sync/pull : retourne les mises à jour (produits modifiés, etc.) depuis un timestamp donné
    - Query : since (timestamp)
    - Retour : { products: [], categories: [], etc. }

---

**5. Synchronisation Offline/Online (Détaillée)**

**5.1 Principe général**

- L'application desktop fonctionne en **offline-first** : toutes les écritures (ventes, mouvements) sont d'abord enregistrées dans la base locale (PGlite) et ajoutées à la file syncQueue.
- Une tâche de fond (déclenchée par la détection de connexion) tente d'envoyer les actions à l'API /api/sync/push.
- Parallèlement, l'application interroge régulièrement /api/sync/pull pour récupérer les modifications depuis le serveur (produits, catégories, etc.) et met à jour la base locale.

**5.2 File d'attente (SyncQueue)**

- Chaque action est stockée avec :
    - id : UUID local
    - action : type d'action (ex: "CREATE_SALE", "UPDATE_PRODUCT")
    - data : JSON contenant les données nécessaires (ex: vente avec ses items)
    - synced : boolean
    - createdAt : date locale
    - updatedAt : date de dernière tentative
- En cas d'échec (réseau, erreur serveur), on incrémente un compteur de tentatives et on réessaie plus tard.

**5.3 Gestion des conflits**

Lors d'un push, le serveur peut détecter un conflit (ex: un produit modifié à la fois en local et sur le serveur depuis la dernière synchro). Stratégies :

- **Horodatage** : on compare les dates de modification. La plus récente l'emporte.
- **Serveur gagne** : en cas de conflit, la version serveur est considérée comme vérité.
- **Client gagne** : pour les ventes, c'est généralement le client qui a raison (car vente réelle). Pour les produits, l'admin est prioritaire.

Implémentation :

- Chaque enregistrement (produit, catégorie) a un champ updatedAt serveur.
- Lors du pull, le serveur renvoie tous les enregistrements modifiés depuis la dernière synchro.
- Lors du push, le serveur examine chaque action et retourne une liste de conflits. L'application desktop peut alors décider (ou demander à l'utilisateur) de résoudre.

**5.4 Processus de synchronisation typique**

1. **Desktop** :
    - Détecte une connexion internet.
    - Récupère les actions non synchronisées (synced = false) depuis SyncQueue.
    - Envoie un lot (max 50) à POST /api/sync/push.
2. **Serveur** :
    - Pour chaque action :
        - Si c'est une vente : crée la vente et les mouvements associés, vérifie la cohérence des stocks.
        - Si c'est une modification de produit : vérifie les conflits.
    - Retourne un tableau de résultats (succès ou conflit).
3. **Desktop** :
    - Pour les actions réussies, passe synced = true.
    - Pour les conflits, applique la stratégie choisie (ex: mise à jour locale depuis serveur, puis re-push si nécessaire).

4. **Pull** : après le push (ou indépendamment), le desktop appelle GET /api/sync/pull?since=... pour récupérer les nouvelles données (produits modifiés, nouvelles catégories, etc.) et met à jour sa base locale.

### 5.5 Initialisation du client desktop

- Première installation : l'application peut demander une synchronisation initiale (téléchargement de tous les produits, catégories, etc.) via un endpoint dédié /api/sync/initial.
- Les données de base sont stockées localement pour permettre le fonctionnement offline.

---

## 6. Interface Utilisateur (UI/UX) – Descriptions détaillées

### 6.1 Design System

- **Framework CSS** : Tailwind CSS
- **Composants** : Headless UI (menu, dialog, tabs) ou Radix UI
- **Icônes** : Heroicons ou Phosphor Icons
- **Thème** : clair/sombre automatique selon préférence système

### 6.2 Écrans principaux

#### Écran de connexion

- Formulaire email/mot de passe
- Lien "Mot de passe oublié" (envoi d'email de réinitialisation)
- Logo de l'entreprise

#### Dashboard Admin

- **En-tête** : date, nom du magasin, profil utilisateur
- **Cartes statistiques** : nb produits, valeur stock, nb alertes, CA du jour
- **Graphique** : évolution du CA sur 7/30 jours (Recharts)
- **Tableau des alertes** : produits sous seuil avec lien direct vers fiche produit
- **Derniers mouvements** : liste des 10 dernières entrées/sorties

#### Gestion des produits (Admin)

- **Liste** : tableau avec colonnes (image, nom, référence, stock, prix vente, actions)
    - Filtres : catégorie, actif, stock bas

    - Recherche globale
    - Pagination
- **Boutons** : Ajouter, Importer CSV, Exporter
- **Formulaire produit** : onglets (informations générales, variations, image)
    - Validation en temps réel
    - Upload d'image (drag & drop)

