ECOM-WATCH

Documentation Technique Frontend

Architecture React · Component-Hook-Service Pattern
Sécurité · Performance · SEO · Design System

Projet	ECOM-WATCH — Plateforme E-commerce Haute Horlogerie
Périmètre	Frontend React (SPA — Client-Side Rendering)
Version du document	v1.0 — Édition initiale
Date de rédaction	Juin 2025
Technologie principale	React 19 · Vite 7 · Tailwind CSS v4
Stack complémentaire	TanStack Query v5 · Zustand · React Router v7 · Zod
Auteur	Équipe Architecture Frontend — ECOM-WATCH
Classification	Confidentiel — Usage interne uniquement

TABLE DES MATIÈRES

Sommaire

CHAPITRE 1

Introduction Architecturale Frontend

ECOM-WATCH est une plateforme e-commerce dédiée à la vente de montres de prestige, déployée sur Vercel et construite selon les standards modernes du développement frontend React. Le présent document constitue la référence technique exhaustive de l'ensemble du périmètre frontend : architecture, sécurité, gestion de l'état, optimisation des performances, accessibilité et stratégie SEO.

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. Elle adopte une approche intentionnelle : 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

L'application est une **Single Page Application (SPA)** reposant sur un rendu côté client (Client-Side Rendering — CSR). Ce choix est justifié par la nature hautement interactive de l'expérience : gestion du panier en temps réel, galeries de produits avec zoom, tiroirs animés et tunnel de commande multi-étapes. Un rendu serveur (SSR) aurait introduit une complexité d'infrastructure disproportionnée au regard des bénéfices SEO, ces derniers étant compensés par une stratégie JSON-LD et react-helmet-async rigoureuse.

Le choix de **Vite 7** comme bundler répond à l'impératif de Developer Experience (DX) : temps de démarrage à froid sous 300ms grâce à l'exploitation des ES Modules natifs du navigateur, Hot Module Replacement (HMR) instantané, et tree-shaking optimisé à la compilation. Par rapport à Create React App (Webpack), Vite offre des performances de build en production supérieures de 3 à 10× sur des projets de cette taille.

React	19.2 — Concurrent Mode, Suspense natif, usage intensif de hooks
Vite	7.3 — Bundler ESM natif, HMR ultra-rapide
React Router	v7.13 — Routage déclaratif, loaders, lazy routes
TanStack Query	v5.90 — Cache serveur, invalidation, retry automatique
Zustand	v5.0 — État global léger sans boilerplate Redux
Axios	v1.13 — Client HTTP avec intercepteurs
Tailwind CSS	v4.1 — Utility-first, dark mode, design tokens
React Hook Form	v7.71 — Formulaires performants, intégration Zod
Zod	v4.3 — Validation typée des schémas
Recharts	v3.7 — Visualisation des données Admin
Lucide React	v0.563 — Bibliothèque d'icônes SVG optimisées

1.2 Le pattern Component-Hook-Service (CHS)

Le pattern architectural fondateur de cette application est le **Component-Hook-Service (CHS)**. Ce patron impose une séparation stricte des préoccupations (Separation of Concerns — SoC) en trois couches distinctes et indépendantes :

- **Couche Service** : Responsable exclusivement des appels réseau (HTTP). Un service ne contient aucun état React, aucun hook, aucune logique d'affichage. Il reçoit des paramètres, appelle l'API, normalise la réponse et la retourne.
- **Couche Hook (Custom Hook)** : Orchestre la logique métier et l'état local. Un hook consomme un ou plusieurs services, gère les états de chargement/erreur, applique la memoization, et expose une interface stable aux composants.
- **Couche Composant** : Se limite à la présentation. Un composant pur ne devrait contenir que des JSX, des gestionnaires d'événements simples, et des appels à des hooks. Il est dit "stupide" (dumb component) — il reçoit des props et affiche.

■ Justification du pattern CHS : Ce découplage garantit une testabilité unitaire de chaque couche de manière isolée, une réutilisabilité maximale (un même hook peut alimenter plusieurs composants) et une maintenabilité à long terme (modifier la logique API n'impacte pas les composants UI).

La figure conceptuelle suivante illustre le flux de données unidirectionnel respecté dans l'ensemble du projet, depuis la requête API jusqu'au rendu DOM :

Figure 1.1 — Flux de données unidirectionnel (CHS Pattern)

1.3 Organisation des dossiers

L'organisation des dossiers suit une structure dite **"feature-first"** (ou "vertical slice architecture"), où chaque fonctionnalité métier (cart, catalogue, checkout, orders) est un module autonome regroupant ses propres composants, hooks, services et types. Cette approche s'oppose à l'organisation "type-first" (tous les composants dans /components, tous les hooks dans /hooks) qui devient ingérable au-delà de quelques dizaines de fichiers.

ECOMWATCH/src/		
■■■ app/	Cœur applicatif	App.jsx, router.jsx, main.jsx
■■■ api/	Client HTTP global	axios.config.js, queryClient.js
■■■ core/	Configuration centrale	config/, constants/, utils/
■■■ pages/	Pages racines	home.jsx
■■■ shared/	Modules transverses	auth/, SEO/, UI/, theme/
■■■ apps/	Domaines métier	
■■■ clients/	Storefront public	features/, layout/, routes.jsx

■■■ admin/	Back-office	features/, layout/, routes.jsx

Figure 1.2 — Arborescence simplifiée du projet

1.4 Conventions architecturales

- **Nommage** : Les hooks customs sont préfixés **use** (useCartLogic, useAdminOrders). Les services sont suffixés **Service** (ProductService, GuestOrderService). Les composants de page sont suffixés **Page** ou **Admin**.
- **Exports** : Exports nommés pour les hooks et services, exports par défaut pour les composants. Cette convention facilite le tree-shaking et l'autocomplete des IDEs.
- **Séparation logique/présentation** : Aucun appel API direct dans un composant JSX. Toute logique asynchrone passe par un Custom Hook.
- **Immutabilité** : L'état React ne doit jamais être muté directement. Toutes les mises à jour passent par les fonctions de mise à jour Zustand ou les setters useState.
- **Lazy Loading** : Toutes les pages de l'application (storefront et admin) sont chargées en différé via React.lazy() pour optimiser le Time To Interactive (TTI).

CHAPITRE 2

Sécurité et Client HTTP — /api & /core/config

La couche de communication HTTP constitue le périmètre de sécurité le plus critique du frontend. Une mauvaise gestion des jetons d'authentification expose l'application aux attaques XSS (Cross-Site Scripting) et CSRF (Cross-Site Request Forgery). Cette section détaille les décisions architecturales prises pour mitiger ces risques.

2.1 Stratégie de sécurisation des jetons

L'architecture de sécurité repose sur une dualité de jetons conforme aux bonnes pratiques OWASP :

- **Access Token (JWT court terme)** : Stocké exclusivement en mémoire vive du module JavaScript (variable de module privée). Durée de vie typiquement de 15 minutes. Jamais persisté dans localStorage, sessionStorage ou un cookie accessible JavaScript.
- **Refresh Token (longue durée)** : Stocké dans un cookie HttpOnly géré exclusivement par le navigateur. Inaccessible à tout code JavaScript, même en cas d'injection XSS réussie.

■ Menace XSS et mitigation : Une attaque XSS consiste à injecter du code JavaScript malveillant dans la page pour voler les credentials stockés dans localStorage. En conservant l'Access Token en mémoire (Closure JavaScript), celui-ci devient physiquement inaccessible depuis la console DevTools ou un script injecté, éliminant cette surface d'attaque.

■ src/api/axios.config.js

Ce fichier constitue le **cœur de la couche sécurité frontend**. Il centralise trois responsabilités critiques : la configuration de l'instance Axios, la gestion du token en mémoire, et la mise en place des intercepteurs de requête/réponse.

