H1Watch — Backend API

> API REST e-commerce pour la vente de montres de luxe. Conçue selon les principes **Service Layer**, **Repository Pattern** et **Clean Architecture** avec Node.js, PostgreSQL et Redis.

Table des Matières

Architecture

L'application suit une architecture en couches strictement découplées :

```
HTTP Request
    │
    ▼
┌──────────────────┐
│   Middlewares    │  ← Helmet, CORS, Rate Limit, Auth, Sanitizer
└──────────────────┘
        │
        ▼
┌──────────────────┐
│   Controllers    │  ← Extraction params HTTP, délégation au Service
└──────────────────┘
        │
        ▼
┌──────────────────┐
│    Services      │  ← Logique métier, orchestration, règles business
└──────────────────┘
        │
        ▼
┌──────────────────┐
│  Repositories    │  ← Accès données, requêtes SQL, mapping camelCase
└──────────────────┘
        │
```

```
      ├──────────────── PostgreSQL (données persistantes)
      └──────────────── Redis      (sessions, cache)
```

Principes Appliqués

Service Layer Pattern — toute la logique métier vit dans les services. Les contrôleurs ne font qu'extraire les paramètres de la requête HTTP et déléguer. Cela rend les services testables indépendamment d'Express.

Repository Pattern — chaque table a son propre repository qui abstrait les requêtes SQL. Le fichier `_mappers.js` centralise la transformation `snake_case` → `camelCase` pour toute la couche de données.

Singleton Pattern — les services critiques (AuthService, CartService, CacheService, etc.) sont des singletons pour éviter les connexions multiples à Redis et garantir la cohérence de l'état.

Prérequis

- **Node.js** ≥ 20 (modules ESM natifs)
- **Docker** & **Docker Compose** v2+
- **npm** ≥ 10

Installation & Démarrage

Démarrage Rapide (Recommandé — Docker)

```
# 1. Cloner le dépôt
git clone <repo-url> && cd h1watch-backend

# 2. Copier et remplir les variables d'environnement
cp .env.example .env
# Éditez .env avec vos valeurs

# 3. Démarrer tous les services (PostgreSQL + Redis + API)
docker compose up -d

# 4. Vérifier l'état de santé
curl http://localhost:3001/health
```

L'API sera disponible sur **http://localhost:3001**. La base de données est initialisée automatiquement par `init-postgres.sql` au premier démarrage du container PostgreSQL.

Développement Local (sans Docker API)

Si vous préférez exécuter l'API directement sur votre machine (avec Nodemon pour le hot-reload) tout en gardant PostgreSQL et Redis dans Docker :

```
# 1. Démarrer uniquement les services d'infrastructure
docker compose up -d db redis

# 2. Installer les dépendances
npm install

# 3. Démarrer en mode développement avec Nodemon
npm run dev
```

> **Note :** En mode `dev`, Sentry est initialisé via `--import ./src/config/instruments.js`. Assurez-vous que votre `SENTRY_DSN` est valide ou commentez la ligne dans `package.json`.

Variables d'Environnement

Créez un fichier `.env` à la racine du projet. Toutes les variables ci-dessous sont **obligatoires** sauf mention contraire.

```
# ─── Serveur ──────────────────────────────────────────────────────────
PORT=3001
NODE_ENV=development        # development | production | test

# ─── PostgreSQL ───────────────────────────────────────────────────────
POSTGRES_HOST=localhost      # 'db' si vous utilisez Docker Compose
POSTGRES_PORT=5432
POSTGRES_USER=h1watch
POSTGRES_PASSWORD=votre_mot_de_passe_fort
POSTGRES_DB=h1watch_db

# ─── Redis ────────────────────────────────────────────────────────────
REDIS_HOST=localhost         # 'redis' si vous utilisez Docker Compose
REDIS_PORT=6379
REDIS_PASSWORD=              # Optionnel, laisser vide si pas de mot de passe

# ─── JWT ──────────────────────────────────────────────────────────────
JWT_ACCESS_SECRET=votre_secret_access_tres_long_et_aleatoire
JWT_REFRESH_SECRET=votre_secret_refresh_different_du_precedent
JWT_ACCESS_EXPIRY=15m        # Optionnel, défaut: 15m
JWT_REFRESH_EXPIRY=7d        # Optionnel, défaut: 7d

# ─── Sécurité ─────────────────────────────────────────────────────────
BCRYPT_ITERATIONS=100000     # Optionnel, défaut: 100000 (norme OWASP)
RATE_LIMIT_WINDOW_MS=900000  # Optionnel, défaut: 15 min
RATE_LIMIT_MAX=100           # Optionnel, défaut: 100 req/fenêtre

# ─── Stripe ───────────────────────────────────────────────────────────
STRIPE_SECRET_KEY=sk_test_...
STRIPE_WEBHOOK_SECRET=whsec_...

# ─── Sentry ───────────────────────────────────────────────────────────
```

```
SENTRY_DSN=https://...@sentry.io/...

# ─── Email (Optionnel) ──────────────────────────────────────────────
# MAIL_HOST=smtp.example.com
# MAIL_PORT=587
# MAIL_USER=noreply@example.com
# MAIL_PASS=votre_mot_de_passe_smtp

# ─── CORS ───────────────────────────────────────────────────────────
CLIENT_URL=http://localhost:5173
CORS_ORIGINS=http://localhost:5173,http://localhost:3000
```

