ECOM-WATCH

Documentation Technique Backend

Architecture Node.js · Pattern Service-Repository

Sécurité · Intégrité transactionnelle · Automatisation

Projet	ECOM-WATCH — Plateforme E-commerce Haute Horlogerie
Périmètre	Backend Node.js (API REST)
Version du document	v1.0 — Édition initiale
Date de rédaction	Juin 2025
Technologie principale	Node.js 20 · Express v5 · PostgreSQL 15
Stack complémentaire	Redis · Stripe · Cloudinary · Sentry · Docker
Classification	Confidentiel — Usage interne uniquement

TABLE DES MATIÈRES

CHAPITRE 1

Introduction Architecturale Backend

ECOM-WATCH est une plateforme e-commerce dédiée à la vente de montres de prestige. Son backend constitue la colonne vertébrale fonctionnelle de l'ensemble du système : il expose une API REST consommée par le frontend React déployé sur Vercel, orchestre les transactions financières via Stripe, et garantit la cohérence des données en toutes circonstances. Le présent document constitue la référence technique exhaustive du périmètre backend.

Cette documentation s'adresse aux architectes logiciels, développeurs seniors et équipes de revue de code souhaitant comprendre non seulement le quoi de l'implémentation, mais surtout le pourquoi de chaque décision architecturale. Chaque patron de conception retenu est justifié par rapport aux alternatives envisagées et aux contraintes propres à un contexte e-commerce de niche luxe.

1.1 Vision globale et justifications technologiques

Le backend repose sur Node.js 20 en mode ES Modules natifs avec Express v5. Ce choix technologique est motivé par plusieurs impératifs : la capacité à gérer de nombreuses connexions concurrentes grâce au modèle événementiel non-bloquant de Node.js, la cohérence avec l'écosystème JavaScript du frontend (partage de constantes, de types et de schémas de validation), et l'écosystème npm mature pour les intégrations tierces (Stripe, Cloudinary, Sentry).

Express v5 est retenu pour sa stabilité, sa légèreté et sa composabilité. Contrairement à des frameworks plus opiniâtres comme NestJS ou Fastify, Express laisse au développeur la liberté d'organiser son architecture selon les besoins du projet, sans contrainte de convention imposée. Cette liberté est exploitée pour mettre en place un Service Layer Pattern rigoureux, décrit en section 1.2.

PostgreSQL 15 constitue la base de données relationnelle centrale. Son choix face à des alternatives NoSQL (MongoDB) s'explique par la nature transactionnelle des données e-commerce : les opérations de réservation de stock, de création de commande et de confirmation de paiement nécessitent des garanties ACID (Atomicité, Cohérence, Isolation, Durabilité) qu'une base relationnelle fournit nativement. Redis complète ce dispositif comme couche de cache et de persistance de sessions.

> ■ ***Note architecturale :*** *Le modèle d'E/S non-bloquant de Node.js le rend particulièrement adapté aux charges I/O-bound (requêtes HTTP, accès base de données, appels API Stripe/Cloudinary). Pour des traitements CPU-intensifs, des Workers Threads ou un service dédié seraient recommandés.*

1.2 Le Service Layer Pattern

Le patron architectural fondateur de ce backend est le Service Layer Pattern (ou Architecture en Couches). Ce patron impose une séparation stricte des responsabilités en trois couches distinctes et indépendantes :

- Couche Contrôleur (/controllers) : Responsable exclusivement de l'interface HTTP. Un contrôleur reçoit la requête Express, valide les paramètres d'entrée, délègue la logique au service approprié et formate la réponse HTTP. Il ne contient aucune logique métier ni aucun accès direct à la base de données.
- Couche Service (/services) : Orchestre la logique métier. Un service coordonne les appels aux repositories, applique les règles métier (calcul de TVA, validation de stock, cycle de vie des commandes), gère les transactions et expose une interface stable aux contrôleurs.
- Couche Repository (/repositories) : Responsable exclusivement de l'accès aux données. Un repository construit les requêtes SQL, exécute les opérations sur PostgreSQL et retourne des objets JavaScript normalisés. Il est le seul endroit où du SQL est écrit.

Ce découplage garantit une testabilité unitaire de chaque couche de manière isolée, une réutilisabilité maximale (un même service peut être appelé par plusieurs contrôleurs ou par des cron jobs) et une maintenabilité à long terme (modifier le schéma SQL n'impacte que les repositories, pas les services ni les contrôleurs).

Couche	Dossier	Responsabilité principale
Entrée HTTP	/controllers	Validation entrée, formatage réponse
Routage	/routes	Mapping URL → Contrôleur, application middlewares
Logique Métier	/services	Orchestration, règles e-commerce, transactions
Accès Données	/repositories	SQL PostgreSQL, normalisation des données
Tâches de fond	/jobs	Cron jobs, nettoyage, archivage

Tableau 1.1 — Couches du Service Layer Pattern

1.3 Stack technique détaillée

Technologie	Version	Rôle
Node.js	20 (LTS)	Runtime JavaScript côté serveur
Express	v5.2	Framework HTTP minimaliste
PostgreSQL	15	Base de données relationnelle principale
Redis	7	Cache applicatif et stockage de sessions
Stripe SDK	v20	Traitement des paiements et webhooks
Cloudinary	v1.41	Hébergement et optimisation des images
Sentry	v10	Monitoring d'erreurs et traçabilité en production
node-cron	v4.2	Planification des tâches automatisées
jsonwebtoken	v9	Génération et vérification des JWT
Vitest	v4	Framework de tests unitaires et d'intégration

Docker	Compose	Conteneurisation locale (DB, API, Redis)

Tableau 1.2 — Stack technique du backend

CHAPITRE 2

Infrastructure et Sécurité — /config

Le dossier /config constitue la colonne vertébrale de l'infrastructure de l'application. Il regroupe l'ensemble des modules de configuration dont chaque démarrage du serveur dépend : variables d'environnement, connexion à la base de données, paramétrage de sécurité et intégrations tierces. La philosophie qui gouverne ce dossier est le principe de Fail-Fast : toute configuration manquante ou invalide doit provoquer une erreur immédiate et explicite au démarrage, plutôt qu'une défaillance silencieuse à l'exécution d'une requête.

2.1 Philosophie de la gestion des variables d'environnement

src/config/environment.js

Ce fichier constitue le point d'entrée unique pour toutes les variables d'environnement de l'application. Sa conception répond à une problématique architecturale fondamentale : éviter la dispersion des accès à process.env dans l'ensemble de la base de code, ce qui rendrait toute migration entre environnements (développement, staging, production) particulièrement risquée et difficile à auditer.

Le module commence par déclarer une liste de variables obligatoires (requiredEnv) et vérifie leur présence dès l'import. Si une ou plusieurs variables sont absentes, une erreur est levée avec un message explicite listant les variables manquantes — aucun accès à une valeur undefined n'est possible par construction. Cette stratégie de validation anticipée transforme les erreurs de configuration en crashs immédiats au démarrage, bien avant que la première requête HTTP n'arrive.

La configuration PostgreSQL fait l'objet d'une logique spécifique : elle accepte soit une DATABASE_URL complète (format utilisé par les hébergeurs cloud tels que Neon ou Render) soit un ensemble de paramètres individuels (POSTGRES_HOST, POSTGRES_USER, etc.), offrant une flexibilité maximale entre environnement local et cloud.

L'objet ENV exporté est gelé via Object.freeze() : cette immutabilité au runtime prévient les mutations accidentelles de configuration depuis n'importe quelle partie de l'application. ENV est la source de vérité de configuration, lisible partout mais modifiable nulle part.

> ■ ***Note architecturale :*** *La centralisation dans environment.js présente un second avantage : en cas d'audit de sécurité, un seul fichier suffit pour identifier l'ensemble des variables sensibles consommées par l'application. La surface d'exposition est documentée et maîtrisée.*

src/config/instruments.js

Ce fichier initialise Sentry, la plateforme de monitoring d'erreurs en production. Son positionnement comme premier import dans le processus de démarrage (avant même Express) est intentionnel : Sentry doit intercepter les erreurs aussi tôt que possible dans le cycle de vie du processus, y compris les exceptions levées pendant la phase d'initialisation des autres modules.

L'instrumentation Sentry avec le profiler (@sentry/profiling-node) permet non seulement le suivi des erreurs, mais aussi le profilage des performances — identification des endpoints lents, des goulots d'étranglement et des transactions anormalement longues. En production, cet outil est indispensable pour maintenir la qualité de service d'une plateforme e-commerce.