**Caisse (Vendeur)**

- **Disposition** : deux colonnes (gauche : recherche/panier, droite : touches rapides)
- **Recherche** : champ avec scanner intégré (focus automatique, déclenche recherche après scan)
- **Résultats** : affichage sous forme de grille de produits (image, nom, prix) avec ajout rapide
- **Panier** : liste déroulante des articles avec quantités +/- et suppression
- **Totaux** : sous-total, TVA, total
- **Paiement** : boutons pour chaque mode, popup de saisie montant, calcul rendu monnaie
- **Ticket** : après validation, affichage récapitulatif avec option impression/email

**Consultation stock (Vendeur)**

- Liste des produits avec stock actuel (lecture seule)
- Recherche et filtres (catégorie, nom)
- Possibilité de voir les mouvements récents (consultation uniquement)

**Rapports (Admin)**

- Sélecteur de période (calendrier)
- Graphiques interactifs
- Tableaux exportables

**Administration (Admin)**

- Gestion utilisateurs : tableau avec rôles, ajout/modif
- Gestion fournisseurs

- Paramètres : TVA, seuils, impression (format ticket)

### 6.3 Composants transversaux

- **Modale de confirmation** : pour actions critiques (suppression, annulation vente)
- **Notifications toast** : succès, erreur, avertissement
- **Skeleton loading** pendant les chargements
- **Gestion d'erreur** : messages explicites

---

## 7. Sécurité & Conformité

### 7.1 Authentification & Autorisation

- Mots de passe hashés avec bcrypt.
- JWT stocké en cookie HttpOnly (web) ou en mémoire (desktop).
- Middleware Next.js pour protéger les routes API et pages selon le rôle.
- Rate limiting sur les endpoints sensibles (login, sync).

### 7.2 Conformité légale française (NF525)

- **Journal des ventes inaltérable** : chaque vente est signée avec un hash calculé à partir des données de la vente et du hash précédent (chaîne de blocs légère).
- **Horodatage certifié** : utilisation d'un serveur NTP et stockage de l'heure de validation.
- **Données conservées 3 ans** : impossible de modifier ou supprimer une vente après validation (seulement annulation avec lien).
- **Export du journal** : fonctionnalité pour exporter le journal des ventes au format standard (pour contrôle fiscal).

Implémentation :

- Table Sale contient un champ previousHash et hash (calculé sur les données de la vente + previousHash).
- À chaque vente, on calcule le hash à partir des données (numéro, date, items, total) et du hash de la vente précédente.
- Le hash est stocké et vérifiable.

### 7.3 Sécurité des données

- Connexions HTTPS uniquement.

- Validation des entrées (Zod pour les schémas).
- Protection contre les injections SQL (via Prisma).
- Gestion sécurisée des fichiers uploadés (images).

---

**8. Développement Phases & Livrables**

**Phase 1 : MVP Web (2 mois)**

- Authentification (Admin/Vendeur)
- Gestion produits (CRUD Admin)
- Caisse simple (vente, 1 mode paiement, impression ticket basique)
- Stock : entrées/sorties manuelles
- Base de données PostgreSQL avec Prisma
- Déploiement sur Vercel

**Livrables** :

- Code source
- Documentation technique (installation, configuration)
- Site web fonctionnel

**Phase 2 : Version desktop offline (1.5 mois)**

- Intégration Electron avec Next.js (via next export)
- Base locale PGlite + Drizzle ORM
- Synchronisation unidirectionnelle (ventes → serveur)
- Gestion de la file d'attente
- Adaptation de l'UI pour desktop (taille de fenêtre, raccourcis)

**Livrables** :

- Application Windows (.exe)
- Documentation utilisateur desktop
- Scripts de build

**Phase 3 : Synchronisation bidirectionnelle (1.5 mois)**

- Pull des mises à jour produits depuis serveur

- Résolution des conflits
- Gestion avancée de la file d'attente (reprise sur erreur)
- Tests de synchronisation intensive

**Livrables** :

- Version stable avec synchro complète
- Rapports de test

**Phase 4 : Fonctionnalités avancées (2 mois)**

- Rapports détaillés
- Remises et codes promo
- Gestion des fournisseurs
- Conformité NF525 (journal inaltérable)
- Multi-devises (optionnel)

**Livrables** :

- Toutes fonctionnalités implémentées
- Tests de conformité

**Phase 5 : Tests & Déploiement final (1 mois)**

- Tests unitaires et d'intégration (couverture > 80%)
- Tests de charge sur API
- Déploiement web final (Vercel + serveur PostgreSQL)
- Build desktop final avec auto-updater
- Formation et documentation complète

**Livrables** :

- Suite de tests automatisés
- Binaires desktop signés
- Site web en production
- Manuel utilisateur (PDF)

---

**9. Contraintes Techniques Additionnelles**

- **Performance** : temps de réponse API < 200ms pour 95% des requêtes.
- **Taille base locale** : ne pas dépasser 500 Mo (prévoir purge automatique des anciennes données synchronisées).
- **Compatibilité navigateur** : Chrome, Firefox, Edge (versions récentes).
- **OS desktop** : Windows 10/11 (x64) ; possibilité d'étendre à macOS/Linux plus tard.
- **Mise à jour desktop** : via electron-builder avec auto-updater (fichiers sur serveur).