Endpoints de l'API

Tous les endpoints sont préfixés par `/api/v1`.

Authentification — `/auth`

Méthode	Endpoint	Auth	Description
POST	/auth/register	—	Créer un compte (retourne access token + cookie refresh)
POST	/auth/login	—	Connexion (retourne access token + cookie refresh)
POST	/auth/logout	—	Déconnexion (révoque la session)
POST	/auth/refresh	Cookie	Renouveler l'access token

```
# Exemple : Inscription
POST /api/v1/auth/register
Content-Type: application/json

{
  "email": "user@example.com",
  "password": "Password123",
  "firstName": "Jean",
  "lastName": "Dupont"
}
```

Produits — `/products`

Méthode	Endpoint	Auth	Description
GET	/products	—	Catalogue (filtres: status, categorySlug, page, limit)
GET	/products/:idOrSlug	—	Détail produit (UUID ou slug SEO)
POST	/products	Admin	Créer un produit avec sa première variante
PATCH	/products/:id	Admin	Mettre à jour un produit

Méthode	Endpoint	Auth	Description
DELETE	/products/:id	Admin	Supprimer un produit (cascade variantes + stock)

Panier — /cart

Méthode	Endpoint	Auth	Description
GET	/cart	☑	Panier complet avec totaux
POST	/cart/items	☑	Ajouter un article
PATCH	/cart/items/:itemId	☑	Mettre à jour la quantité
DELETE	/cart/items/:itemId	☑	Retirer un article
DELETE	/cart	☑	Vider le panier

Commandes — /orders

Méthode	Endpoint	Auth	Description
POST	/orders/checkout	☑	Transformer le panier en commande
GET	/orders/my-orders	☑	Historique des commandes
GET	/orders/:orderId	☑	Détail d'une commande
GET	/orders	Admin	Toutes les commandes (paginées)
PATCH	/orders/:orderId/status	Admin	Changer le statut

Paiements — /payments

Méthode	Endpoint	Auth	Description
POST	/payments/create-session/:orderId	☑	Créer une session Stripe Checkout
GET	/payments/status/:orderId	☑	Statut du paiement
POST	/payments/webhook/stripe	—	Webhook Stripe (signature requise)

Autres Modules

```
/users          → Profil, gestion des mots de passe (Admin: liste, suppression)
/categories     → CRUD catégories (lecture publique, écriture Admin)
/inventory      → Alertes stock bas, ajustements (Admin uniquement)
/shipping       → Carnet d'adresses, estimation frais de port
/admin          → Statistiques tableau de bord (Admin uniquement)
/health         → État de santé de l'API (PostgreSQL + Redis)
```

Sécurité

Authentification JWT

L'API utilise une architecture double-token :

```
┌──────────────────────────────────────────────────────────┐
│  Access Token (15 min)    ← Envoyé dans: Authorization: Bearer
│  Stockage client: mémoire JS (jamais localStorage)       │
│                                                          │
│  Refresh Token (7 jours)  ← Envoyé dans: Cookie HttpOnly │
│  Stockage serveur: Redis (cache) + PostgreSQL (source vérité)│
└──────────────────────────────────────────────────────────┘
```