2.2 Gestion du cycle de vie de la base de données

src/config/database.js

Ce module gère l'initialisation et le cycle de vie du pool de connexions PostgreSQL. L'utilisation d'un pool de connexions (Pool du driver pg) est fondamentale pour les performances : plutôt que d'ouvrir une nouvelle connexion TCP pour chaque requête HTTP (coûteux en latence et en ressources), le pool maintient un ensemble de connexions persistantes réutilisables. Le pool est configuré avec un maximum de 20 connexions simultanées, un délai d'inactivité de 30 secondes et un timeout de connexion de 5 secondes.

La fonction connectPostgres() implémente une validation de connexion au démarrage, appelée après que le serveur HTTP a commencé à écouter. Ce choix d'ordre de démarrage est délibéré : sur les plateformes cloud (Render), le service doit répondre aux health checks HTTP avant que la base de données soit disponible. Le serveur accepte donc les requêtes dès son démarrage, mais retourne une erreur 503 si la base n'est pas encore connectée — comportement géré par le middleware de health check.

La configuration SSL est adaptée dynamiquement selon l'environnement : en production, rejectUnauthorized est positionné à false pour accepter les certificats auto-signés des serveurs Neon (hébergeur PostgreSQL serverless), tout en maintenant le chiffrement TLS du flux de données.

2.3 Stratégie de sécurité — Rate Limiting, CORS, Helmet

src/config/security.js

Ce module concentre l'ensemble des mécanismes de défense périmétriques de l'API. Six rate limiters distincts y sont définis, chacun ciblant une surface d'attaque spécifique — cette granularité est préférable à un limiteur global unique qui serait soit trop permissif (laissant passer des attaques ciblées) soit trop restrictif (bloquant des usages légitimes).

Limiteur	Fenêtre	Max requêtes	Surface protégée
generalLimiter	15 min	100	Toutes les routes — rempart anti-scraping
authLimiter	15 min	Configurable	Login/Register — anti-brute-force
passwordChangeLimiter	15 min	3	Changement MDP — anti-force brute ciblée
passwordResetLimiter	1 heure	5	Reset MDP — anti-abus d'envoi d'emails

trackingGuestLimiter	15 min	5000	Suivi guest — anti-énumération de commandes
profileGeneralLimiter	15 min	5000	Profil — permissif pour le polling UI

Tableau 2.1 — Inventaire des Rate Limiters

La fonction getClientIp() mérite une attention particulière : elle extrait l'IP réelle depuis l'en-tête X-Forwarded-For lorsque l'application est déployée derrière un reverse proxy (Render, Heroku). Sans cette extraction, tous les utilisateurs partageraient l'IP du proxy, rendant le rate limiting totalement inefficace. La directive app.set('trust proxy', 1) dans app.js active cette confiance envers le proxy en amont.

Le middleware Helmet est configuré avec une Content Security Policy (CSP) restrictive qui autorise explicitement uniquement les origines nécessaires (Cloudinary pour les images, Sentry pour le reporting d'erreurs, le frontend Vercel pour les connexions). Cette liste blanche explicite réduit drastiquement la surface d'exploitation XSS.

La configuration CORS distingue le mode production (liste d'origines explicites incluant le domaine Vercel du frontend) du mode développement (sans restriction). Cette distinction évite l'erreur courante de déployer en production une configuration CORS permissive héritée du développement.

2.4 Configuration des services externes

src/config/cloudinary.js

Ce module configure l'upload d'images vers Cloudinary via multer-storage-cloudinary. La configuration du storage Cloudinary centralise les règles d'upload : dossier de destination, transformation des images à la réception, gestion des types de fichiers autorisés. En concentrant ces règles ici plutôt que dans les routes ou contrôleurs, toute modification des politiques d'upload (nouveau dossier, nouvelle transformation) s'applique globalement sans recherche dans la base de code.

src/config/multer.config.js

Ce module expose le middleware handleUpload qui encapsule la gestion des erreurs spécifiques à multer (taille de fichier dépassée, type MIME non autorisé). Cette encapsulation est nécessaire car multer lève des erreurs de type MulterError qui ne seraient pas capturées par le gestionnaire d'erreurs global sans traitement spécifique. Le wrapper garantit que ces erreurs sont transformées en AppError avec un code HTTP approprié avant d'atteindre le client.

CHAPITRE 3

Schémas de Données — init-postgres.sql & /models

La modélisation de la base de données est le fondement sur lequel repose l'ensemble de la logique métier. Un schéma mal conçu se traduit inévitablement par des requêtes complexes, des incohérences de données et des performances dégradées. Le schéma ECOM-WATCH est conçu pour refléter fidèlement les réalités du domaine e-commerce tout en anticipant les contraintes de concurrence inhérentes à une plateforme transactionnelle.

3.1 Philosophie de la modélisation relationnelle

init-postgres.sql — Schéma principal

Le fichier init-postgres.sql constitue la source de vérité de la structure de données. Sa conception repose sur plusieurs principes fondamentaux. Premièrement, l'usage d'UUID comme clé primaire sur toutes les entités critiques (users, products, orders) plutôt que des séquences entières : les UUID éliminent les risques d'énumération (un attaquant ne peut pas deviner l'ID suivant) et facilitent les migrations et fusions de données entre environnements.

Deuxièmement, la mise en place de types ENUM PostgreSQL (user_role_enum, order_status_enum, payment_status_enum, product_status_enum) garantit l'intégrité des valeurs directement au niveau de la base, sans dépendre de la logique applicative. Un statut de commande ne peut physiquement contenir qu'une valeur du cycle de vie défini.

Troisièmement, le déclencheur (trigger) update_updated_at_column est déclaré une seule fois et réutilisé sur toutes les tables qui nécessitent un horodatage de modification automatique. Cette centralisation évite la duplication de logique et garantit un comportement cohérent.

3.2 Entités principales et relations

Le schéma organise les données autour de six domaines métier interconnectés :

- Domaine Utilisateurs : Tables users, roles et user_roles (relation N:N) avec historique de mots de passe (password_history) et tokens de réinitialisation (password_reset_tokens). La table refresh_tokens maintient les sessions actives avec expiration automatique.
- Domaine Catalogue : Tables products et product_variants implémentent un modèle produit-variante qui reflète la réalité horlogère (une montre peut exister en plusieurs tailles de boîtier ou coloris). La relation N:N via product_categories permet à un produit d'appartenir à plusieurs catégories.
- Domaine Inventaire : La table inventory est distincte de product_variants par design. Cette séparation des préoccupations permet de modifier les politiques de stock (colonne reserved_stock) sans altérer la définition du produit. L'inventaire peut évoluer indépendamment du catalogue.
- Domaine Promotions : Tables promotions, product_promotions et variant_promotions permettent d'appliquer des remises à deux niveaux de granularité (produit entier ou variante spécifique),

avec des dates de validité et des types de remise (PERCENTAGE ou FIXED).

- Domaine Commandes : La table orders supporte nativement les commandes guest (user_id IS NULL) et authentifiées. Les adresses sont stockées en JSONB plutôt qu'en tables normalisées : ce choix est délibéré — une adresse de commande est un instantané figé au moment de l'achat, et ne doit pas changer si l'utilisateur modifie ses adresses ultérieurement.
- Domaine Expédition et Paiement : Tables shipments et payments tracent respectivement les informations logistiques et les transactions financières, chacune rattachée à une commande.

3.3 Gestion de la concurrence et de l'intégrité

migrations/002_production_optimizations.sql

Ce fichier de migration complète le schéma initial avec des optimisations critiques pour la production. La création d'index composites sur les colonnes fréquemment filtrées (status + created_at sur orders, variant_id sur inventory) réduit significativement le temps de réponse des requêtes de liste et de recherche sous charge.

Les vues matérialisées (materialized views) pour les statistiques du dashboard constituent un mécanisme d'optimisation essentiel. Recalculées périodiquement par le cron stats-refresh, elles évitent de recalculer des agrégats coûteux (SUM, COUNT, GROUP BY sur des millions de commandes) à chaque visite du tableau de bord administrateur.

Les fonctions stockées cleanup_abandoned_orders() et cleanup_expired_tokens() encapsulent des opérations de maintenance récurrentes directement en base. Cette approche présente un double avantage : d'une part, les opérations s'exécutent en SQL natif sans aller-retour réseau entre l'application et la base ; d'autre part, elles restent transactionnelles et bénéficient pleinement des garanties ACID de PostgreSQL.