---

**10. Documentation Attendue**

1. **README** : présentation, technologies, installation rapide.
2. **Guide d'installation** : déploiement web, configuration de la base de données, variables d'environnement.
3. **Guide utilisateur** : avec captures d'écran, explication des fonctionnalités pour Admin et Vendeur.
4. **Documentation technique** : architecture, schéma BDD, endpoints API, flux de synchronisation.
5. **Guide de contribution** (si open source).

---

**11. Exigences Non Fonctionnelles**

- **Maintenabilité** : code commenté, respect des principes SOLID, tests.
- **Scalabilité** : l'API doit pouvoir gérer plusieurs centaines d'utilisateurs simultanés.
- **Sécurité** : audits réguliers, pas de faille XSS/CSRF.
- **Ergonomie** : interface intuitive, formation minimale requise.

---

Ce prompt exhaustif couvre tous les aspects fonctionnels, techniques et organisationnels nécessaires au développement de votre application de gestion de stock et caisse. Il peut être directement utilisé comme cahier des charges par une équipe de développement. Souhaitez-vous des précisions sur un point particulier ?

peux tu mettre tous ces details dans un fichier words?

We need to provide the user with a Word document containing all the details we've discussed. Since we are an AI, we cannot directly create a file, but we can guide them on how to create it. We can also offer to generate the content in a format that can be copied into Word. Perhaps we can provide the text formatted with headings and lists, and they can copy-paste.

We should ask if they want us to generate a downloadable link or just provide the text. However, we can't generate actual files, but we can output the content in a markdown or plain text with instructions to copy into Word.

Given the context, the user likely wants a single document summarizing the prompt. We can output the entire prompt in a structured way with headings, bullet points, and tables, which can be easily copied into Word. We can also mention that they can save it as .docx.

I'll provide a final answer that includes the entire prompt formatted for easy copy-pasting into Word, perhaps with a note that they can paste into Word and adjust formatting.

But we should also note that this is a long text, so we need to ensure it's well-structured. I'll reproduce the exhaustive prompt from the last response with slight formatting improvements for Word (using headings, tables, etc.). Since the last message already had the exhaustive prompt, I can reuse it. I'll just wrap it with instructions.

Let's craft the final answer.

Voici le contenu complet du prompt, formaté pour être facilement copié et collé dans un document Word. Il contient toutes les spécifications détaillées pour le développement de votre logiciel de gestion de stock avec caisse, intégrant les technologies Next.js, Prisma, PostgreSQL, PGlite et Electron.

Il ne vous reste qu'à copier le texte ci-dessous (y compris les titres et tableaux), le coller dans un nouveau document Word, puis ajuster la mise en forme selon vos préférences (police, marges, etc.).

---

text

# PROMPT EXHAUSTIF : LOGICIEL DE GESTION DE STOCK & CAISSE (WEB + DESKTOP)

## 1. Présentation Générale

Développer une application complète de **gestion de stock** avec **caisse enregistreuse intégrée**, fonctionnant à la fois en mode **web (hébergé sur serveur)** et en mode **desktop (offline sur PC)** avec un **code unique**.

### Objectifs principaux

- **Web** : Accès en ligne pour la gestion centralisée (administrateurs).

- **Desktop** : Application Windows autonome pour les points de vente, fonctionnant même sans connexion internet.

- **Synchronisation automatique** dès que la connexion est rétablie.

- **Deux rôles utilisateurs** : Admin (gestion complète) et Vendeur (utilisation caisse + consultation stock).

- **Technologies imposées** : Next.js, Prisma, PostgreSQL (serveur), PGlite (local), Electron, Drizzle ORM (optionnel), NextAuth.js.