2.2 Le Closure Pattern pour la gestion du token

La variable _accessToken est déclarée au niveau du **module** JavaScript (hors de toute fonction exportée), créant ainsi une fermeture (Closure) qui rend cette variable inaccessible depuis l'extérieur du module. Les seuls accès possibles passent par les trois fonctions exportées : setAccessToken(), getAccessToken() et clearAccessToken().

```
// Variable de module privée — inaccessible depuis window.*
let _accessToken = null;

export const setAccessToken = (token) => { _accessToken = token; };
export const getAccessToken = () => _accessToken;
export const clearAccessToken = () => { _accessToken = null; };

// Intercepteur de requête — injection automatique
api.interceptors.request.use((config) => {
if (_accessToken) {
config.headers.Authorization = `Bearer ${_accessToken}`;
```

```
}
return config; // Mode guest si pas de token
});
```

Cette implémentation garantit que le token ne peut jamais être lu depuis la console du navigateur (window._accessToken retourne undefined). La contrepartie acceptée de ce choix est la perte du token au rechargement de page (*hard refresh*), ce qui est intentionnel : la restauration de session est assurée par le Refresh Token via l'endpoint /auth/refresh lors du montage de l'AuthProvider.

2.3 Intercepteur de réponse — Rotation silencieuse des jetons

L'intercepteur de réponse implémente un mécanisme sophistiqué de **token rotation transparente**. Lorsque le serveur retourne une erreur HTTP 401 (Unauthorized) sur une requête d'un utilisateur connecté, l'intercepteur tente automatiquement de renouveler le token avant de retransmettre la requête échouée, sans que l'utilisateur perçoive l'interruption.

La condition critique distinguant un utilisateur connecté d'un invité (Guest) est la vérification de la présence d'un _accessToken en mémoire. Sans cette distinction, l'intercepteur tenterait un refresh pour chaque requête échouant en 401 dans le cadre du checkout invité, créant une boucle d'erreur inutile et une latence dégradée.

401 + token présent	→ Tentative de refresh → Retry de la requête originale
401 + pas de token	→ Propagation directe (mode Guest, comportement attendu)
401 sur /auth/refresh	→ Propagation directe (évite la boucle infinie)
Toute autre erreur	→ Propagation directe vers le composant appelant

■ Principe de Fail-Open vs Fail-Secure : Pour le panier (validateCart), l'application adopte une stratégie Fail-Open — en cas d'échec de la validation API, le panier est considéré valide pour ne pas bloquer l'utilisateur. Pour l'authentification, une stratégie Fail-Secure est appliquée : en cas d'échec du refresh, le token est effacé et l'utilisateur perd sa session.

■ `src/api/queryClient.js`

Ce fichier configure l'instance TanStack Query (anciennement React Query), le gestionnaire de cache serveur de l'application. La configuration globale définit des comportements par défaut appliqués à toutes les requêtes de l'application :

staleTime: 5 min	Les données restent "fraîches" 5 minutes. Aucune requête réseau redondante pendant ce délai.
cacheTime: 30 min	Les données inactives restent en cache mémoire 30 minutes avant garbage collection.
retry: 1	Une seule nouvelle tentative en cas d'échec réseau. Évite les hammering d'API instables.
refetchOnWindowFocus: false	Désactivé pour les données e-commerce : un retour sur onglet ne doit pas déclencher de requêtes non sollicitées sur les prix ou stocks.

2.4 Configuration de l'environnement

■ src/core/config/env.js

Ce fichier centralise l'accès aux variables d'environnement Vite (préfixées VITE_). Il constitue le **point unique de vérité** pour la configuration runtime de l'application, évitant la dispersion des import.meta.env.* dans l'ensemble du code. Cette centralisation facilite les migrations entre environnements (développement, staging, production) et prévient les erreurs de typo dans les noms de variables.

■ src/core/config/app.js

appConfig agrège les constantes métier de l'application : seuils de stock, limites de pagination, délais de debounce, clés de localStorage. Cette approche évite le **Magic Number anti-pattern** où des valeurs numériques sont dispersées dans le code sans contexte ni point de modification centralisé. Toute modification d'un seuil (ex: LOW_STOCK_THRESHOLD: 3) se propage automatiquement à l'ensemble des composants qui le consomment.

■ src/core/constants/orderStatus.js

Ce fichier illustre un usage avancé du patron **Enum-like Object Freeze** en JavaScript. L'objet ORDER_STATUS est gelé (Object.freeze()) pour garantir l'immuabilité des constantes au runtime, prévenant les mutations accidentelles. Il fournit également des fonctions utilitaires (isValidTransition(), getProgressPercentage()) qui centralisent la logique métier de workflow commande, évitant sa duplication entre les composants Admin et Storefront.

2.5 Utilitaires transverses

■ src/core/utils/logger.js

Le logger centralisé implémente un filtrage conditionnel selon l'environnement d'exécution. En production (IS_DEVELOPMENT = false), les niveaux info et debug sont silencieux, limitant les fuites d'information dans la console du navigateur et réduisant la surface d'information disponible pour une analyse de sécurité. Seuls les niveaux warn et error restent actifs en production, conformément aux bonnes pratiques de sécurité frontend.

■ src/core/utils/ScrollToTop.js

Ce composant utilitaire résout une limitation connue de React Router : lors de la navigation entre pages, la position de défilement n'est pas automatiquement réinitialisée. ScrollToTop écoute les changements de pathname via useLocation() et déclenche window.scrollTo avec un délai de 0ms (setTimeout), garantissant que le scroll s'exécute après le rendu React (queue de microtâches vidée).

CHAPITRE 3

Authentification — /shared/auth

Le module d'authentification est conçu selon le principe de la **responsabilité unique modulaire** : chaque aspect de l'authentification est isolé dans un fichier dédié, permettant une évolution indépendante de chaque composante (validation, état, service, UI). Le dossier /shared/auth est délibérément placé hors des dossiers /apps/clients et /apps/admin car il est partagé par les deux domaines de l'application.

3.1 Architecture du système d'authentification

Le système d'authentification repose sur une architecture en couches interdépendantes mais découplées :

Couche	Fichier(s)	Responsabilité
UI	Login.jsx, Register.jsx, ForgotPassword.jsx	Rendu des formulaires, feedback visuel
Form Logic	useAuthForm.js	Validation, soumission, navigation post-auth
State	useAuthStore.js (Zustand)	État global utilisateur, persistance mémoire
Orchestration	useAuth.js	Coordination service + store + side-effects
Service	auth.service.js	Appels HTTP vers /auth/* endpoints
Validation	auth.schema.js (Zod)	Schémas de validation des formulaires
Guards	GuestGuard.jsx, RoleGuard.jsx	Protection des routes par état/rôle
Context	AuthContext.jsx	Provider initialisant la session au montage

Tableau 3.1 — Couches du système d'authentification

3.2 État global avec Zustand (useAuthStore)

■ `src/shared/auth/hooks/useAuthStore.js`

useAuthStore implémente le store Zustand de l'état d'authentification. Zustand a été retenu face à Redux/Context API pour sa légèreté (moins de 1ko gzippé), son absence de boilerplate (pas d'actions, reducers, selectors distincts) et ses performances de re-rendu supérieures grâce à la sélection fine d'état (*selector pattern*).

Le store expose trois champs d'état (user, isAuthenticated, isInitialized) et trois mutateurs. Le flag isInitialized joue un rôle critique : il indique que la vérification de session initiale (appel à /auth/refresh au montage de l'app) est terminée, quel que soit son résultat. Sans ce mécanisme, l'application afficherait des routes protégées avant de savoir si l'utilisateur est connecté, causant des *flash of unauthorized content* (FOUC des droits).