> ■ ***Note architecturale :*** *La colonne reserved_stock dans la table inventory est la clé du mécanisme de gestion concurrentielle du stock. Lorsqu'un utilisateur initie une commande, le stock est déplacé de available_stock vers reserved_stock — il n'est pas encore vendu, mais n'est plus disponible. La confirmation du paiement transforme cette réservation en vente définitive via confirmSale().*

CHAPITRE 4

La Couche d'Entrée — /controllers

Les contrôleurs constituent la frontière entre le protocole HTTP et la logique applicative. Conformément au Service Layer Pattern, leur responsabilité est strictement limitée : extraire les données de la requête, déléguer au service approprié et formater la réponse. Aucun contrôleur ne doit contenir de requête SQL, de règle métier ou de logique de validation complexe.

4.1 Rôle des contrôleurs comme orchestrateurs de requêtes

Chaque méthode d'un contrôleur suit un schéma invariant en quatre étapes : extraction des paramètres depuis req.params, req.body, req.query et req.user ; appel du service approprié avec ces paramètres ; formatage de la réponse via l'utilitaire response.js ; et propagation des erreurs vers le gestionnaire global via next(). L'encapsulation dans asyncHandler() garantit que les erreurs asynchrones (rejections de Promises) sont correctement transmises à Express, évitant les silences d'erreur dans les handlers asynchrones.

4.2 Analyse des contrôleurs métier

src/controllers/auth.controller.js

Le contrôleur d'authentification orchestre les quatre flux du cycle de vie de session : inscription (register), connexion (login), renouvellement de token (refresh) et déconnexion (logout). Un point de conception notable est la gestion du refresh token : il est transporté via un cookie HttpOnly plutôt qu'une réponse JSON, ce qui le rend inaccessible au JavaScript côté client. La méthode refresh extrait ce cookie via req.cookies.refreshToken avant de le transmettre au service.

La réponse du login et du register inclut les champs claimedOrders et claimedOrderNumbers : ces données permettent au frontend de synchroniser son état localStorage en retirant les commandes guest qui ont été automatiquement rattachées au compte lors de la connexion — un mécanisme de réconciliation transparent pour l'utilisateur.

src/controllers/product.controller.js

Ce contrôleur expose les opérations CRUD du catalogue produits avec une distinction nette entre les routes publiques (getAll, getOne) et administratives (create, update, delete, addVariant). Pour la création, la logique de désérialisation du champ variant (potentiellement transmis comme chaîne JSON dans un formulaire multipart) est délibérément maintenue dans le contrôleur car elle relève du format HTTP, non de la logique métier.

La route validate-variants illustre un cas où une logique simple est maintenue dans la route elle-même plutôt que déléguée à un contrôleur dédié. Ce pragmatisme est acceptable pour des opérations de lecture sans effets de bord, où la création d'un contrôleur complet serait une sur-ingénierie.

src/controllers/order.controller.js

Ce contrôleur est le plus complexe de l'application car il doit gérer deux modes d'accès radicalement différents : l'utilisateur authentifié (req.user disponible) et le visiteur non authentifié (mode guest, accès par email). La méthode getOrderDetail illustre cette dualité : si req.user est présent, l'accès est authentifié ; sinon, l'email est extrait des query params pour vérification côté service.

La méthode checkout, point d'entrée du tunnel de commande, démontre la séparation des préoccupations : le contrôleur ne calcule aucun total, ne valide aucun stock. Il se contente de transmettre les données au service et de retourner la réponse. La complexité transactionnelle (réservation de stock, calcul de taxes, persistance de commande) est entièrement gérée par orderService.

src/controllers/payment.controller.js

La méthode handleStripeWebhook présente une contrainte technique importante : la vérification de signature Stripe nécessite le corps brut (Buffer) de la requête, pas son interprétation JSON. C'est pourquoi app.js préserve req.rawBody via la fonction verify d'express.json() uniquement pour les routes /webhook. Le contrôleur transmet ce rawBody au service sans transformation.

La distinction entre les pages de succès (handleSuccess) et d'annulation (handleCancel) est déléguée à des méthodes dédiées qui retournent des redirections HTML simples. Ces pages sont des points de retour Stripe post-paiement : leur rôle est uniquement de rediriger vers le frontend avec les paramètres appropriés.

src/controllers/admin.controller.js

Le contrôleur administrateur expose deux endpoints de monitoring du dashboard : les statistiques KPI (getStats) et l'historique des ventes (getSalesHistory). Sa sobriété reflète la philosophie des contrôleurs : toute la logique d'agrégation, de calcul de moyennes et de formatage des données réside dans adminService, non dans ce fichier.

src/controllers/inventory.controller.js

Ce contrôleur expose les opérations de consultation et de gestion du stock à destination des administrateurs. Il inclut également les alertes de stock bas (getLowStockAlerts), qui alimentent le dashboard admin en temps réel. La méthode adjustStock distingue deux modes d'opération (réapprovisionnement vs ajustement négatif) via le signe de la quantité transmise.

CHAPITRE 5

Le Routage — /routes

Le dossier /routes définit l'interface publique de l'API. Chaque fichier de route est responsable d'un domaine fonctionnel spécifique. L'ordre de déclaration des middlewares et des routes au sein de chaque fichier est critique : Express les évalue séquentiellement, et une route déclarée après une autre ne sera jamais atteinte si la première capte la requête.

5.1 Organisation des points d'entrée de l'API

src/routes/index.routes.js

Ce fichier est le routeur racine de l'application. Il agrège les treize sous-routeurs métier sous le préfixe /api/v1 et applique le generalLimiter en tête de chaîne — avant toute route spécifique. Ce positionnement garantit que le rate limiter global s'applique à l'ensemble du trafic API sans exception, quelle que soit la route appelée.

L'inclusion du versioning /api/v1 dans l'URL de base est une décision de conception délibérée. Elle permet l'introduction d'une version v2 de l'API sans rupture de compatibilité avec les clients existants, et signale explicitement aux consommateurs de l'API que celle-ci est versionnée et évolutive.

Préfixe de route	Module	Domaine fonctionnel
/auth	auth.routes.js	Authentification, sessions, reset de mot de passe
/users	users.routes.js	Gestion du profil utilisateur et administration des comptes
/products	products.routes.js	Catalogue produits, variantes, filtres
/categories	categories.routes.js	Gestion des catégories produit
/promotions	promotions.routes.js	Promotions et remises
/cart	cart.routes.js	Panier d'achat (authentifié)
/orders	order.routes.js	Cycle de vie des commandes (guest et authentifié)
/shipping	shipping.routes.js	Frais de port et expédition
/payments	payment.routes.js	Sessions Stripe, webhooks, confirmations
/inventory	inventory.routes.js	Gestion des stocks (administration)
/taxes	tax.routes.js	Calcul des taxes par pays
/admin	admin.routes.js	Dashboard et monitoring (ADMIN uniquement)

/	sitemap.routes.js	Sitemap XML pour le référencement

Tableau 5.1 — Inventaire des modules de routes

5.2 Hiérarchie des ressources et sécurité par route

src/routes/auth.routes.js

Ce module définit les routes d'authentification avec une stratégie de rate limiting différenciée. Les routes /login et /register appliquent authLimiter (strict, anti-brute-force), tandis que /forgot-password et /reset-password appliquent passwordResetLimiter (encore plus strict, limité à 5 tentatives par heure). La validation des données d'entrée (format email, robustesse du mot de passe) est réalisée en middleware de route via les fonctions de utils/validation.js, avant même d'atteindre le contrôleur.

src/routes/products.routes.js

Ce fichier illustre une contrainte de routage Express critique documentée en commentaire : les routes statiques (/filters, /validate-variants) doivent impérativement être déclarées avant les routes paramétriques (/:idOrSlug). Sans cet ordre, Express interpréterait "filters" comme une valeur de paramètre plutôt que comme un segment de chemin statique, rendant la route inaccessible.

La séparation des permissions est appliquée via router.use(protect, restrictTo('ADMIN')) : toutes les routes déclarées après cette ligne héritent automatiquement de la double protection authentification + rôle ADMIN, sans avoir à répéter les middlewares sur chaque route individuelle. Ce mécanisme exploite le pattern Middleware Chain d'Express.

src/routes/order.routes.js

Ce module implémente la dualité guest/authentifié à travers trois mécanismes distincts. Le middleware optionalAuth, contrairement à protect, n'échoue pas si aucun token n'est présent : il enrichit req.user s'il existe, laisse req.user undefined sinon. Cette approche permet aux routes de commande de servir les deux types d'utilisateurs sans duplication.