## 2. Rôles Utilisateurs et Permissions

| Fonctionnalité | Admin | Vendeur |
|---|:---:|:---:|
| Connexion | ✅ | ✅ |
| Tableau de bord (vue synthétique) | ✅ | ✅ (limitée) |
| **Gestion des produits** | | |
| - Consultation liste | ✅ | ✅ |
| - Ajout / Modification / Suppression | ✅ | ❌ |
| - Import/Export CSV | ✅ | ❌ |
| **Gestion des stocks** | | |
| - Entrées (achats, retours) | ✅ | ❌ |
| - Sorties manuelles (casse, perte) | ✅ | ❌ |
| - Alertes stock | ✅ | ✅ (lecture) |
| **Caisse** | | |

| - Effectuer une vente | ✅ | ✅ |

| - Remboursement / Annulation | ✅ | ✅ (si autorisé) |

| - Impression ticket | ✅ | ✅ |

| - Historique des ventes | ✅ | ✅ (ses propres) |

| **Rapports** | ✅ | ❌ |

| **Gestion des utilisateurs** | ✅ | ❌ |

| **Paramètres (configuration)** | ✅ | ❌ |

## 3. Modules Fonctionnels

### 3.1 Module Authentification

- Page de connexion unique (email + mot de passe).

- Utilisation de NextAuth.js (Credentials provider) avec JWT.

- Redirection automatique selon le rôle après login.

- En mode desktop : authentification locale possible (hors-ligne) avec vérification périodique en ligne.

### 3.2 Module Gestion de Stock (Admin uniquement)

#### Tableau de bord stock

- **Indicateurs** : Nombre total de produits, valeur du stock (prix d'achat et vente), nombre de produits en rupture, en dessous du seuil.

- **Graphiques** : Évolution des entrées/sorties sur 30 jours, rotation des stocks.

- **Alertes** : Liste des produits critiques avec seuil dépassé.

#### Gestion des produits (CRUD)

- **Champs** :

  - `id` (UUID)

- `codeBarre` (string, unique, nullable)
- `reference` (string, unique)
- `nom` (string)
- `description` (text)
- `categorieId` (foreign key vers Category)
- `prixAchat` (decimal)
- `prixVente` (decimal)
- `tva` (decimal, ex: 20.0)
- `seuilAlerte` (integer)
- `stockActuel` (integer, calculé dynamiquement)
- `imageUrl` (string, optionnelle)
- `variations` (JSON, pour tailles/couleurs)
- `actif` (boolean)

- **Fonctionnalités** :
  - Ajout / modification / suppression (soft delete possible)
  - Duplication d'un produit
  - Import/Export CSV/Excel
  - Gestion des catégories (arborescente)

#### Mouvements de stock

- **Entrées** :
  - Formulaire d'ajout : sélection produit, quantité, prix d'achat (optionnel), fournisseur, commentaire.
  - Import de commande fournisseur (fichier CSV avec code produit et quantité).
  - Historique des entrées avec date, utilisateur, fournisseur.
- **Sorties** :
  - Sorties pour vente (automatiques via caisse)
  - Sorties manuelles : motif (casse, perte, don, inventaire), quantité, commentaire.

  - Historique des sorties.

- **Journal des mouvements** : Tableau listant tous les mouvements (entrées/sorties) avec filtres (type, date, produit, utilisateur).

### 3.3 Module Caisse (Vendeur + Admin)

#### Interface de vente

- **Recherche produit** :
  - Scan de code-barres (via appareil photo ou scanner USB)
  - Recherche textuelle (nom, référence) avec autocomplétion
  - Affichage du stock disponible en temps réel
- **Panier** :
  - Liste des articles ajoutés avec quantité modifiable, prix unitaire, total ligne
  - Suppression d'article
  - Application de remises (pourcentage ou montant fixe) par article ou sur total
  - Sous-total, TVA par taux, total TTC
- **Validation** :
  - Sélection du mode de paiement (espèces, carte, chèque, virement)
  - Paiement mixte possible (ex: 20€ espèces + 30€ carte)
  - Calcul de la monnaie à rendre (pour espèces)
  - Confirmation de vente → déduction immédiate du stock

#### Gestion des tickets

- Génération automatique d'un numéro de ticket unique (format : `YYYYMMDD-XXXX`)
- Impression sur imprimante thermique (via plugin Electron)
- Envoi par email au client (optionnel)
- Recherche d'un ticket par numéro ou par date

#### Remboursements

- Recherche du ticket original

- Sélection des articles à rembourser

- Calcul du montant remboursé

- Mise à jour du stock (réintégration)

- Enregistrement de la transaction de remboursement (lien avec le ticket original)

### 3.4 Module Rapports (Admin)

- **Chiffre d'affaires** : par jour, semaine, mois, année (graphique et tableau)

- **Top produits** : les plus vendus (quantité et montant)

- **Marge** : par produit, catégorie, global (calculée à partir du prix d'achat)

- **Rotation des stocks** : vitesse d'écoulement

- **Inventaire** : comparaison stock théorique vs physique après comptage

- **Export** : PDF, Excel, CSV

### 3.5 Module Administration (Admin)

- **Gestion des utilisateurs** : ajout, modification, suppression, changement de rôle.

- **Gestion des fournisseurs** : nom, contact, adresse, historique des commandes.

- **Paramètres généraux** : TVA par défaut, seuil d'alerte global, devise, informations légales (pour tickets).