3.3 Orchestration métier (useAuth)

■ src/shared/auth/hooks/useAuth.js

useAuth est le hook d'orchestration principal. Il combine le store Zustand, le service d'authentification, et la gestion des effets de bord liés à la session. Ses responsabilités incluent :

- **Initialisation de session** : La fonction checkAuth(), mémoïsée avec useCallback, est appelée une unique fois au montage de l'AuthProvider. Elle tente un refresh silencieux et met à jour le store selon le résultat.
- **Rattrapage des commandes Guest** : Lors du login ou de l'inscription, si des commandes ont été passées en mode invité avec le même email, le backend retourne claimedOrderNumbers. Le hook synchronise alors le localStorage pour retirer ces commandes du suivi local, évitant les doublons dans l'UI.
- **Nettoyage de session** : La fonction logout() appelle le service (révocation du refresh token côté serveur) puis nettoie le store et le token en mémoire, dans un bloc finally garantissant le nettoyage même en cas d'erreur réseau.

■ Gestion du claimedOrders : Ce mécanisme avancé résout un problème de cohérence d'état multi-surface. Quand un guest se connecte, ses commandes passées anonymement sont "réclamées" par son compte côté serveur. Le frontend doit alors nettoyer son localStorage pour éviter qu'un utilisateur voit ses propres commandes affichées deux fois (une fois en tant que guest, une fois en tant qu'authentifié).

3.4 Guards de navigation

■ src/shared/auth/guards/GuestGuard.jsx

GuestGuard protège les routes réservées aux utilisateurs non authentifiés (Login, Register). Si un utilisateur connecté tente d'accéder à /login, il est redirigé vers /. Le rendu est conditionné par isInitialized pour éviter un redirect prématuré pendant la vérification de session initiale.

■ src/shared/auth/guards/RoleGuard.jsx

RoleGuard étend le mécanisme à un contrôle basé sur les rôles (RBAC — Role-Based Access Control). Il accepte une prop allowedRoles et vérifie que l'utilisateur connecté possède au moins l'un des rôles autorisés via Array.prototype.some(). Ce composant protège l'intégralité de la section /admin, garantissant qu'un utilisateur avec le rôle USER ne peut jamais accéder au back-office, même en manipulant l'URL.

■ Limitation du contrôle d'accès côté client : Les Guards React protègent l'interface utilisateur mais ne constituent pas une barrière de sécurité définitive. La vérification des droits doit toujours être redondante côté serveur. Un utilisateur malveillant capable de modifier l'état Zustand en mémoire contournerait le RoleGuard — mais l'API backend refuserait ses requêtes avec une erreur 403.

3.5 Validation des formulaires (Zod)

■ src/shared/auth/schemas/auth.schema.js

Les schémas Zod définissent les règles de validation typées pour tous les formulaires d'authentification. L'usage de Zod en combinaison avec @hookform/resolvers/zod et React Hook Form présente plusieurs avantages architecturaux décisifs par rapport à la validation ad-hoc dans les composants :

- **Single Source of Truth** : Les règles de validation sont définies en un seul endroit et réutilisables sur plusieurs formulaires ou contextes (validation API côté client).
- **Inférence de types** : Zod génère automatiquement les types TypeScript (ou JSDoc) des données validées, réduisant les erreurs de typage.
- **Messages d'erreur centralisés** : Les messages sont définis au niveau du schéma et non éparpillés dans les composants, facilitant la localisation (i18n).
- **Validation conditionnelle** : Le schéma registerSchema utilise .refine() pour valider la correspondance des mots de passe, une validation cross-field impossible avec HTML5 natif.

Le schéma de registration impose des contraintes de complexité sur le mot de passe (8 caractères minimum, au moins une majuscule, au moins un chiffre) via des expressions régulières. Ces contraintes sont cohérentes avec les recommandations NIST SP 800-63B pour la gestion des mots de passe.

3.6 Hook de formulaire d'authentification

■ src/shared/auth/hooks/useAuthForm.js

useAuthForm est un Custom Hook polymorphique qui gère deux modes distincts (login et register) via un paramètre mode. Cette conception évite la duplication de logique entre les deux pages et maintient une interface cohérente. Le hook utilise useLocation pour implémenter la redirection post-login vers la page d'origine (pattern *redirect to referrer*), permettant à un utilisateur redirigé vers le login depuis une page protégée de retrouver son contexte après authentification.

La gestion des erreurs dans onSubmit illustre une distinction architecturale importante : les erreurs métier (identifiants invalides — HTTP 401/403) sont traitées comme des cas normaux et affichées via toast, tandis que les erreurs serveur inattendues (HTTP 500+) sont également loguées dans la console pour faciliter le débogage. Les erreurs métier ne sont délibérément pas loguées pour éviter de remplir les logs de production avec du bruit fonctionnel.

CHAPITRE 4

Gestion du Panier — /features/cart

La gestion du panier est l'une des fonctionnalités les plus critiques d'une application e-commerce. Elle doit supporter la dualité Guest/Authentifié, la persistance entre sessions, la validation en temps réel du stock, et l'animation fluide du tiroir panier. L'architecture retenue repose sur le **Context + Custom Hook Pattern**, avec une isolation stricte de la logique de chaque sous-problème.

4.1 Architecture du système de panier

Le système de panier comporte quatre niveaux d'abstraction :

- **CartContext + CartProvider** : Fournit l'état du panier à l'ensemble de l'arbre de composants via le Context API React. La clé de panier (cartKey) est calculée dynamiquement selon l'état d'authentification, isolant les paniers entre comptes.
- **useCartLogic** : Encapsule toute la logique métier (ajout, suppression, modification, validation, persistance). Ce hook est le seul autorisé à modifier l'état du panier.
- **useCart** : Hook de consommation simple. Valide que l'appel est bien effectué dans l'arbre du CartProvider.
- **CartDrawer** : Composant de présentation. Consomme useCart et délègue toute la logique au hook.

4.2 Isolation des paniers par utilisateur

■ `src/apps/clients/features/cart/context/CartProvider.jsx`

Le CartProvider implémente un pattern ingénieux d'**isolation par clé**. La prop key du composant interne CartProviderInner est calculée comme h1-cart-{user.id} pour un utilisateur authentifié, ou h1-cart-guest pour un invité. Lorsque cette clé change (connexion ou déconnexion), React **démonte et remonte** le composant enfant, réinitialisant complètement l'état du panier.

Ce mécanisme garantit qu'un utilisateur se connectant après avoir ajouté des articles en mode invité voit son panier correctement géré selon la politique de l'application (fusion ou remplacement), sans état résiduel du panier précédent. Cette isolation est préférable à une gestion manuelle de la transition dans useEffect, qui serait sujette aux race conditions.

4.3 Logique métier centralisée (useCartLogic)

■ `src/apps/clients/features/cart/hooks/useCartLogic.js`

useCartLogic est le hook le plus complexe de l'application. Il orchestre six problématiques distinctes :

4.3.1 Initialisation depuis localStorage

L'initialisation lazy du state via la fonction passée à useState garantit que le parsing JSON du localStorage n'est exécuté qu'une seule fois, lors du montage du composant. Cette technique d'**initialisation paresseuse** est critique pour les performances : sans elle, le parsing serait ré-exécuté à chaque re-rendu du Provider, ce qui sur une liste de 20+ articles représenterait une opération coûteuse répétée des dizaines de fois.

4.3.2 Validation asynchrone du stock

La fonction validateCart appelle l'endpoint /products/validate-variants avec la liste des identifiants de variantes du panier. Cet appel est mémoïsé avec useCallback (dépendances vides) pour éviter sa re-création à chaque rendu. L'utilisation d'une cartRef (référence mutable) au lieu de la dépendance directe à cart dans useCallback brise le cycle de dépendances qui résulterait autrement en une re-création infinie du callback.

La validation est déclenchée à trois moments clés :

- **Montage initial** : Une unique validation au montage permet de détecter les articles devenus indisponibles depuis la dernière session.
- **Ouverture du tiroir** : Validation immédiate puis polling toutes les 10 secondes pendant que le tiroir est ouvert, garantissant la cohérence en cas de stock modifié par un achat concurrent.
- **Avant checkout** : Validation bloquante avant la redirection vers le tunnel de commande.