La regex ORDER_NUMBER_REGEX (/^ORD-\d{4}-\d{1,10}$/) validant le format du numéro de commande pour la route de suivi guest est déclarée au niveau du module et non dans le handler. Cette extraction facilite les modifications futures du format et centralise la définition en un seul endroit.

src/routes/payment.routes.js

L'architecture des routes de paiement reflète la réalité des webhooks Stripe : ces derniers sont des appels HTTP émis par Stripe vers le backend, sans authentification JWT. Les routes /webhook/stripe et /webhook/paypal sont donc délibérément publiques, la sécurité étant assurée par la vérification de signature HMAC dans le service.

src/routes/admin.routes.js

La protection des routes admin est appliquée en bloc via router.use(protect) suivi de router.use(restrictTo('ADMIN')) en tête du fichier. Ce pattern garantit que l'intégralité des routes définies dans ce module est protégée par les deux middlewares sans risque d'oubli sur une route spécifique. Ce dossier expose également les routes de gestion des cron jobs (status, execute, stop, restart) qui permettent aux administrateurs de superviser et déclencher manuellement les tâches

planifiées.

CHAPITRE 6

Le Cœur Métier — /services

Les services constituent le cœur de l'application. Ils orchestrent les appels aux repositories, appliquent les règles métier, gèrent les transactions PostgreSQL et coordonnent les interactions entre domaines. Chaque service est implémenté comme un Singleton : une seule instance est créée et partagée pour toute la durée de vie de l'application, garantissant un état cohérent et économisant les ressources de construction.

6.1 Architecture générale des Services

Tous les services partagent un même pattern structurel. Le constructeur vérifie si une instance existe déjà (pattern Singleton avec guard clause) et retourne l'instance existante si c'est le cas. L'instance est ensuite gelée via Object.freeze() pour prévenir toute modification accidentelle de ses méthodes ou propriétés depuis l'extérieur. Cette immutabilité est une garantie de fiabilité dans un environnement concurrent.

Les méthodes privées (préfixées # en JavaScript moderne) sont utilisées extensivement pour encapsuler les helpers internes : calculs de coût, invalidation de cache, comparaisons timing-safe. Ces méthodes ne font pas partie de l'interface publique du service et ne peuvent être appelées depuis l'extérieur, maintenant une séparation stricte entre interface et implémentation.

6.2 Service d'Authentification

src/services/auth.service.js

Ce service orchestre l'inscription, la connexion et le renouvellement de session. Sa responsabilité la plus critique est de garantir l'atomicité de la création de compte : la création de l'utilisateur et l'attribution du rôle USER par défaut s'exécutent dans une seule transaction PostgreSQL. En cas d'échec sur l'attribution du rôle, l'utilisateur n'est pas créé — évitant ainsi des comptes orphelins sans rôle qui seraient inaccessibles pour toute opération nécessitant des permissions.

Le mécanisme d'auto-claim (autoClaimGuestOrders) est déclenché automatiquement lors du login et de l'inscription. Il transfère silencieusement toutes les commandes passées en mode guest avec le même email vers le compte nouvellement authentifié. Cette réconciliation automatique améliore significativement l'expérience utilisateur : un client ayant commandé sans créer de compte retrouve son historique dès sa première connexion.

La méthode privée #createAuthSession est partagée entre register et login, respectant le principe DRY (Don't Repeat Yourself). Elle génère les tokens, crée la session et retourne l'objet utilisateur enrichi des rôles — ce dernier point est essentiel pour que le frontend puisse afficher le lien vers le dashboard admin dès la connexion, sans attendre un refresh.

> ■ ***Note architecturale :*** *La notification d'inscription (#sendRegistrationNotification) est déclenchée en fire-and-forget : une erreur SMTP n'échoue jamais l'inscription. Ce choix de conception priorise la robustesse métier (l'utilisateur doit pouvoir créer son compte même si le serveur mail est défaillant) sur la certitude de délivrance de l'email de bienvenue.*

6.3 Service de Gestion des Tokens

src/services/token.service.js

Ce service encapsule toute la logique JWT (JSON Web Token). Il maintient une distinction claire entre les deux types de tokens selon leur finalité et leur durée de vie. L'access token (court terme, 15 minutes par défaut) transporte les claims d'identité (sub, email, roles) nécessaires à l'autorisation de chaque requête. Le refresh token (long terme, 7 jours) ne transporte que l'identifiant de l'utilisateur (sub) et sert uniquement à renouveler l'access token.

Les tokens sont signés avec des secrets distincts (JWT_ACCESS_SECRET et JWT_REFRESH_SECRET). Cette séparation est une pratique de sécurité fondamentale : si un secret venait à être compromis, seul le type de token correspondant serait affecté. La révocation sélective de tous les access ou refresh tokens est possible indépendamment.

La méthode verifyAccessToken et verifyRefreshToken retournent null en cas d'échec (token expiré, signature invalide) plutôt que de lever une exception. Ce choix permet aux appelants de traiter l'échec de vérification comme un cas normal (session expirée) sans avoir à gérer une exception try/catch.

6.4 Service de Gestion des Sessions

src/services/session.service.js

Ce service implémente une stratégie hybride de persistance de sessions : Redis comme couche rapide pour les lectures fréquentes, PostgreSQL comme source de vérité pour la persistance durée. Lors de la création d'une session, le refresh token est persisté simultanément en base et en cache. Lors de la validation, Redis est consulté en priorité ; en cas d'absence (expiration du cache, redémarrage de Redis), la base est interrogée et le cache est reconstruit — mécanisme dit de self-healing.

La configuration des cookies HttpOnly est adaptée dynamiquement selon l'environnement. En production, SameSite: None avec Secure: true est requis pour les cookies cross-domain (frontend Vercel vers API Render). En développement, SameSite: Lax avec Secure: false est utilisé car les navigateurs bloquent SameSite: None sans HTTPS.

6.5 Service de Sécurité des Mots de Passe

src/services/password.service.js

Ce service implémente le hachage des mots de passe via PBKDF2 (Password-Based Key Derivation Function 2) avec SHA-512. Le choix de PBKDF2 face à bcrypt est justifié par les recommandations OWASP 2024 : PBKDF2-SHA512 avec 100 000 itérations minimum offre une résistance équivalente aux attaques par force brute, tout en étant natif à Node.js (module crypto)

sans dépendance externe.

La comparaison de mots de passe utilise crypto.timingSafeEqual() plutôt qu'une comparaison directe (===). Cette précaution élimine les timing attacks : une comparaison octet par octet qui s'arrête au premier octet différent révèle, par la différence de durée d'exécution, la longueur du préfixe commun entre le hash tenté et le hash réel. timingSafeEqual garantit un temps d'exécution constant indépendamment de la position du premier octet différent.

La génération d'un salt aléatoire cryptographiquement fort (crypto.randomBytes) pour chaque utilisateur est fondamentale : deux utilisateurs ayant le même mot de passe produiront des hashes différents, rendant les attaques par table arc-en-ciel (rainbow tables) inopérantes.

6.6 Service Produits

src/services/products.service.js

Ce service gère l'ensemble du cycle de vie du catalogue produits avec une optimisation Redis systématique. La stratégie cache-aside est appliquée sur les lectures : la clé de cache est calculée à partir de l'identifiant ou du slug du produit, Redis est consulté en premier, et la base n'est interrogée qu'en cas de cache miss. Les mutations (create, update, delete) invalident systématiquement les entrées de cache concernées.

La méthode createProductWithVariant illustre la création transactionnelle multi-entités : produit, associations de catégories, variante initiale et entrée d'inventaire sont créés dans une seule transaction atomique. Tout échec partiel (variante créée mais inventaire non initialisé) est prévenu par le rollback automatique.

La méthode privée #resolveDisplayPrice présente une logique de résolution de prix promotionnel subtile. Elle parcourt l'intégralité des variantes pour trouver le prix le plus bas promu — y compris les variantes non présentées en premier dans la liste. Cette exhaustivité garantit que le prix affiché en vitrine correspond toujours à la meilleure offre disponible, quelle que soit l'ordre des variantes retournées par la base.

> ■ ***Note architecturale :*** *La clé de cache du catalogue (catalog:list:*) est construite à partir d'un hash Base64 des filtres de recherche. Cette approche garantit que chaque combinaison de filtres possède une entrée de cache distincte, sans risque de collision entre des ensembles de filtres différents.*

6.7 Service Commandes

src/services/orders.service.js

Ce service est le plus complexe de l'application. Il orchestre le cycle de vie complet des commandes, de la création à l'annulation, en gérant à la fois les utilisateurs authentifiés et les invités. Sa méthode centrale, createOrderFromCart, encapsule le flux transactionnel le plus critique du système : réservation de stock, résolution des prix promotionnels, calcul des totaux (sous-total, frais de port, taxes) et persistance atomique de l'ensemble.