Pour accéder aux endpoints protégés, le client doit inclure l'access token dans chaque requête :

```
GET /api/v1/cart
Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Hachage des Mots de Passe

PBKDF2-SHA512 avec 100 000 itérations (norme OWASP 2024) et sel aléatoire unique par utilisateur. La comparaison utilise `crypto.timingSafeEqual` pour prévenir les timing attacks.

Protections en Place

- **Helmet** — durcissement des headers HTTP (CSP, HSTS, XSS Protection)
- **CORS** — origines autorisées configurables via `CORS_ORIGINS`
- **Rate Limiting** — limites séparées pour les routes d'auth et les routes générales
- **Sanitizer** — échappement HTML sur tous les champs body/query/params
- **Requêtes paramétrées** — protection systématique contre les injections SQL

Infrastructure Docker

L'application tourne avec trois services Docker orchestrés par Compose :

```
# Résumé de l'architecture Docker
h1watch_db     → PostgreSQL 15 (port 5433 exposé)
h1watch_redis  → Redis 7 (port 6379 exposé)
h1watch_api    → Node.js 20 Alpine (port 3001 exposé)
```

Le service `api` dépend de `db` (avec healthcheck `pg_isready`) et de `redis` (condition `service_started`), garantissant l'ordre de démarrage correct.

```
# Démarrer tous les services
docker compose up -d
```

```
# Voir les logs en temps réel
docker compose logs -f api

# Rebuilder l'image API après modification du code
docker compose up -d --build api

# Arrêter et supprimer les containers (données persistées dans les volumes)
docker compose down

# Arrêter ET supprimer les volumes (reset complet de la BDD)
docker compose down -v
```

Health Check

```
curl http://localhost:3001/health
# Réponse 200 si tout est UP :
# { "uptime": 42, "services": { "database": "UP", "cache": "UP" } }
# Réponse 503 si une dépendance est DOWN
```

Tests

Le projet utilise **Vitest** pour les tests unitaires.

```
# Lancer tous les tests une fois
npm run test:run

# Mode watch (relance à chaque sauvegarde)
npm test

# Interface graphique Vitest
npm run test:ui

# Rapport de couverture de code
npm run test:coverage
```

Organisation des Tests

Les tests se trouvent dans `src/tests/` et couvrent les domaines principaux :

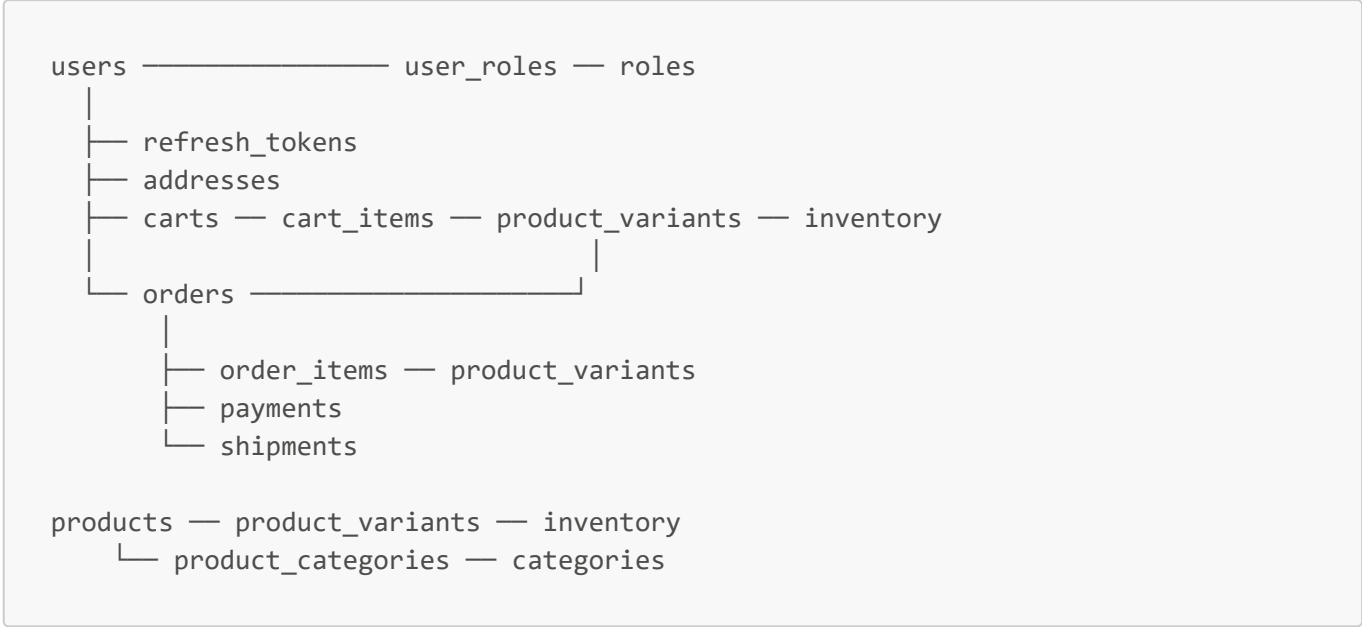
```
src/tests/
├── auth.service.test.js        → Inscription, connexion
├── cart.controller.test.js     → Contrôleur panier
├── inventory.service.test.js   → Ajustements de stock
├── order.service.test.js       → Création de commande
├── payment.service.test.js     → Sessions Stripe, webhooks
```