4.3.3 Gestion des promotions dans le panier

Lors de l'ajout d'un article, le hook distingue le prix effectif (effectivePrice) du prix original (originalPrice). Si une promotion est active sur la variante, le prix remisé (promotion.discountedPrice) est utilisé comme prix de référence pour le calcul du total, tandis que le prix original est conservé pour l'affichage barré. Cette dualité garantit que le total du panier reflète toujours le prix facturé réel.

4.3.4 Memoization des valeurs dérivées

Les valeurs totalPrice, totalItems et hasOutOfStock sont calculées via useMemo, ce qui garantit leur recalcul uniquement quand l'état cart change effectivement. Sans memoization, ces calculs seraient exécutés à chaque re-rendu du Provider (ce qui inclut tous les re-rendus des composants parents), entraînant des calculs redondants particulièrement coûteux pour totalPrice sur un panier volumineux.

4.4 Persistance automatique

Un useEffect surveillant [cart, cartKey] assure la synchronisation automatique de l'état en mémoire vers le localStorage. Ce pattern de **persistance réactive** garantit que toute modification du panier (ajout, suppression, modification de quantité) est immédiatement sauvegardée, sans nécessiter d'appels explicites à une fonction de sauvegarde dans chaque mutateur. La combinaison de la clé de panier dans les dépendances garantit que le bon emplacement localStorage est utilisé après un changement de compte.

4.5 CartDrawer — Composant de présentation

■ `src/apps/clients/features/cart/components/CartDrawer/index.jsx`

Le CartDrawer est un composant de présentation pur qui délègue toute la logique au hook useCart. Il gère deux aspects d'expérience utilisateur essentiels :

- **Verrouillage du scroll** : Lorsque le tiroir est ouvert, document.body.style.overflow = "hidden" empêche le défilement de la page en arrière-plan, une exigence UX fondamentale pour les modales et tiroirs. Le nettoyage dans le return de l'useEffect garantit la restauration du scroll même si le composant est démonté pendant que le tiroir est ouvert.
- **Transition CSS** : L'animation d'entrée/sortie est réalisée via les classes Tailwind transform transition-transform translate-x-full/translate-x-0, tirant parti des transitions CSS nativement accélérées par le GPU (propriété transform), sans recours à des bibliothèques d'animation JavaScript.

■ `src/apps/clients/features/cart/components/CartDrawer/CartDrawerItem.jsx`

Ce composant illustre l'application rigoureuse des standards d'**accessibilité WCAG 2.1**. Chaque bouton d'action (supprimer, diminuer/augmenter la quantité) dispose d'un attribut aria-label descriptif incluant le nom de l'article, permettant aux lecteurs d'écran de contextualiser l'action. Le compteur de quantité utilise aria-live="polite" pour notifier les technologies d'assistance des changements de valeur sans interrompre la lecture en cours.

CHAPITRE 5

Catalogue et Produits — /features/catalogue & /products

Le module Catalogue constitue le cœur fonctionnel de l'expérience client. Il doit satisfaire des exigences contradictoires : filtrage instantané côté client pour une UX fluide, persistence des filtres dans l'URL pour le partage de liens (deep linking) et l'indexation SEO, et affichage performant de potentiellement plusieurs centaines de produits.

5.1 Architecture URL-driven du Catalogue

■ src/apps/clients/features/catalogue/hooks/useCatalogueLogic.js

useCatalogueLogic illustre le pattern architectural **URL comme source de vérité** (URL as State). Plutôt que de stocker les filtres (recherche, catégorie, plage de prix) dans un state React local, ils sont persistés dans les paramètres de l'URL via useSearchParams. Cette décision architecturale majeure offre plusieurs bénéfices :

- **Deep Linking** : Un utilisateur peut partager un lien filtré (/catalogue?category=sport&minPrice=1000&maxPrice=5000) et le destinataire verra exactement le même résultat.
- **SEO** : Les robots d'indexation peuvent crawler les pages de catégorie avec leurs URLs canoniques.
- **Historique navigateur** : Les filtres sont intégrés dans l'historique de navigation, permettant le bouton "Retour" de naviguer entre états de filtres.
- **Persistance session** : Le rechargement de page conserve les filtres, contrairement à un state React local.

Le moteur de filtrage côté client (filteredProducts useMemo) applique les filtres de recherche et de prix sur les produits déjà chargés en mémoire. La filtrage par catégorie, lui, est délégué au backend (paramètre categorySlug dans la requête API) pour éviter de charger l'intégralité du catalogue non filtré. Cette hybridation API-filtrage/Client-filtrage représente le compromis optimal entre performance réseau et réactivité UI.

5.2 Transformation et normalisation des données

■ src/apps/clients/features/products/api/product.service.js

ProductService implémente la fonction de transformation transformProductData(), un pattern crucial dans les applications e-commerce dont les APIs évoluent fréquemment. Cette fonction normalise les données brutes du backend en un format stable et prédictible pour les composants, absorbant les variations de nommage (variantsPreview vs variants_preview vs variants) et les incohérences de types.

Cette normalisation est **localisée dans le Service Layer**, garantissant que les composants et les hooks ne dépendent jamais directement du format de l'API. Lorsque le backend modifie la structure d'une réponse, seule

la fonction transformProductData() nécessite une mise à jour, sans impact sur l'interface utilisateur.

getAll()	Récupération paginée avec filtres. Retourne un tableau normalisé.
getOne(idOrSlug)	Récupération d'un produit par ID ou slug SEO.
validateVariants(ids)	Vérification multi-variantes pour la validation du panier.
findFeatured()	Produits mis en avant pour la homepage (isFeatured=true, limit=4).
getMaxProductPrice()	Calcul du prix maximum pour initialiser les filtres de plage de prix.
generateSegments()	Génération dynamique des segments de prix (0-25%, 25-50%, etc.)

5.3 Composants de présentation produit

■ src/apps/clients/features/products/components/ProductCard.jsx

ProductCard est le composant de présentation fondamental de l'application. Ses interactions (changement de variante, ajout au panier, navigation vers le détail) sont entièrement orchestrées par useProductCardLogic, réduisant le composant à un rôle de rendu pur. Cette conception présente un avantage notable pour la testabilité : le comportement peut être testé dans le hook indépendamment du rendu.

La prop isLCP (Largest Contentful Paint) mérite une attention particulière. Elle implémente une stratégie d'optimisation Lighthouse avancée : la première carte du catalogue reçoit fetchPriority="high" et loading="eager", signalant au navigateur de traiter cette image avec la plus haute priorité réseau. Toutes les autres cartes conservent le comportement par défaut (loading="lazy"), différant leur chargement jusqu'à leur entrée dans le viewport.

■ src/apps/clients/features/products/components/ProductGallery.jsx

La galerie produit implémente un système de zoom interactif multi-surface — souris sur desktop, pinch-to-zoom sur mobile — entièrement géré par le hook useProductGallery. La séparation stricte entre la logique d'interaction (positions, niveaux de zoom, gestion des événements touch) et le rendu respecte exemplarirement le principe de séparation des préoccupations.

- **Desktop** : Zoom au clic avec suivi de la souris pour le déplacement de l'image zoomée. Boutons +/- pour contrôle précis.
- **Mobile** : Double-tap pour zoom, pinch pour agrandir/réduire. Hint gestuel disparaissant après 3s.
- **Accessibilité** : Zone interactive annotée avec role="img" et aria-label contextuel.
- **UX** : Skeleton loader pendant le chargement, fallback image en cas d'erreur.

5.4 Page Détail Produit

■ src/apps/clients/features/products/pages/ProductDetail.jsx

La page Détail Produit orchestre quatre sous-composants (ProductGallery, ProductInfo, ProductActions) via le hook useProductDetailLogic. Elle gère trois états de rendu distincts :

État Loading	Spinner centré avec SEO noIndex pour éviter l'indexation d'une page de chargement.
État Error / 404	Message d'erreur élégant avec bouton de retour. SEO noIndex.
État Success	Layout responsive mobile-first (colonnes empilées) et desktop (grille 2 colonnes).