La méthode #resolveEffectivePrice illustre un principe de cohérence transactionnelle : elle s'exécute à l'intérieur de la même transaction que la réservation de stock. Cette co-localisation garantit que le prix utilisé lors de la création de commande correspond exactement à la promotion active au même

instant, sans possibilité qu'une promotion expire entre la lecture du prix et la création de la commande.

La méthode cancelOrderAndReleaseStock est la source de vérité pour toute annulation, qu'elle soit déclenchée par l'utilisateur, le webhook Stripe ou le cron de nettoyage. Trois garanties sont offertes : atomicité (la mise à jour du statut et la libération du stock sont indivisibles), idempotence (une annulation sur une commande déjà annulée n'a aucun effet) et traçabilité (le motif d'annulation est logué pour l'audit).

Scénario d'annulation	Déclencheur	Motif logué
Utilisateur clique "Annuler"	cancelPendingOrder()	user_cancel
Session Stripe expirée	Webhook checkout.session.expired	checkout.session.expired
Commande PENDING > 24h	Cron orders-cleanup	Fonction SQL cleanup_abandoned_orders()
Admin met à jour le statut	updateOrderStatus() → CANCELLED	admin_status_update

Tableau 6.1 — Sources d'annulation de commande et garanties

La comparaison d'emails #timingSafeEmailCompare protège l'endpoint de suivi guest contre les timing attacks. En mode guest, la seule vérification d'identité est l'email : une comparaison classique révèlerait par la durée d'exécution si l'email est partiellement correct. La comparaison en temps constant élimine ce vecteur d'attaque.

La méthode #artificialDelay() introduit une latence aléatoire entre 200ms et 500ms sur les réponses "commande introuvable" en mode guest. Ce délai artificiel prévient l'énumération de commandes par chronométrage des réponses : un attaquant ne peut pas distinguer "commande inexistante" de "email incorrect" par la durée de réponse.

6.8 Service Paiement

src/services/payment.service.js

Ce service gère l'intégralité de l'intégration Stripe. Sa responsabilité principale est de créer des sessions Stripe Checkout et de traiter les webhooks entrants. La création de session inclut la configuration dynamique selon le mode d'accès (guest vs authentifié) : pour un guest, Stripe collecte l'email directement ; pour un utilisateur authentifié, l'email est récupéré depuis la base et pré-rempli dans la session.

La méthode _handleCheckoutCompleted est la plus critique du service. Elle s'exécute lors de la réception du webhook checkout.session.completed et effectue deux opérations dans une seule transaction : la mise à jour du statut de commande en PAID et la confirmation définitive de la sortie de stock (confirmSale). L'atomicité de ces deux opérations est fondamentale : un paiement confirmé sans confirmation de stock laisserait le stock réservé indéfiniment, corrompant les niveaux d'inventaire.

L'idempotence de la gestion des webhooks est implémentée dans _handleCheckoutExpired : avant d'annuler la commande, le service vérifie son statut actuel. Si elle est déjà PAID ou CANCELLED, l'événement est ignoré sans erreur. Cette idempotence est essentielle car Stripe peut retransmettre

un même webhook plusieurs fois en cas d'échec de délivrance.

> ■ ***Note architecturale :*** *La vérification de signature Stripe (webhooks.constructEvent) utilise le rawBody Buffer de la requête, pas son équivalent JSON parsé. Express modifie le corps lors de la désérialisation JSON, invalidant la signature. La préservation du rawBody dans app.js via la fonction verify d'express.json() est donc une précondition architecturale indispensable au traitement correct des webhooks.*

6.9 Service Inventaire

src/services/inventory.service.js

Ce service orchestre la gestion des niveaux de stock avec une approche axée sur la résilience et l'intégrité. Sa méthode cleanupExpiredReservations est appelée par le cron inventory-cleanup toutes les 15 minutes. Elle identifie les commandes PENDING dont la création remonte à plus de 30 minutes (délai supérieur à la durée de vie d'une session Stripe, pour éviter d'annuler une commande en cours de paiement) et libère atomiquement leur stock réservé.

L'utilisation de Promise.allSettled() plutôt que Promise.all() pour le traitement par lot des commandes expirées est un choix de résilience délibéré. Avec Promise.all(), une erreur sur une seule commande interromprait le traitement de l'ensemble du lot. Promise.allSettled() garantit que chaque commande est traitée indépendamment ; les échecs sont logués mais n'empêchent pas les traitements suivants.

La constante #LOW_STOCK_THRESHOLD (5 unités) est centralisée dans le service plutôt que dispersée dans les différents appelants (cron, dashboard). Cette centralisation garantit qu'un seul changement de valeur propage l'ajustement à l'ensemble des alertes et affichages qui en dépendent.

6.10 Service Expédition

src/services/shipping.service.js

Ce service gère le calcul des frais de port selon une grille tarifaire organisée par zones géographiques (FRANCE, EUROPE, INTERNATIONAL) et méthodes de livraison (STANDARD, EXPRESS, RELAY). La grille est définie comme propriété privée du service, rendant toute modification tarifaire centralisée et auditable sans toucher à la logique de calcul.

La formule de calcul applique systématiquement la règle de franco de port : si le sous-total de commande dépasse le seuil défini pour la zone et la méthode choisies, le coût retourné est 0 et le flag isFree est levé. Cette logique est encapsulée dans calculateShippingCost(), la seule méthode qui réalise ce calcul, garantissant une cohérence absolue entre les frais affichés en prévisualisation et ceux facturés lors du checkout.

Le mapping pays → zone (#countryZones) avec une valeur DEFAULT pour les pays non listés garantit que le service ne lève jamais d'erreur sur un pays inconnu : il le traite comme une destination internationale. Cette stratégie de dégradation gracieuse est préférable à un crash qui bloquerait le tunnel de commande d'un client international.

6.11 Service Taxes

src/services/tax.service.js

Ce service centralise la logique de calcul de la TVA selon les règles fiscales par pays. La table des taux (#taxRates) couvre les principales destinations européennes avec leurs différents niveaux de taxation (standard, réduit, intermédiaire, super-réduit). Ce regroupement en un seul service garantit que toute modification réglementaire (changement de taux de TVA dans un pays) est appliquée à l'ensemble de la plateforme en modifiant un seul fichier.

La méthode extractTaxFromTotal() réalise l'opération inverse du calcul standard : elle décompose un montant TTC en ses composantes HT et TVA. Cette méthode est utile pour les rapports comptables et les déclarations fiscales où la distinction HT/TVA est obligatoire.

6.12 Service Panier

src/services/cart.service.js

Ce service gère la persistance et la cohérence du panier pour les utilisateurs authentifiés. Il implémente le pattern cache-aside avec une TTL longue (86 400 secondes = 24 heures) : le panier est persistant en base via cartsRepo, mais Redis évite les lectures répétées à chaque changement de page. Toute mutation (ajout, suppression, modification de quantité) invalide immédiatement l'entrée de cache correspondante.

La validation de stock est réalisée en temps réel avant chaque ajout : la disponibilité est vérifiée contre la base de données (source de vérité), non contre le cache. Cette approche évite la vente d'articles dont le stock aurait été épuisé entre la mise en cache et l'ajout au panier.

La méthode mergeCarts() résout le cas d'usage de la fusion entre un panier guest (existant avant connexion) et le panier du compte connecté. Elle délègue à addToCart() pour chaque article, bénéficiant ainsi des validations de stock et de l'invalidation de cache déjà implémentées dans cette méthode.

6.13 Service Promotions

src/services/promotions.service.js

Ce service gère les promotions applicables à deux niveaux de granularité : le produit entier ou une variante spécifique. L'invalidation du cache est particulièrement soignée : lors de la création ou modification d'une promotion, le service reconstruit les clés de cache de chaque produit affecté (par id et par slug) et les invalide. Cette invalidation ciblée évite de purger l'intégralité du cache pour une modification qui n'affecte que quelques produits.

La validation des dates (#validateDates) et du pourcentage (#validatePercentage) sont des helpers privés appelés systématiquement avant toute écriture. Cette validation anticipée (fail-fast) évite de persister des promotions incohérentes (date de fin antérieure à la date de début, remise supérieure à 100%) qui produiraient des calculs de prix aberrants.

6.14 Service Cache

src/services/cache.service.js

Ce service encapsule l'accès à Redis avec sérialisation/désérialisation JSON automatique. Son rôle est de fournir une interface unifiée (get, set, delete, deleteMany) à l'ensemble des services qui utilisent Redis, masquant les détails de connexion et de sérialisation. La méthode deleteMany() utilise Promise.all() pour paralléliser les suppressions de plusieurs clés simultanément, minimisant la latence des opérations d'invalidation de cache complexes.