- **Logs d'activité** : consultation des actions des utilisateurs.

## 4. Architecture Technique Détaillée

### 4.1 Structure du projet (monorepo)

```
mon-app/
├— packages/
```

```
│  ├── web/ # Application Next.js (pages, composants)
│  │  ├── pages/
│  │  ├── components/
│  │  ├── lib/ # Utilitaires, hooks
│  │  ├── styles/
│  │  └── public/
│  ├── desktop/ # Électron (main.js, preload.js)
│  │  ├── main.js
│  │  ├── preload.js
│  │  └── build/ # Fichiers de build (next export)
│  ├── shared/ # Code partagé
│  │  ├── types/ # Types TypeScript communs
│  │  ├── utils/
│  │  ├── db-abstract.ts # Interface de base de données
│  │  └── sync/
│  └── sync-api/ # API de synchronisation (intégrée dans web)
├── prisma/ # Schéma Prisma (base serveur)
│  └── schema.prisma
├── drizzle/ # Schémas Drizzle (pour PGlite)
│  └── schema.ts
├── electron-builder.json # Config build desktop
├── turbo.json # (optionnel) pour monorepo
└── package.json
```

text

### 4.2 Stack technologique

| Environnement | Technologie |
|---|---|
| **Frontend (web & desktop)** | Next.js (React, TypeScript) |
| **Backend API (web)** | Next.js API routes |
| **Base de données serveur** | PostgreSQL + Prisma ORM |
| **Base de données locale (desktop)** | PGlite (PostgreSQL embarqué WASM) + Drizzle ORM |
| **Authentification** | NextAuth.js (Credentials) + JWT |

| **Desktop wrapper** | Electron |

| **UI Components** | Tailwind CSS + Headless UI / Radix UI |

| **State management** | Zustand ou Context API |

| **Synchronisation** | API REST + file d'attente locale |

| **Tests** | Jest, React Testing Library, Playwright |

| **Build & déploiement** | Vercel (web), electron-builder (desktop) |

### 4.3 Base de données : Schéma détaillé

#### Tables serveur (PostgreSQL) – via Prisma

```prisma
model User {
 id          String    @id @default(uuid())
 email       String    @unique
 password    String    // hashé
 name        String?
 role        Role      @default(VENDEUR)
 storeId     String?   // pour multi-magasins futur
 createdAt   DateTime  @default(now())
 updatedAt   DateTime  @updatedAt
 sales       Sale[]
 movements   StockMovement[]
}

model Role {
 id   String @id @default(uuid())
 name  String @unique // "ADMIN", "VENDEUR"
```

```
  users User[]
}

model Category {
  id        String   @id @default(uuid())
  name      String   @unique
  description String?
  parentId   String?  @map("parent_id")
  parent    Category? @relation("CategoryToCategory", fields: [parentId], references: [id])
  children   Category[] @relation("CategoryToCategory")
  products   Product[]
  createdAt  DateTime  @default(now())
  updatedAt  DateTime  @updatedAt
}

model Product {
  id         String      @id @default(uuid())
  codeBarre   String?     @unique
  reference   String      @unique
  nom         String
  description  String?
  categorieId  String
  categorie   Category    @relation(fields: [categorieId], references: [id])
  prixAchat   Decimal     @map("prix_achat")
  prixVente   Decimal     @map("prix_vente")
  tva         Decimal     @default(20.0)
  seuilAlerte  Int        @default(5) @map("seuil_alerte")
  imageUrl    String?     @map("image_url")
```

```
  variations    Json?       // pour tailles/couleurs
  actif         Boolean     @default(true)
  createdAt     DateTime    @default(now())
  updatedAt     DateTime    @updatedAt
  stockMovements StockMovement[]
  saleItems     SaleItem[]
}

model StockMovement {
  id        String       @id @default(uuid())
  type      MovementType   // ENTRY, EXIT_SALE, EXIT_MANUAL
  quantity  Int
  productId String
  product   Product      @relation(fields: [productId], references: [id])
  userId    String?
  user      User?        @relation(fields: [userId], references: [id])
  saleId    String?      // si lié à une vente
  sale      Sale?        @relation(fields: [saleId], references: [id])
  motif     String?      // pour sorties manuelles
  fournisseur String?    // pour entrées
  createdAt DateTime     @default(now())
}

model Sale {
  id        String    @id @default(uuid())
  numero    String    @unique // format: YYYYMMDD-XXXX
  date      DateTime  @default(now())
  userId    String
```

```
  user       User       @relation(fields: [userId], references: [id])
  items      SaleItem[]
  payments   Payment[]
  totalHT    Decimal
  totalTVA   Decimal
  totalTTC   Decimal
  remise     Decimal?   @default(0)
  statut     SaleStatus  @default(COMPLETED) // COMPLETED, REFUNDED, CANCELLED
  refundOfId  String?    // si c'est un remboursement, lien vers la vente originale
  refundOf   Sale?      @relation("Refund", fields: [refundOfId], references: [id])
  refunds    Sale[]     @relation("Refund")
  createdAt  DateTime   @default(now())
}

model SaleItem {
  id        String   @id @default(uuid())
  saleId    String
  sale      Sale     @relation(fields: [saleId], references: [id])
  productId  String
  product   Product  @relation(fields: [productId], references: [id])
  quantity  Int
  prixUnitaire Decimal // au moment de la vente
  tva       Decimal
  remise    Decimal? @default(0)
}

model Payment {
  id        String    @id @default(uuid())
```

```
  saleId     String
  sale       Sale       @relation(fields: [saleId], references: [id])
  mode       PaymentMode // CASH, CARD, CHECK, TRANSFER
  montant    Decimal
}

enum MovementType {
  ENTRY
  EXIT_SALE
  EXIT_MANUAL
}

enum SaleStatus {
  COMPLETED
  REFUNDED
  CANCELLED
}

enum PaymentMode {
  CASH
  CARD
  CHECK
  TRANSFER
}
```