La génération des métadonnées SEO est dynamique et contextuelle : le titre inclut le nom du produit, la description est construite depuis la description produit tronquée, et l'URL canonique est basée sur le slug unique du produit. Le schéma JSON-LD Product (Schema.org) inclut les informations d'offre, de disponibilité et de garantie pour les Rich Results Google.

■ src/apps/clients/features/products/hooks/useProductDetailLogic.js

Ce hook implémente un algorithme de sélection intelligente de variante en quatre niveaux de priorité décroissante :

- **Niveau 1** : Choix explicite de l'utilisateur (clic sur une variante).
- **Niveau 2** : Variante transmise par la navigation (deep link depuis le catalogue avec location.state.initialVariantId).
- **Niveau 3** : Première variante disponible en stock (sélection automatique intelligente).
- **Niveau 4** : Première variante du tableau (dernier recours si toutes épuisées).

Ce mécanisme de fallback garantit qu'un produit n'apparaît jamais sans variante sélectionnée, évitant les états d'UI incomplets qui seraient problématiques pour l'affichage du prix et du stock.

CHAPITRE 6

Checkout et Paiement — /features/checkout

Le tunnel de commande (Checkout) est le flux le plus critique de l'application en termes de fiabilité et de sécurité. Une erreur dans ce flux peut entraîner une perte directe de revenu et une dégradation significative de l'expérience utilisateur. L'architecture retenue privilégie la robustesse (stratégie de fallback, validation en double) et la transparence pour l'utilisateur.

6.1 Architecture du tunnel de commande

Le tunnel de commande suit un flux en quatre étapes séquentielles, orchestrées par useCheckout et rendues par CheckoutPage :

Étape	Action	Source de données
1. Saisie formulaire	Informations client + adresse	useState (formData)
2. Calcul livraison	Options de livraison selon pays	API /shipping/calculate ou fallback local
3. Calcul TVA	Taux selon pays de livraison	API /taxes/rates ou constante locale
4. Soumission	Création commande + session Stripe	POST /orders/checkout → POST /payments/create-session

6.2 Stratégie de fallback des frais de livraison

■ `src/apps/clients/features/checkout/hooks/useCheckout.js`

Un des patterns les plus sophistiqués de l'application réside dans la gestion des options de livraison. L'endpoint /shipping/calculate est protégé (authentification optionnelle mais préférable) et peut être inaccessible dans le contexte d'un checkout Guest en raison de l'absence de token.

useCheckout implémente une **stratégie de degradation gracieuse** : si l'API est inaccessible (erreur réseau, 401, 503), le hook bascule silencieusement vers un calcul local basé sur les constantes SHIPPING_RULES (définies dans shippingRules.js). Cette stratégie garantit que le tunnel de commande reste fonctionnel même en cas de défaillance partielle du backend.

- **Zones de livraison** : FRANCE (Standard, Express, Point Relais), EUROPE (Standard, Express), WORLD (Standard).
- **Calcul dynamique** : Coût = Base + (poids × tarif/kg). Livraison gratuite si sous-total ≥ seuil.
- **Localisation TVA** : Taux par défaut par pays (France: 20%, Allemagne: 19%, Italie: 22%).

6.3 Backup du panier et sécurité de la transaction

Avant la redirection vers Stripe, le hook exécute CartBackupService.backup(cart). Ce mécanisme de sauvegarde résout un problème de fiabilité critique : si l'utilisateur ferme son navigateur pendant le paiement ou si une erreur survient côté Stripe, le contenu du panier est préservé dans un emplacement localStorage séparé (*backup*). Au retour sur l'application, useCartRestore peut restaurer automatiquement ce backup.

L'identifiant de commande (orderId) et l'email client sont stockés dans sessionStorage (non localStorage) avant la redirection. Ce choix garantit que ces informations sont perdues à la fermeture du navigateur, réduisant la surface d'exposition des données de session, tout en étant disponibles au retour depuis Stripe (même onglet/session).

6.4 Pages de résultat de paiement

■ `src/apps/clients/features/checkout/pages/PaymentSuccess.jsx`

La page de succès implémente une logique de réconciliation post-paiement :

- **Persistence Guest** : Si l'utilisateur n'est pas authentifié, la commande est ajoutée au GuestOrderService pour permettre le suivi ultérieur.
- **Nettoyage** : Le panier est vidé (clearCart) et le backup supprimé (CartBackupService.clear()).
- **Invalidation du cache** : Les requêtes TanStack Query des produits sont invalidées (queryClient.invalidateQueries) pour forcer un rechargement des stocks à jour, reflétant l'achat effectué.
- **Hook usePaymentResult** : Vérifie la validité du paiement via l'orderId stocké en sessionStorage avant d'afficher la confirmation.

CHAPITRE 7

Commandes et Suivi — /features/orders

La gestion des commandes doit couvrir deux cas d'usage radicalement différents : l'utilisateur authentifié, dont les commandes sont persistées côté serveur et récupérables via l'API, et l'invité (Guest), dont les commandes sont stockées localement dans le navigateur pour permettre un suivi minimal sans obligation de création de compte.

7.1 GuestOrderService — Persistance locale robuste

■ `src/apps/clients/features/orders/api/GuestOrder.service.js`

GuestOrderService est un service "pur" (sans dépendances React) gérant le localStorage pour les commandes des utilisateurs non authentifiés. Son architecture intègre plusieurs mécanismes de robustesse :

Statuts persistables	Seuls PAID, PROCESSING, SHIPPED, DELIVERED, CANCELLED, REFUNDED sont persistés. PENDING est exclu pour éviter les fantômes à 0€ (le webhook Stripe n'a pas encore confirmé).
Expiration 30 jours	Nettoyage automatique des commandes de plus de 30 jours lors de la lecture. Aucune commande obsolète dans le storage.
Limite 10 commandes	Cap de 10 commandes stockées. Les plus récentes sont conservées en cas de débordement.
Déduplication	Ajout par orderNumber : une commande existante est remplacée (mise à jour statut) plutôt que dupliquée.
Cross-tab sync	Dispatch d'un CustomEvent "guestOrdersChanged" à chaque modification, permettant la synchronisation entre onglets.

La méthode syncWithClaimed(orderNumbers) implémente la réconciliation entre localStorage et serveur lors du login. Quand un utilisateur se connecte, le backend retourne les numéros de commandes guest réclamées. Ce service retire alors ces commandes du storage local, évitant qu'elles n'apparaissent dans la vue Guest alors qu'elles sont désormais liées au compte et accessibles via l'API.

7.2 OrderService — Communication API

■ `src/apps/clients/features/orders/api/Order.service.js`

OrderService gère la communication avec l'API pour les commandes. Il supporte les deux modes d'accès : authentifié (token Bearer) et guest (paramètre email en query string). La méthode getOrderDetails() normalise la structure de réponse du backend, qui peut présenter l'objet commande sous deux structures selon l'endpoint

(data.data.order ou data.data directement).

La méthode claimOrder() implémente le rattachement manuel d'une commande guest à un compte. Elle appelle l'API, puis synchronise immédiatement le localStorage via GuestOrderService.syncWithClaimed() pour maintenir la cohérence état serveur / état local. Sans cette synchronisation, l'interface afficherait une commande que le serveur refuserait de retourner en mode guest (la commande ayant maintenant un user_id non null côté serveur).

7.3 Hooks de gestion des commandes

■ src/apps/clients/features/orders/hooks/useOrders.js

useOrders gère le chargement et la pagination des commandes pour les utilisateurs authentifiés. Il implémente un pattern de protection conditionnel : si l'utilisateur n'est pas authentifié, aucune requête n'est déclenchée et l'état est immédiatement réinitialisé. La fonction fetchOrders est mémoïsée avec useCallback et dépend d' isAuthenticated, garantissant que le hook réagit correctement aux changements d'état d'authentification (connexion/déconnexion).