6.15 Service Notifications

src/services/notifications/notification.service.js

Ce service centralise la logique de dispatch des notifications email selon les événements métier. Son rôle est de décider quand et quel email envoyer, en déléguant le comment à emailService. La méthode notifyOrderStatusChange orchestre l'envoi de la notification appropriée (confirmation de paiement, expédition, livraison, annulation) selon la transition de statut observée, sans duplication dans les différents services qui modifient le statut des commandes.

La résolution de l'adresse email du destinataire (_getCustomerEmail) illustre la dualité guest/authentifié : elle priorise l'email de l'adresse de livraison (disponible pour les deux types) avant de tenter une résolution par l'ID utilisateur. Cette hiérarchie garantit que les notifications atteignent le bon destinataire quelle que soit la nature de la commande.

6.16 Service Administration

src/services/admin.service.js

Ce service agrège les données transversales nécessaires au tableau de bord administrateur. La méthode getDashboardStats() lance les quatre requêtes d'agrégation (nombre d'utilisateurs, statistiques de ventes, alertes de stock, nombre de produits) en parallèle via Promise.all(). Cette parallélisation réduit le temps de chargement du dashboard à celui de la requête la plus lente, plutôt qu'à la somme de toutes les durées si elles étaient exécutées séquentiellement.

La méthode getSalesHistory() expose l'historique journalier des ventes pour le graphique de chiffre d'affaires. La validation du paramètre days (borné entre 1 et 365 jours) prévient les requêtes d'agrégation excessivement coûteuses sur des fenêtres temporelles indéfiniment larges.

CHAPITRE 7

Persistance des Données — /repositories

La couche repository est la seule autorisée à écrire du SQL dans cette application. Ce confinement strict du SQL garantit que les optimisations de requêtes (ajout d'index, réécriture de jointures, utilisation de vues matérialisées) restent localisées sans impact sur les couches supérieures. Chaque repository est un objet exporté contenant des méthodes asynchrones — pas de classe, pas d'héritage, une interface fonctionnelle simple et testable.

7.1 Stratégie d'accès aux données

Les repositories acceptent tous un paramètre optionnel client (instance PoolClient) qui se substitue au pool global pgPool lorsqu'il est fourni. Ce pattern permet aux services de passer leur client de transaction aux repositories, garantissant que toutes les opérations d'une transaction s'exécutent sur la même connexion avec la même visibilité des données. Sans cette capacité, une transaction qui insère une commande et réserve du stock serait répartie sur deux connexions différentes, brisant l'atomicité.

Les requêtes utilisent systématiquement des paramètres positionnels ($1, $2, ...) plutôt que des interpolations de chaînes. Cette pratique élimine les injections SQL par construction : le driver pg envoie la requête et les paramètres séparément au serveur PostgreSQL, qui les traite comme des données, jamais comme du SQL exécutable.

7.2 Le Mapper camelCase

src/repositories/_mappers.js

Ce module expose deux fonctions utilitaires, mapRow() et mapRows(), qui convertissent automatiquement les noms de colonnes PostgreSQL (snake_case, convention SQL standard) en propriétés JavaScript (camelCase, convention JS). Cette transformation systématique garantit une cohérence de nommage dans l'ensemble de la couche applicative : aucun service ni contrôleur n'a à gérer les deux conventions simultanément.

La centralisation de cette logique de transformation est préférable à sa duplication dans chaque repository. Si la convention de nommage devait évoluer (par exemple pour supporter TypeScript avec des interfaces strictes), un seul fichier est à modifier.

7.3 Repository Produits

src/repositories/products.repo.js

Ce repository est le plus complexe de l'application. Sa méthode list() construit dynamiquement une requête SQL qui agrège, en une seule passe, les informations du produit, ses variantes, ses catégories, ses niveaux de stock et ses promotions actives. La construction de ce JSON complexe directement en SQL (jsonb_build_object, json_agg, COALESCE) évite le problème N+1 : sans cette approche, chaque produit listé nécessiterait des requêtes supplémentaires pour charger ses

variantes, catégories et promotions.

La méthode findActivePromotionPrice() résout le prix effectif d'une variante en tenant compte des promotions de deux niveaux (variante directe et produit parent). Cette résolution est exprimée en SQL via des jointures CASE WHEN : l'engine PostgreSQL résout le calcul de prix (ROUND(v.price * (1 - discount_value / 100.0), 2)) en base, sans aller-retour applicatif. La méthode accepte un client de transaction pour garantir qu'elle observe le même snapshot de données que la réservation de stock qui la précède dans createOrderFromCart.

> ■ ***Note architecturale :*** *La requête de getFullDetails() retourne intentionnellement un JSON complètement assemblé (catégories, variantes avec inventaire et promotion) en une seule requête SQL. Ce choix privilégie la latence perçue (une seule requête réseau) sur la lisibilité SQL. Le commentaire explicatif dans le code documente ce compromis pour les futurs mainteneurs.*

7.4 Repository Inventaire

src/repositories/inventory.repo.js

Ce repository est le gardien de l'intégrité des niveaux de stock. Ses méthodes de mutation (reserve, release, confirmSale) implémentent toutes une forme de vérification atomique de précondition directement dans la requête SQL UPDATE.

La méthode reserve() illustre ce principe : la clause WHERE available_stock >= $2 garantit que la mise à jour ne s'effectue que si le stock disponible est suffisant. Si rowCount est 0, la mise à jour n'a pas eu lieu (stock insuffisant) et une BusinessError est levée. Cette approche atomique évite la race condition classique : "vérifier le stock, puis décrémenter" — dans un système concurrent, deux requêtes simultanées pourraient toutes deux vérifier que le stock est disponible, puis toutes deux décrémenter, produisant un stock négatif.

La méthode release() utilise GREATEST(0, reserved_stock - $2) pour protéger contre un reserved_stock négatif en cas de désynchronisation. Même si une erreur de logique amenait à libérer plus de stock que réservé, la contrainte GREATEST(0, ...) empêche une valeur négative en base.

Méthode	Opération SQL	Garantie
reserve()	available -= q, reserved += q (si available >= q)	Atomique, anti-race-condition
release()	available += q, reserved -= q (GREATEST 0)	Anti-stock négatif
confirmSale()	reserved -= q (si reserved >= q)	Idempotente, anti-désynchronisation
addStock()	available += q	Réapprovisionnement simple
upsert()	INSERT ... ON CONFLICT DO UPDATE	Création ou mise à jour idempotente

Tableau 7.1 — Garanties des opérations de stock

7.5 Repository Commandes

src/repositories/orders.repo.js

Ce repository applique un principe de sécurité fondamental décrit dans son en-tête : la colonne user_id est la source de vérité pour le périmètre d'accès. Les méthodes publiques de lecture guest (findGuestOnlyById, findByOrderNumberAndEmail, findGuestOrdersByEmail) incluent systématiquement la condition user_id IS NULL dans leur clause WHERE, directement en SQL, pas en logique applicative. Cette approche garantit qu'aucune erreur de programmation dans la couche service ne pourrait exposer une commande privée en mode guest.

La méthode transferOwnership() (rattachement d'une commande guest à un compte) utilise FOR UPDATE pour verrouiller la ligne lue : ce verrou pessimiste empêche deux requêtes concurrentes de réclamer simultanément la même commande, ce qui pourrait affecter plusieurs utilisateurs. La vérification que user_id IS NULL dans la transaction garantit l'idempotence : une commande déjà rattachée ne peut pas être rattachée une seconde fois.

La comparaison d'emails dans findByOrderNumberAndEmail() est réalisée en JavaScript avec crypto.timingSafeEqual() plutôt qu'en SQL. Ce choix est délibéré : le WHERE SQL retournerait des résultats différents selon que l'email est correct ou non, révélant ainsi de l'information. En récupérant la commande sans filtre email puis en comparant en mémoire, le temps de réponse de la requête SQL est identique quel que soit l'email fourni.

7.6 Repository Utilisateurs

src/repositories/users.repo.js

Ce repository applique le principe de moindre privilège sur les données exposées. La méthode list() utilise une projection explicite des colonnes (id, email, first_name, last_name, phone, is_active, created_at) plutôt qu'un SELECT *. Cette précaution garantit que password_hash et salt ne peuvent jamais apparaître dans une réponse JSON sérialisée par inadvertance.