```
├── products.service.test.js    → Catalogue, détail produit
└── shipping.service.test.js    → Frais de port, expéditions
```

Structure du Projet

```
h1watch-backend/
├── docker/
│   ├── Dockerfile              # Image Node.js 20 Alpine
│   └── .dockerignore
├── src/
│   ├── config/
│   │   ├── database.js         # Pool PostgreSQL (pg)
│   │   ├── environment.js      # Variables d'env centralisées + validation
│   │   ├── instruments.js      # Initialisation Sentry (doit être chargé EN
PREMIER)
│   │   └── security.js         # Helmet, CORS, Rate Limiters
│   ├── constants/
│   │   ├── enums.js            # Valeurs ENUM PostgreSQL (ORDER_STATUS, etc.)
│   │   ├── errors.js           # Messages d'erreur centralisés
│   │   └── httpStatus.js       # Codes HTTP nommés
│   ├── controllers/            # Extraction params + délégation au Service
│   ├── middlewares/
│   │   ├── auth.middleware.js  # Vérification JWT, hydratation req.user
│   │   ├── role.middleware.js  # RBAC (restrictTo)
│   │   ├── sanitizer.middleware.js
│   │   ├── validator.middleware.js
│   │   └── erroHandler.middleware.js # Handler global des erreurs
│   ├── models/
│   │   └── index.js            # TypeDefs JSDoc pour l'autocomplétion
│   ├── repositories/
│   │   ├── _mappers.js         # snake_case → camelCase centralisé
│   │   ├── index.js            # Barrel export
│   │   ├── users.repo.js
│   │   ├── products.repo.js
│   │   ├── carts.repo.js
│   │   ├── orders.repo.js
│   │   ├── inventory.repo.js
│   │   └── ...
│   ├── routes/                 # Définition des routes Express
│   ├── services/               # Logique métier (Singletons)
│   ├── tests/                  # Tests unitaires Vitest
│   └── utils/
│       ├── appError.js         # Hiérarchie d'erreurs (AppError, NotFoundError,
etc.)
│       ├── asyncHandler.js     # Wrapper Express async → catch automatique
│       ├── logger.js           # Logging + intégration Sentry
│       └── validation.js       # Validateurs (UUID, email, password, slug...)
├── init-postgres.sql           # Schéma SQL initial (tables, indexes, enums)
├── docker-compose.yml
└── package.json
```

Schéma de Base de Données



Les **enums PostgreSQL** utilisés : `user_role_enum`, `order_status_enum`, `payment_status_enum`, `product_status_enum`.

La vue `view_inventory_status` agrège produits, variantes et stocks pour le tableau de bord admin.

Cron Jobs

Un job s'exécute **toutes les heures** pour libérer le stock réservé par des commandes `PENDING` non payées depuis plus de 24 heures, et les passer au statut `CANCELLED`. Configuré dans `src/app.js` via `node-cron`.

Licence

ISC — Voir `package.json`.