**Tables locales (PGlite) – via Drizzle ORM**

```
ts
// drizzle/schema.ts (exemple pour SyncQueue)
import { pgTable, uuid, text, timestamp, boolean, json } from "drizzle-orm/pg-core";
```

```
export const syncQueue = pgTable("sync_queue", {
  id: uuid("id").defaultRandom().primaryKey(),
  action: text("action").notNull(), // "CREATE_SALE", "UPDATE_PRODUCT", etc.
  data: json("data").notNull(),     // payload de l'action
  synced: boolean("synced").default(false),
  createdAt: timestamp("created_at").defaultNow(),
  updatedAt: timestamp("updated_at").defaultNow(),
});
```

**4.4 API Endpoints (REST)**

Tous les endpoints sont préfixés par /api/. Authentification via JWT (Bearer token).

**Authentification**

- POST /api/auth/login : { email, password } → { token, user }
- POST /api/auth/register (admin only)
- GET /api/auth/me : retourne l'utilisateur courant

**Produits**

- GET /api/products : liste paginée, filtres (catégorie, actif, recherche)
- GET /api/products/:id : détail
- POST /api/products : création (admin)
- PUT /api/products/:id : modification (admin)
- DELETE /api/products/:id : suppression (admin)
- POST /api/products/import : upload CSV

**Catégories**

- CRUD standard

**Mouvements de stock**

- GET /api/stock-movements : liste avec filtres
- POST /api/stock-movements/entry : ajout entrée (admin)
- POST /api/stock-movements/exit-manual : sortie manuelle (admin)

**Ventes**

- GET /api/sales : liste (admin toutes, vendeur les siennes)
- GET /api/sales/:id : détail
- POST /api/sales : création vente (accessible aux deux rôles)
- POST /api/sales/:id/refund : remboursement

**Rapports**

- GET /api/reports/turnover : params (from, to, groupBy)
- GET /api/reports/top-products : params (from, to, limit)
- GET /api/reports/margins : etc.

**Synchronisation (desktop)**

- POST /api/sync/push : reçoit un lot d'actions (ventes, mouvements) non synchronisées
    - Body : { actions: [{ action, data, localId }] }
    - Retour : { success: true, conflicts: [] } ou liste des conflits
- GET /api/sync/pull : retourne les mises à jour (produits modifiés, etc.) depuis un timestamp donné
    - Query : since (timestamp)
    - Retour : { products: [], categories: [], etc. }

**5. Synchronisation Offline/Online (Détaillée)**

**5.1 Principe général**

- L'application desktop fonctionne en **offline-first** : toutes les écritures (ventes, mouvements) sont d'abord enregistrées dans la base locale (PGlite) et ajoutées à la file syncQueue.
- Une tâche de fond (déclenchée par la détection de connexion) tente d'envoyer les actions à l'API /api/sync/push.
- Parallèlement, l'application interroge régulièrement /api/sync/pull pour récupérer les modifications depuis le serveur (produits, catégories, etc.) et met à jour la base locale.

**5.2 File d'attente (SyncQueue)**

- Chaque action est stockée avec :

    - id : UUID local
    - action : type d'action (ex: "CREATE_SALE", "UPDATE_PRODUCT")
    - data : JSON contenant les données nécessaires (ex: vente avec ses items)
    - synced : boolean
    - createdAt : date locale
    - updatedAt : date de dernière tentative
- En cas d'échec (réseau, erreur serveur), on incrémente un compteur de tentatives et on réessaie plus tard.