La normalisation de l'email en minuscules est appliquée dès la persistance (LOWER($1) dans l'INSERT) et dans toutes les recherches (LOWER(email) = LOWER($1) dans les SELECT). Cette double normalisation garantit l'unicité fonctionnelle de l'email indépendamment de la casse saisie par l'utilisateur.

L'historique des mots de passe (getPasswordHistory, addToHistory) alimente la fonctionnalité de prévention de réutilisation de mot de passe. Lors d'un changement de mot de passe, le service consulte les 5 derniers hashes pour vérifier que le nouveau mot de passe ne correspond pas à l'un d'eux — protection contre la rotation cyclique de mots de passe.

CHAPITRE 8

Automatisation et Tâches de Fond — /jobs

Les tâches de fond automatisées sont essentielles à la maintenance d'une plateforme e-commerce en bonne santé. Elles traitent les problèmes systémiques qui ne peuvent pas être résolus de façon synchrone dans le flux de requête HTTP : libération du stock abandonné, nettoyage des sessions expirées, mise à jour des statistiques. L'architecture /jobs sépare la définition des tâches (dossier cron/) de leur orchestration (dossier schedulers/).

8.1 Architecture du Scheduler centralisé

src/jobs/schedulers/cronScheduler.js

La classe CronScheduler est l'orchestrateur central de toutes les tâches planifiées. Elle maintient un Map de jobs enregistrés, chacun associé à une expression cron, une fonction d'exécution et une référence à la tâche node-cron. Cette architecture centralisée offre plusieurs avantages par rapport à l'instanciation directe de tâches cron dans chaque module : un point unique de démarrage et d'arrêt de l'ensemble des tâches, une interface d'introspection (listJobs) permettant au dashboard admin de visualiser l'état des crons, et la possibilité de déclencher manuellement n'importe quelle tâche via executeNow() pour les besoins de maintenance ou de test.

La validation de l'expression cron (cron.validate(schedule)) avant l'enregistrement prévient les erreurs de configuration silencieuses : une expression cron invalide ne serait jamais déclenchée, et sans cette validation, le problème ne serait découvert qu'à l'occasion d'un audit ou d'un incident. Toutes les tâches s'exécutent dans le fuseau Europe/Paris, garantissant un comportement prévisible quelle que soit la localisation du serveur (Render peut être hébergé dans différentes régions).

8.2 Inventaire des Cron Jobs

Nom	Expression cron	Fréquence	Responsabilité
inventory-cleanup	*/15 * * * *	Toutes les 15 min	Libère le stock des commandes PENDING > 30 min
orders-cleanup	30 3 * * *	Quotidien 3h30	Annule les commandes PENDING > 24h (fonction SQL)
sessions-cleanup	0 3 * * *	Quotidien 3h00	Supprime les refresh tokens expirés
stats-refresh	0 * * * *	Toutes les heures	Rafraîchit les vues matérialisées du dashboard
orders-archive	0 4 1 * *	1er du mois 4h	Archive les commandes > 2 ans dans orders_archive

Tableau 8.1 — Inventaire des Cron Jobs et leurs responsabilités

src/jobs/cron/inventory.cron.js — Nettoyage du stock

Ce cron est le plus fréquent (toutes les 15 minutes) car la libération du stock réservé est une contrainte temps-réel critique. Sans ce mécanisme, le stock d'une montre populaire pourrait rester bloqué indéfiniment par des paniers abandonnés, rendant l'article apparemment indisponible alors que les unités sont physiquement en stock. La fenêtre de 30 minutes (supérieure au délai d'expiration d'une session Stripe) est calibrée pour ne jamais annuler une commande dont le paiement est encore en cours.

Le retour null en cas d'erreur (plutôt qu'un re-throw) garantit que l'échec de ce cron n'interrompt pas l'exécution des autres tâches planifiées. La résilience partielle est préférable à une défaillance totale du scheduler.

src/jobs/cron/orders.cron.js — Nettoyage des commandes

Ce cron délègue l'intégralité de son traitement à la fonction SQL cleanup_abandoned_orders(), définie dans migrations/002. Cette délégation à la base de données est un choix de performance : la fonction SQL traite les annulations en une seule opération atomique sur le moteur PostgreSQL, sans aller-retour réseau pour chaque commande. L'exécution à 3h30 du matin minimise l'impact sur la charge applicative quotidienne.

src/jobs/cron/sessions.cron.js — Nettoyage des sessions

Ce cron maintient la table refresh_tokens à une taille raisonnable en supprimant les entrées expirées. Sans ce nettoyage, la table croîtrait indéfiniment, dégradant progressivement les performances des requêtes d'authentification. La fonction SQL cleanup_expired_tokens() supprime toutes les lignes dont expires_at est dans le passé, en une seule instruction DELETE efficace.

src/jobs/cron/stats.cron.js — Rafraîchissement des statistiques

Ce cron toutes les heures maintient la fraîcheur des vues matérialisées PostgreSQL qui alimentent le dashboard administrateur. Les vues matérialisées sont des "snapshots" pré-calculés de requêtes d'agrégation coûteuses. Sans MATERIALIZED VIEW, chaque visite du dashboard déclencherait un GROUP BY complet sur l'ensemble de la table orders. Avec la vue, la réponse est quasi-instantanée au prix d'un rafraîchissement horaire qui accepte une légère décalage des données.

src/jobs/cron/archive.cron.js — Archivage des commandes

Ce cron mensuel déplace les commandes de plus de deux ans dans une table d'archivage distincte (orders_archive). Cette opération de tiering de données préserve les performances des requêtes opérationnelles courantes : une table orders contenant uniquement les commandes récentes est plus petite, ses index sont moins fragmentés et ses scans sont plus rapides. Les données archivées restent consultables mais dans une table dédiée, séparée du flux transactionnel quotidien.

CHAPITRE 9

Flux Transverses — Middlewares & Utils

Les middlewares et les utilitaires constituent l'infrastructure transverse de l'application. Ils s'appliquent de façon horizontale à l'ensemble des requêtes ou sont réutilisés par de multiples composants. Leur conception détermine la cohérence, la sécurité et la maintenabilité de l'ensemble du système.

9.1 Middleware d'Authentification

src/middlewares/auth.middleware.js

Le middleware protect est le gardien de toutes les routes protégées. Son rôle est d'extraire le JWT Bearer de l'en-tête Authorization, de le vérifier, de charger l'utilisateur depuis la base et d'enrichir req.user avec ses données et rôles. La vérification de la propriété isActive garantit qu'un compte suspendu par un administrateur perd immédiatement l'accès, même si son token est encore valide — le token est invalide de fait, même si sa signature est correcte.

Le chargement des rôles depuis la base (rolesRepo.listUserRoles) plutôt que depuis le token JWT est un choix de sécurité délibéré. Si les rôles étaient uniquement stockés dans le token, une élévation de privilèges (ajout du rôle ADMIN à un compte) ne prendrait effet qu'à la prochaine génération d'un token, créant une fenêtre d'incohérence. En rechargeant les rôles à chaque requête, le changement de rôle est instantanément effectif.

src/middlewares/optionalAuth.middleware.js

Ce middleware est la version non-bloquante de protect. Il tente de vérifier le token si présent, mais ne lève aucune erreur s'il est absent ou invalide. Il enrichit req.user uniquement si l'authentification réussit, laissant req.user à undefined pour les visiteurs non identifiés. Ce middleware permet aux routes de commande de servir les deux types d'utilisateurs avec une logique unifiée.

src/middlewares/role.middleware.js

Le middleware restrictTo(role) est un générateur de middleware : il prend un ou plusieurs rôles en paramètre et retourne un middleware qui vérifie que req.user possède au moins l'un de ces rôles. Son positionnement systématiquement après protect dans la chaîne de middlewares garantit que req.user est toujours défini au moment de la vérification des rôles.

9.2 Gestionnaire d'Erreurs Global

src/middlewares/erroHandler.middleware.js

Ce middleware est le dernier maillon de la chaîne Express. Sa signature à quatre paramètres (err, req, res, next) est obligatoire pour qu'Express le reconnaisse comme gestionnaire d'erreurs. Il centralise le formatage de toutes les réponses d'erreur, garantissant une structure de réponse cohérente quelle que soit la source de l'erreur.

La distinction development/production est fondamentale pour la sécurité : en développement, la stack trace complète et l'objet d'erreur entier sont exposés pour faciliter le débogage. En production, seul le message est exposé pour les erreurs opérationnelles (instances d'AppError avec isOperational: true), et un message générique "Une erreur interne est survenue" remplace tout bug non anticipé.