■ src/apps/clients/features/orders/hooks/useGuestOrders.js

useGuestOrders implémente une synchronisation réactive du localStorage. Contrairement à un simple useState, ce hook écoute deux types d'événements pour rester synchronisé :

- **CustomEvent "guestOrdersChanged"** : Déclenché par GuestOrderService._save() pour synchroniser les composants dans le même onglet.
- **Événement "storage"** : Déclenché par le navigateur pour synchroniser les changements provenant d'autres onglets (un achat dans un autre onglet met à jour la liste dans le premier).

CHAPITRE 8

Profil Utilisateur — /features/user

8.1 Architecture de la page Profil

■ `src/apps/clients/features/user/pages/Profile.jsx`

La page Profil est un composant **orchestrateur de haut niveau** qui adapte son interface selon l'état d'authentification. Elle affiche trois vues radicalement différentes selon le contexte :

Utilisateur authentifié	Onglets : Mes Commandes (OrderHistory), Informations personnelles (ProfileDetail), Sécurité (ChangePasswordForm).
Invité avec commandes	Espace Invité avec liste GuestOrdersList et CTA de conversion vers inscription.
Invité sans commandes	Formulaire de suivi de commande (GuestTrackingForm).

Le composant ProfileSidebar, extrait en sous-composant local, centralise la navigation entre les onglets. Cette extraction améliore la lisibilité du composant parent tout en permettant une évolution indépendante de la navigation (ajout d'un nouvel onglet sans modifier le corps de la page).

8.2 useProfile — Gestion du cycle de vie des données

■ `src/apps/clients/features/user/hooks/useProfile.js`

useProfile implémente trois patterns avancés de gestion d'état :

Optimistic Update (mise à jour optimiste)

Lors de la mise à jour du profil (updateProfile()), l'état local est mis à jour immédiatement (*optimistic update*) avant la confirmation du serveur. Si la requête échoue, le profil précédent est restauré via la variable previousProfile capturée avant la mutation (*rollback*). Ce pattern garantit une expérience perçue instantanée pour l'utilisateur, conforme aux standards UX des applications modernes.

Gestion granulaire des erreurs HTTP

Les erreurs API sont traitées avec une granularité fine selon leur code HTTP : 401 (session expirée), 400 (données invalides — message du backend affiché), 429 (rate limiting — message spécifique invitant à patienter). Cette approche évite le message d'erreur générique «Une erreur est survenue» qui ne fournit aucune information actionnable à l'utilisateur.

Protection contre les requêtes en double

Le flag isAuthenticated conditionne chaque requête. Si l'utilisateur se déconnecte pendant une requête en cours, le hook reçoit le changement via la dépendance du useEffect et réinitialise l'état, évitant l'affichage de données d'un autre utilisateur après un changement de session rapide.

CHAPITRE 9

Interface d'Administration — /apps/admin

L'interface d'administration constitue le back-office de la plateforme ECOM-WATCH. Elle est conçue comme une application distincte partageant les mêmes primitives techniques (composants, hooks, services) mais avec ses propres layouts, routes et patterns UX adaptés à un usage professionnel intensif. L'accès est conditionné par le rôle ADMIN via le RoleGuard.

9.1 Architecture de l'espace Admin

L'administration est organisée selon le même pattern **feature-first** que le storefront, avec sept modules métier indépendants :

Module	URL	Fonctionnalités
Dashboard	/admin	KPIs, graphique revenus, commandes récentes, alertes stock
Produits	/admin/products	CRUD produits, gestion variantes, images
Stock	/admin/inventory	Consultation stock, ajustements, alertes réassort
Catégories	/admin/categories	CRUD catégories, slugs URL
Commandes	/admin/orders	Consultation, changement de statut, filtrage
Clients	/admin/users	Consultation, modification rôle/statut
Promotions	/admin/promotions	CRUD promotions, activation/désactivation

9.2 Layout Admin — AdminLayout, AdminSidebar, AdminNavbar

■ `src/apps/admin/layout/AdminLayout.jsx`

Le layout Admin adopte une structure en deux colonnes fixes : sidebar de navigation (264px) et zone de contenu principale (flex-1). Le scroll est isolé dans la zone principale (overflow-y-auto), permettant à la sidebar et à la navbar de rester fixes pendant le défilement du contenu. Sur mobile, la sidebar se transforme en tiroir (*drawer*) avec overlay semi-transparent, évitant de sacrifier l'espace écran sur les petits dispositifs.

■ `src/apps/admin/layout/AdminNavbar.jsx`

La navbar Admin intègre deux fonctionnalités distinctives par rapport à la navbar storefront :

- **Dark/Light Mode Toggle** : Bouton bascule entre thème clair et sombre via useTheme(). Le thème est persisté en localStorage et appliqué via la classe dark sur le html root, déclenchant les variantes dark: de Tailwind CSS sur l'ensemble de l'application Admin.
- **Identification administrateur** : Affichage du prénom et d'un avatar initiales coloré inversé selon le thème (fond noir/texte blanc en light, fond blanc/texte noir en dark), renforçant la conscience de rôle de l'opérateur.

9.3 Design System Admin — Composants Partagés

■ src/apps/admin/features/shared/AdminTable.jsx

AdminTable est le composant générique fondateur de l'interface Admin. Il implémente le pattern **Render Props** via la prop renderRow : le composant gère la structure du tableau, les états vides, et la pagination, tandis que chaque module fournit son propre rendu de ligne (ProductTableRow, OrderTableRow, etc.). Cette conception permet une personnalisation maximale avec une réutilisation maximale.

La pagination est intégrée directement dans le composant via la prop paginationData. Ce choix architectural garantit que chaque tableau peut être paginé de manière indépendante sans duplication du code de pagination dans chaque page. Les boutons de navigation sont désactivés conditionnellement (disabled) aux extrémités de la pagination.

Le composant supporte nativement le dark mode via les classes Tailwind dark: sur chaque élément de style (fond, bordures, textes), garantissant un rendu cohérent quelle que soit la préférence de thème de l'opérateur.

■ src/apps/admin/features/shared/AdminDrawer.jsx

AdminDrawer est le tiroir générique utilisé par tous les formulaires d'édition Admin. Son implémentation gère rigoureusement le cycle de vie d'animation :

- **shouldRender** : Contrôle le montage/démontage du DOM. Le tiroir reste monté pendant la transition de fermeture pour que l'animation de sortie reste visible.
- **isAnimated** : Contrôle les classes CSS d'animation. Basculé via requestAnimationFrame pour s'assurer que le navigateur a commis le rendu initial avant de déclencher la transition CSS.
- **overflow lock** : Blocage du scroll body lors de l'ouverture, restauration lors de la fermeture (via le return du useEffect).

Ce pattern de gestion du cycle de vie des tiroirs (mount → animate-in → animate-out → unmount) est identique à celui utilisé dans le CartDrawer du storefront, garantissant une cohérence comportementale à travers toute l'application.

9.4 Dashboard — Agrégation de données

■ src/apps/admin/features/dashboard/pages/Dashboard.jsx

Le Dashboard Admin illustre le pattern **Orchestrator Component** à son expression la plus pure : le composant lui-même ne contient aucune logique de fetching, aucun état, aucun effet. Il délègue intégralement à

useDashboardLogic et passe les données aux composants de présentation (KpiSection, RevenueChart, RecentOrdersTable, QuickLinksSection).

■ src/apps/admin/features/dashboard/hooks/useDashboardLogic.js

useDashboardLogic implémente le pattern **Promise.all Parallel Fetching** : les quatre requêtes (stats KPI, historique ventes, commandes récentes, alertes stock) sont lancées simultanément pour minimiser le temps de chargement total. Sans cette parallélisation, le chargement séquentiel représenterait 4× la latence d'une seule requête.

Chaque requête individuelle est encapsulée dans un .catch() qui retourne une valeur par défaut, garantissant qu'une défaillance d'un endpoint (ex: /inventory/alerts indisponible) n'empêche pas l'affichage des autres sections. La transformation des données pour Recharts (format {name: string, total: number}) est centralisée dans un useMemo pour éviter les recomputations inutiles.