**5.3 Gestion des conflits**

Lors d'un push, le serveur peut détecter un conflit (ex: un produit modifié à la fois en local et sur le serveur depuis la dernière synchro). Stratégies :

- **Horodatage** : on compare les dates de modification. La plus récente l'emporte.
- **Serveur gagne** : en cas de conflit, la version serveur est considérée comme vérité.
- **Client gagne** : pour les ventes, c'est généralement le client qui a raison (car vente réelle). Pour les produits, l'admin est prioritaire.

Implémentation :

- Chaque enregistrement (produit, catégorie) a un champ updatedAt serveur.
- Lors du pull, le serveur renvoie tous les enregistrements modifiés depuis la dernière synchro.
- Lors du push, le serveur examine chaque action et retourne une liste de conflits. L'application desktop peut alors décider (ou demander à l'utilisateur) de résoudre.

**5.4 Processus de synchronisation typique**

1. **Desktop** :
    - Détecte une connexion internet.
    - Récupère les actions non synchronisées (synced = false) depuis SyncQueue.
    - Envoie un lot (max 50) à POST /api/sync/push.
2. **Serveur** :
    - Pour chaque action :

        - Si c'est une vente : crée la vente et les mouvements associés, vérifie la cohérence des stocks.
        - Si c'est une modification de produit : vérifie les conflits.
    - Retourne un tableau de résultats (succès ou conflit).
3. **Desktop** :
    - Pour les actions réussies, passe synced = true.
    - Pour les conflits, applique la stratégie choisie (ex: mise à jour locale depuis serveur, puis re-push si nécessaire).
4. **Pull** : après le push (ou indépendamment), le desktop appelle GET /api/sync/pull?since=... pour récupérer les nouvelles données (produits modifiés, nouvelles catégories, etc.) et met à jour sa base locale.

**5.5 Initialisation du client desktop**

- Première installation : l'application peut demander une synchronisation initiale (téléchargement de tous les produits, catégories, etc.) via un endpoint dédié /api/sync/initial.
- Les données de base sont stockées localement pour permettre le fonctionnement offline.

**6. Interface Utilisateur (UI/UX) – Descriptions détaillées**

**6.1 Design System**

- **Framework CSS** : Tailwind CSS
- **Composants** : Headless UI (menu, dialog, tabs) ou Radix UI
- **Icônes** : Heroicons ou Phosphor Icons
- **Thème** : clair/sombre automatique selon préférence système

**6.2 Écrans principaux**

**Écran de connexion**

- Formulaire email/mot de passe
- Lien "Mot de passe oublié" (envoi d'email de réinitialisation)
- Logo de l'entreprise

**Dashboard Admin**

- **En-tête** : date, nom du magasin, profil utilisateur

- **Cartes statistiques** : nb produits, valeur stock, nb alertes, CA du jour
- **Graphique** : évolution du CA sur 7/30 jours (Recharts)
- **Tableau des alertes** : produits sous seuil avec lien direct vers fiche produit
- **Derniers mouvements** : liste des 10 dernières entrées/sorties

**Gestion des produits (Admin)**

- **Liste** : tableau avec colonnes (image, nom, référence, stock, prix vente, actions)
    - Filtres : catégorie, actif, stock bas
    - Recherche globale
    - Pagination
- **Boutons** : Ajouter, Importer CSV, Exporter
- **Formulaire produit** : onglets (informations générales, variations, image)
    - Validation en temps réel
    - Upload d'image (drag & drop)

**Caisse (Vendeur)**

- **Disposition** : deux colonnes (gauche : recherche/panier, droite : touches rapides)
- **Recherche** : champ avec scanner intégré (focus automatique, déclenche recherche après scan)
- **Résultats** : affichage sous forme de grille de produits (image, nom, prix) avec ajout rapide
- **Panier** : liste déroulante des articles avec quantités +/- et suppression
- **Totaux** : sous-total, TVA, total
- **Paiement** : boutons pour chaque mode, popup de saisie montant, calcul rendu monnaie
- **Ticket** : après validation, affichage récapitulatif avec option impression/email

**Consultation stock (Vendeur)**

- Liste des produits avec stock actuel (lecture seule)
- Recherche et filtres (catégorie, nom)
- Possibilité de voir les mouvements récents (consultation uniquement)

**Rapports (Admin)**

- Sélecteur de période (calendrier)
- Graphiques interactifs
- Tableaux exportables

**Administration (Admin)**

- Gestion utilisateurs : tableau avec rôles, ajout/modif
- Gestion fournisseurs
- Paramètres : TVA, seuils, impression (format ticket)

**6.3 Composants transversaux**

- **Modale de confirmation** : pour actions critiques (suppression, annulation vente)
- **Notifications toast** : succès, erreur, avertissement
- **Skeleton loading** pendant les chargements
- **Gestion d'erreur** : messages explicites

**7. Sécurité & Conformité**

**7.1 Authentification & Autorisation**

- Mots de passe hashés avec bcrypt.
- JWT stocké en cookie HttpOnly (web) ou en mémoire (desktop).
- Middleware Next.js pour protéger les routes API et pages selon le rôle.
- Rate limiting sur les endpoints sensibles (login, sync).