La normalisation des erreurs tierces (codes PostgreSQL 23505/23503/22P02, erreurs JWT) en AppError garantit que ces erreurs bénéficient du même traitement que les erreurs applicatives : un message lisible, un code HTTP approprié, et une distinction opérationnelle/bug.

src/utils/appError.js

Ce module définit la hiérarchie des classes d'erreur de l'application. AppError étend Error nativement et introduit le flag isOperational, qui permet au gestionnaire d'erreurs de distinguer les erreurs attendues (stock insuffisant, email déjà utilisé, token invalide) des bugs imprévus. Quatre sous-classes spécialisent AppError avec des codes HTTP prédéfinis : NotFoundError (404), ValidationError (400), ConflictError (409), BusinessError (422).

9.3 Utilitaires applicatifs

src/utils/asyncHandler.js

Ce wrapper encapsule les handlers Express asynchrones pour propager automatiquement les rejections de Promise vers le middleware suivant (next). Sans asyncHandler, une exception levée dans un handler async passerait silencieusement, laissant la requête sans réponse. asyncHandler résout ce problème structurel d'Express pour les handlers asynchrones.

src/utils/logger.js

Le logger centralisé adapte son comportement selon l'environnement. En production, les logs sont formatés en JSON structuré pour être parsés par les agrégateurs de logs (Sentry, Datadog). En développement, un format lisible est utilisé. Le logger structure les informations d'erreur (message, stack, contexte) de façon cohérente, facilitant l'analyse post-mortem d'incidents.

src/utils/validation.js

Ce module expose des fonctions de validation réutilisables (validateEmail, validatePasswordStrength, validateUUID, validateSlug, validateRequired) utilisées dans les routes et les repositories. Centraliser ces validations garantit l'application uniforme des règles (format UUID v4, robustesse du mot de passe conforme NIST SP 800-63B, format slug URL-safe) sans duplication dans la base de code.

src/utils/response.js

Ce module standardise la structure des réponses API de succès. Toute réponse positive suit le format { status: 'success', data: {...} }, garantissant une interface prévisible pour le frontend. Cette standardisation facilite la gestion des réponses côté client (le frontend peut toujours accéder à response.data) et simplifie les tests d'intégration (une seule structure à vérifier).

src/utils/healthCheck.js

Ce module implémente la vérification de santé de l'application (endpoint GET /health). Il vérifie simultanément la connectivité PostgreSQL et Redis, retournant un rapport structuré avec le statut de chaque composant. Cet endpoint est appelé par le pipeline CD après chaque déploiement pour vérifier que le service est opérationnel, et par Render pour les health checks de routage.

CHAPITRE 10

Infrastructure de Déploiement — Docker & CI/CD

L'infrastructure de déploiement garantit la reproductibilité des environnements et l'automatisation du cycle de livraison. ECOM-WATCH adopte une philosophie Infrastructure-as-Code : chaque configuration d'environnement est versionnée dans le dépôt Git et peut être reconstruite de manière identique.

10.1 Containerisation Docker

docker/Dockerfile & docker-compose.yml

Le Dockerfile décrit l'image de production du backend. Il part d'une image Node.js 20 Alpine (minimaliste, réduit la surface d'attaque et la taille d'image), installe les dépendances de production uniquement (npm ci --only=production), et définit le point d'entrée de l'application.

Le docker-compose.yml orchestre trois services pour l'environnement de développement local : l'API backend (port 3001), PostgreSQL 15 Alpine (port 5433 exposé pour éviter les conflits avec une installation locale) et Redis 7 Alpine. La dépendance health check garantit que l'API ne démarre qu'une fois la base de données prête à accepter des connexions.

La persistance des données locales est assurée par des volumes Docker nommés (postgres_data, redis_data), garantissant que les données survivent aux redémarrages des conteneurs. Le fichier init-postgres.sql est monté en volume dans le conteneur PostgreSQL et exécuté automatiquement à l'initialisation de la base.

10.2 Pipeline CI/CD

.github/workflows/CI.yaml

Le pipeline d'Intégration Continue s'exécute sur chaque push et pull request vers les branches main et develop. Il comprend trois jobs parallélisés après validation du lint : vérification statique (ESLint), tests d'intégration avec des services PostgreSQL et Redis locaux au runner GitHub Actions, et validation du build Docker.

La configuration de concurrence (cancel-in-progress: true) annule automatiquement les runs en cours sur une branche dès qu'un nouveau commit arrive. Cette optimisation évite de consommer des minutes CI sur du code déjà remplacé.

Les tests d'intégration s'exécutent contre une base de données dédiée (ecom_watch_test) avec le schéma complet appliqué via init-postgres.sql et la migration 002. Cette isolation totale des tests vis-à-vis de la base de production garantit qu'aucun test ne peut corrompre des données réelles.

.github/workflows/CD.yaml

Le pipeline de Déploiement Continu s'exécute uniquement sur les pushs vers main. Il suit un ordre strict : migration de la base de données Neon en production, puis déclenchement du déploiement Render via deploy hook. Si la migration échoue, le déploiement n'est pas déclenché — garantissant

que le code déployé correspond toujours au schéma de base.

Le job migrate applique les scripts SQL dans l'ordre numérique sur la base Neon de production. Les deux scripts sont conçus pour être idempotents (IF NOT EXISTS, ON CONFLICT DO NOTHING, CREATE INDEX CONCURRENTLY IF NOT EXISTS) : leur ré-application sur une base existante est sans effet de bord, prévenant les erreurs de double application.

La vérification de disponibilité post-déploiement interroge l'endpoint /health toutes les 15 secondes pendant 5 minutes maximum. Si l'API ne répond pas par un HTTP 200 dans ce délai, le job échoue et l'équipe est notifiée. Ce polling évite de considérer un déploiement comme réussi alors que le service ne serait pas encore opérationnel.

Phase	Déclencheur	Jobs exécutés	Critère de succès
CI	Push/PR sur main ou develop	Lint + Tests + Build Docker	Tous les jobs passent
CD	Push sur main uniquement	Migration Neon + Deploy Render	API /health répond 200

Tableau 10.1 — Synthèse des pipelines CI/CD

ANNEXE A

Glossaire Technique

Terme	Définition
ACID	Atomicité, Cohérence, Isolation, Durabilité. Propriétés garantissant la fiabilité des transactions PostgreSQL.
Anti-pattern N+1	Situation où N+1 requêtes sont exécutées pour charger N entités et leurs associations. Résolu par les jointures SQL ou le chargement en lot.
Atomicité	Propriété garantissant qu'une transaction est soit entièrement appliquée, soit entièrement annulée (ROLLBACK).
Cache-aside	Pattern de cache : lecture dans le cache, puis en base si absent, puis mise en cache du résultat.
Fail-Fast	Stratégie de détection précoce des erreurs de configuration, au démarrage plutôt qu'à l'exécution.
Fire-and-forget	Pattern d'envoi asynchrone sans attente de la réponse (utilisé pour les notifications email).
Idempotence	Propriété d'une opération qui produit le même résultat quel que soit le nombre de fois qu'elle est exécutée.
Pool de connexions	Ensemble de connexions PostgreSQL pré-ouvertes et réutilisables, évitant le coût d'ouverture à chaque requête.
RBAC	Role-Based Access Control. Contrôle d'accès basé sur les rôles assignés à chaque utilisateur.
Race Condition	Comportement indéterministe causé par deux opérations concurrentes accédant à une ressource partagée sans coordination.
Repository Pattern	Abstraction de la couche d'accès aux données, exposant des méthodes métier plutôt que des opérations SQL brutes.
Service Layer	Couche architecturale encapsulant la logique métier, intermédiaire entre les contrôleurs et les repositories.
Singleton	Pattern garantissant qu'une seule instance d'une classe est créée et partagée dans l'application.
Timing Attack	Attaque exploitant la variation du temps de réponse pour inférer des informations sur des données secrètes.
Transaction	Ensemble d'opérations SQL exécutées comme une unité atomique, avec possibilité de COMMIT ou ROLLBACK.
Webhook	Callback HTTP envoyé par un service tiers (Stripe) pour notifier un événement asynchrone.

JWT	JSON Web Token. Standard de token d'authentification signé cryptographiquement.
PBKDF2	Password-Based Key Derivation Function 2. Algorithme de hachage de mot de passe résistant aux attaques par force brute.
Self-healing	Capacité d'un système à se réparer automatiquement (reconstruction du cache depuis la base en cas de perte Redis).
Materialized View	Vue PostgreSQL dont le résultat est pré-calculé et stocké, offrant des performances de lecture élevées.

— Fin de la documentation —

ECOM-WATCH · Documentation Technique Backend v1.0 · Juin 2025

Architecture Node.js — Confidentiel — Usage interne