9.5 Modules CRUD — Pattern Cohérent

Chaque module CRUD de l'administration (Produits, Commandes, Catégories, Utilisateurs, Promotions, Inventaire) suit un pattern architectural identique, garantissant la cohérence UX et facilitant la montée en compétence des nouveaux membres de l'équipe :

Page (Orchestrateur)	Compose AdminTable + TableToolbar + DrawerFormulaire + ConfirmDialog. Gère l'état local des modales.
Hook (useAdmin*)	Gère fetching, pagination, recherche (debounce), CRUD actions et confirmation de suppression.
Service (Admin*Service)	Appels API CRUD avec paramètres de filtrage et pagination.
TableRow	Composant de rendu d'une ligne — affiché via renderRow de AdminTable.
FormDrawer	Formulaire d'édition/création dans AdminDrawer. Consomme un hook de logique de formulaire.

■ src/apps/admin/features/products/pages/productsAdmin.jsx

La page Produits Admin illustre la gestion de la complexité des formulaires imbriqués : elle orchestre trois tiroirs distincts (ProductFormDrawer pour les produits, VariantFormDrawer pour l'ajout de variantes, VariantEditDrawer pour la modification). L'état de chaque tiroir est géré localement dans la page, maintenant une isolation claire entre les différents flux d'édition.

Le service AdminProductService implémente la normalisation avancée transformAdminProduct() qui couvre quatre formats distincts de retour de la catégorie selon l'endpoint appelé (liste vs détail), garantissant que la propriété displayCategory est toujours disponible dans les composants UI quel que soit l'endpoint source.

CHAPITRE 10

SEO et Composants Partagés — /shared

10.1 Stratégie SEO — react-helmet-async

■ src/shared/SEO/SEOHead.jsx

SEOHead est le composant SEO universel de l'application. Basé sur react-helmet-async (version async-safe du Helmet originel, compatible avec le rendu concurrent de React 18+), il injecte dynamiquement l'ensemble des balises <head> selon la page affichée.

Pour une SPA React, le SEO dynamique est un défi particulier : les robots d'indexation exécutent généralement le JavaScript, mais avec des délais et des limitations. react-helmet-async garantit que les balises méta sont correctement définies lors du rendu initial côté client, maximisant les chances d'indexation correcte des pages produit et catalogue.

Title	Balise <title> pour la barre du navigateur et les SERPs.
Description	Meta description — jusqu'à 160 caractères pour l'extrait dans les SERPs.
Canonical	Évite le contenu dupliqué pour les pages avec paramètres de filtrage.
noIndex	Appliqué sur les pages transactionnelles (Checkout, Profil, Success) et les pages filtrées (prix)
Open Graph	og:title, og:description, og:image pour le partage social (Facebook, LinkedIn).
Twitter Card	summary_large_image pour un rendu riche sur Twitter/X.
JSON-LD	Structured Data Schema.org pour les Rich Results Google.

■ src/shared/SEO/seoSchemas.js

Ce fichier centralise les générateurs de schémas JSON-LD (JavaScript Object Notation for Linked Data) conformes au vocabulaire Schema.org. Ces schémas permettent aux moteurs de recherche de comprendre la sémantique du contenu et d'afficher des Rich Results (résultats enrichis) dans les SERPs.

buildHomeSchema()	ItemList des produits featured — améliore le taux de clic en SERP.
buildCatalogueSchema()	CollectionPage avec BreadcrumbList — navigation SERP enrichie.
buildProductSchema()	Product complet avec Offer/AggregateOffer, garantie 2 ans, politique de retour.

buildOrderConfirmationSchema()	Order — suivi des conversions dans Google Search Console.

Le schéma produit (buildProductSchema()) mérite une attention particulière : il distingue automatiquement le cas d'une variante unique (Offer) du cas multi-variantes (AggregateOffer avec lowPrice et highPrice). Il inclut également les détails de livraison (OfferShippingDetails) et la politique de retour (MerchantReturnPolicy), deux champs valorisés par Google Shopping.

10.2 Composants UI Réutilisables

■ src/shared/UI/ConfirmDialog.jsx

ConfirmDialog implémente la modale de confirmation standardisée de l'application. La prop isDangerous contrôle le style du bouton de confirmation (rouge pour les suppressions irréversibles, neutre pour les actions réversibles), appliquant une convention UX cohérente à travers tous les modules Admin. Le composant utilise une gestion de focus accessible et un fond backdrop-blur pour la perception de profondeur.

■ src/shared/UI/ProductSkeleton.jsx

ProductSkeleton implémente le pattern **Skeleton Loading** : un placeholder animé représentant la structure de la future carte produit. Cette technique améliore significativement la perception de performance comparativement aux spinners : l'utilisateur voit immédiatement la structure de la page et comprend que du contenu va apparaître, réduisant la perception d'attente. L'animation shimmer est réalisée via une transition CSS background-position native, évitant l'usage de bibliothèques JavaScript d'animation.

■ src/shared/UI/EmptyState.jsx

EmptyState gère l'état vide du catalogue avec une illustration élégante de montre (SVG inline) et une communication claire : «Aucune montre trouvée». Si des filtres actifs sont responsables de l'état vide, un bouton «Réinitialiser les filtres» est affiché conditionnellement via la prop onClearFilters. Cette contextualisation évite l'état vide inexpliqué qui génère de la frustration utilisateur.

CHAPITRE 11

Routage et Contrôle d'Accès

11.1 Architecture du routage

■ src/app/router.jsx

L'application utilise **React Router v7** avec l'API createBrowserRouter. Le routeur global assemble deux branches distinctes : clientsRoutes (storefront public) et adminRoutes (back-office protégé). Cette séparation par fichier de routes permet une évolution indépendante de chaque périmètre.

■ src/apps/clients/routes.jsx

■ src/apps/admin/routes.jsx

Les deux modules de routes adoptent une structure identique : un layout racine wrappé dans Suspense, avec les pages chargées en **lazy loading** via React.lazy().

11.2 Lazy Loading et Code Splitting

L'ensemble des composants de page sont chargés en différé (lazy loading). Cette stratégie de **code splitting** garantit que le bundle JavaScript initial est minimal, accélérant le First Contentful Paint (FCP) et le Time to Interactive (TTI). Vite gère automatiquement la création de chunks séparés pour chaque page lazy-loadée.

Le fallback Suspense (PageLoader pour le storefront, AdminLoader pour l'administration) affiche un indicateur minimal pendant le chargement du chunk. Ces deux loaders sont distincts pour respecter le contexte visuel de chaque domaine de l'application.

11.3 Protection des routes

La stratégie de protection des routes est implémentée à deux niveaux :

GuestGuard	Redirige les utilisateurs authentifiés hors des routes Login/Register. Évite l'accès aux formulaires d'auth depuis un compte connecté.
RoleGuard	Vérifie le rôle ADMIN pour toute la section /admin. Double vérification : isAuthenticated ET hasRole.
Redirect to referrer	useAuthForm stocke la location courante et redirige vers elle post-login (via location.state.from.pathname).

Le routeur Admin encapsule son layout dans le RoleGuard au niveau de la route parente, protégeant ainsi l'intégralité des routes enfants (/admin/products, /admin/orders, etc.) en un seul point de contrôle, conformément au principe DRY (Don't Repeat Yourself).

CHAPITRE 12

Performance et Qualité

12.1 Stratégies de memoization

L'application fait un usage extensif des primitives de memoization de React pour prévenir les re-rendus inutiles. Les règles appliquées systématiquement sont les suivantes :

Primitive	Usage	Exemple dans le projet
useCallback	Mémoïser les fonctions passées en prop ou dépendances d'effet	fetchOrders, handleCategoryChange, validateCart
useMemo	Mémoïser les valeurs calculées coûteuses	filteredProducts, totalPrice, formattedChartData
useRef	Référence mutable sans re-rendu	cartRef dans useCartLogic (évite la dépendance cyclique)
Initialisation lazy	Éviter le re-calcul de l'état initial	useState(() => JSON.parse(localStorage...))