**7.2 Conformité légale française (NF525)**

- **Journal des ventes inaltérable** : chaque vente est signée avec un hash calculé à partir des données de la vente et du hash précédent (chaîne de blocs légère).
- **Horodatage certifié** : utilisation d'un serveur NTP et stockage de l'heure de validation.
- **Données conservées 3 ans** : impossible de modifier ou supprimer une vente après validation (seulement annulation avec lien).
- **Export du journal** : fonctionnalité pour exporter le journal des ventes au format standard (pour contrôle fiscal).

Implémentation :

- Table Sale contient un champ previousHash et hash (calculé sur les données de la vente + previousHash).
- À chaque vente, on calcule le hash à partir des données (numéro, date, items, total) et du hash de la vente précédente.
- Le hash est stocké et vérifiable.

**7.3 Sécurité des données**

- Connexions HTTPS uniquement.
- Validation des entrées (Zod pour les schémas).
- Protection contre les injections SQL (via Prisma).
- Gestion sécurisée des fichiers uploadés (images).

**8. Développement Phases & Livrables**

**Phase 1 : MVP Web (2 mois)**

- Authentification (Admin/Vendeur)
- Gestion produits (CRUD Admin)
- Caisse simple (vente, 1 mode paiement, impression ticket basique)
- Stock : entrées/sorties manuelles
- Base de données PostgreSQL avec Prisma
- Déploiement sur Vercel

**Livrables** :

- Code source
- Documentation technique (installation, configuration)
- Site web fonctionnel

**Phase 2 : Version desktop offline (1.5 mois)**

- Intégration Electron avec Next.js (via next export)
- Base locale PGlite + Drizzle ORM
- Synchronisation unidirectionnelle (ventes → serveur)
- Gestion de la file d'attente
- Adaptation de l'UI pour desktop (taille de fenêtre, raccourcis)

**Livrables** :

- Application Windows (.exe)
- Documentation utilisateur desktop
- Scripts de build

**Phase 3 : Synchronisation bidirectionnelle (1.5 mois)**

- Pull des mises à jour produits depuis serveur
- Résolution des conflits
- Gestion avancée de la file d'attente (reprise sur erreur)
- Tests de synchronisation intensive

**Livrables** :

- Version stable avec synchro complète
- Rapports de test

**Phase 4 : Fonctionnalités avancées (2 mois)**

- Rapports détaillés
- Remises et codes promo
- Gestion des fournisseurs
- Conformité NF525 (journal inaltérable)
- Multi-devises (optionnel)

**Livrables** :

- Toutes fonctionnalités implémentées
- Tests de conformité

**Phase 5 : Tests & Déploiement final (1 mois)**

- Tests unitaires et d'intégration (couverture > 80%)
- Tests de charge sur API
- Déploiement web final (Vercel + serveur PostgreSQL)
- Build desktop final avec auto-updater
- Formation et documentation complète

**Livrables** :

- Suite de tests automatisés

- Binaires desktop signés
- Site web en production
- Manuel utilisateur (PDF)

**9. Contraintes Techniques Additionnelles**

- **Performance** : temps de réponse API < 200ms pour 95% des requêtes.
- **Taille base locale** : ne pas dépasser 500 Mo (prévoir purge automatique des anciennes données synchronisées).
- **Compatibilité navigateur** : Chrome, Firefox, Edge (versions récentes).
- **OS desktop** : Windows 10/11 (x64) ; possibilité d'étendre à macOS/Linux plus tard.
- **Mise à jour desktop** : via electron-builder avec auto-updater (fichiers sur serveur).

**10. Documentation Attendue**

1. **README** : présentation, technologies, installation rapide.
2. **Guide d'installation** : déploiement web, configuration de la base de données, variables d'environnement.
3. **Guide utilisateur** : avec captures d'écran, explication des fonctionnalités pour Admin et Vendeur.
4. **Documentation technique** : architecture, schéma BDD, endpoints API, flux de synchronisation.
5. **Guide de contribution** (si open source).

**11. Exigences Non Fonctionnelles**

- **Maintenabilité** : code commenté, respect des principes SOLID, tests.
- **Scalabilité** : l'API doit pouvoir gérer plusieurs centaines d'utilisateurs simultanés.
- **Sécurité** : audits réguliers, pas de faille XSS/CSRF.
- **Ergonomie** : interface intuitive, formation minimale requise.