12.2 Optimisation des images (LCP)

L'application implémente une stratégie d'optimisation des images alignée sur les métriques Core Web Vitals de Google, en particulier le **Largest Contentful Paint (LCP)** :

- **Format WebP** : L'image hero de la homepage (featured.webp) utilise le format WebP, offrant une compression supérieure de 25-35% au JPEG à qualité équivalente.
- **fetchPriority="high"** : La première carte produit (prop isLCP=true) et l'image hero reçoivent la priority maximale dans la file réseau du navigateur.
- **loading="eager"** : Désactive le lazy-loading pour les images critiques du viewport initial, garantissant leur découverte immédiate par le navigateur.
- **Dimensions explicites** : Les attributs width et height sur les images hero permettent au navigateur de réserver l'espace avant le chargement, éliminant le Cumulative Layout Shift (CLS).
- **loading="lazy"** : Appliqué sur toutes les images hors-viewport initial, différant leur chargement et économisant la bande passante.

12.3 Accessibilité (WCAG 2.1)

L'application respecte les directives WCAG 2.1 niveau AA sur plusieurs axes :

- **Contraste des couleurs** : Correction identifiée et appliquée dans le footer — passage de text-gray-500 (ratio 3.8:1, échec AA) à text-gray-300 (ratio 10:1, conforme).
- **Éléments interactifs** : Tous les boutons disposent d'aria-label explicites. Les icônes décoratives sont annotées aria-hidden="true".

- **Formulaires** : Association label/id explicite sur le sélecteur de taille de la ProductCard (htmlFor={`size-select-${product.id}`}), évitant les doublons d'ID en grille.
- **Hiérarchie des titres** : Correction du footer — remplacement des balises <h4> orphelines par des <p> stylisés, évitant les avertissements Lighthouse «Heading order».
- **États ARIA** : aria-pressed sur les boutons de sélection de variante, aria-live sur les compteurs de quantité.
- **Navigation clavier** : Tous les éléments interactifs sont accessibles au clavier via tabulation.

12.4 CI/CD et déploiement

L'application est déployée sur **Vercel**, avec une configuration de routage définie dans vercel.json. Ce fichier configure la réécriture de toutes les routes vers index.html (*catch-all rewrite*), nécessaire pour le fonctionnement d'une SPA avec des URLs profondes (ex: /product/rolex-submariner rechargé directement).

Le dossier .github/workflows/ contient deux pipelines GitHub Actions :

ci.yml (Continuous Integration)	Exécuté sur chaque Pull Request. Lint (eslint), vérification des types, build de validation.
cd.yml (Continuous Deployment)	Exécuté sur merge vers main. Déploiement automatique sur Vercel en production.

La configuration .npmrc fixe les versions de dépendances pour garantir la reproductibilité des builds entre développement local et CI/CD. Le script generateSitemap.mjs génère automatiquement le fichier sitemap.xml avec les URLs des produits, optimisant l'indexation par les moteurs de recherche.

12.5 Synthèse des décisions architecturales

Le tableau suivant synthétise les principales décisions architecturales et leurs justifications :

Décision	Alternative rejetée	Justification
Token en mémoire (Closure)	localStorage	Immunité aux attaques XSS
Zustand vs Redux	Redux Toolkit	Légèreté, absence de boilerplate, sélection fine
URL comme état des filtres	useState local	Deep linking, SEO, historique navigateur
Pattern CHS	Logique dans les composants	Testabilité, réutilisabilité, maintenabilité
CSR vs SSR	Next.js SSR	Complexité infra vs richesse interactive
Feature-first	Type-first	Scalabilité, couplage réduit, cohésion maximale
Tailwind CSS v4	CSS Modules / Styled Components	Utilities atomiques, dark mode, purging automatique
TanStack Query	Context API pour le cache	Gestion native stale/fresh, invalidation, devtools
Lazy Loading pages	Bundle monolithique	TTI amélioré, bundle initial minimal

Promise.all Dashboard	Requêtes séquentielles	Temps de chargement réduit de N× la latence

Tableau 12.1 — Matrice des décisions architecturales

ANNEXES

Référence Technique et Glossaire

A. Glossaire technique

CHS Pattern
Component-Hook-Service. Architecture de séparation des préoccupations en trois couches.

Closure
En JavaScript, une fonction qui capture les variables de son environnement lexical parent.

CSR
Client-Side Rendering. Le rendu HTML est effectué par JavaScript dans le navigateur.

Custom Hook
Fonction JavaScript préfixée use* encapsulant de la logique React réutilisable.

Deep Linking
Capacité d'accéder directement à une URL profonde de l'application avec l'état correspondant.

Dumb Component
Composant React limité à la présentation, sans logique métier propre.

Feature-First
Organisation des fichiers par fonctionnalité métier plutôt que par type technique.

HttpOnly Cookie
Cookie inaccessible à JavaScript, géré exclusivement par le navigateur.

Hydratation
Processus d'attachement des event handlers React sur un DOM pré-rendu côté serveur.

JWT
JSON Web Token. Standard d'authentification basé sur un token signé cryptographiquement.

LCP
Largest Contentful Paint. Métrique Core Web Vitals mesurant le rendu du plus grand élément.

Memoization
Technique de cache calculant un résultat une seule fois et le réutilisant si les entrées n'ont pas changé.

RBAC
Role-Based Access Control. Contrôle d'accès basé sur les rôles utilisateur.

Rich Results
Résultats enrichis dans les SERPs Google grâce aux données structurées JSON-LD.

SoC
Separation of Concerns. Principe architectural séparant les responsabilités.

SPA
Single Page Application. Application JavaScript chargée une fois, naviguant sans rechargement.

Tree Shaking
Élimination du code mort (exports non utilisés) lors du build de production.

TTI

Time to Interactive. Délai avant que la page soit pleinement interactive.

XSS
Cross-Site Scripting. Injection de code JavaScript malveillant dans une page web.

Optimistic Update
Mise à jour de l'UI avant confirmation serveur, avec rollback en cas d'erreur.

B. Inventaire des fichiers documentés

Fichier	Chapitre	Pattern
App.jsx	1	Provider Tree Root
main.jsx	1	Entry Point
router.jsx	11	Route Configuration
axios.config.js	2	Closure Security Pattern
queryClient.js	2	Cache Configuration
env.js / app.js	2	Configuration Module
orderStatus.js	2	Enum Freeze Pattern
logger.js	2	Environment-aware Logger
AuthContext.jsx	3	Provider Pattern
useAuthStore.js	3	Zustand Store
useAuth.js	3	Orchestration Hook
GuestGuard.jsx / RoleGuard.jsx	3/11	Navigation Guard
auth.schema.js	3	Zod Validation Schema
CartProvider.jsx	4	Context + Key Isolation
useCartLogic.js	4	Complex Business Hook
CartDrawer & sub-components	4	Dumb Presentation Components
useCatalogueLogic.js	5	URL-as-State Pattern
product.service.js	5	Service + Normalizer
ProductDetail.jsx	5	Orchestrator Page
ProductCard.jsx	5	Dumb Component + LCP Optimization
ProductGallery.jsx	5	Interaction Delegation Pattern
useCheckout.js	6	Fallback Strategy Hook
checkout.service.js	6	Service Layer
PaymentSuccess.jsx	6	Post-Transaction Reconciliation
GuestOrderService.js	7	Robust LocalStorage Service
OrderService.js	7	API Service + Sync

useProfile.js	8	Optimistic Update Pattern
AdminLayout/Sidebar/Navbar	9	Shell Layout
AdminTable.jsx	9	Render Props + Generic Component
AdminDrawer.jsx	9	Animation Lifecycle Management
useDashboardLogic.js	9	Parallel Fetching (Promise.all)
SEOHead.jsx	10	Dynamic Head Management
seoSchemas.js	10	JSON-LD Generators
ConfirmDialog.jsx	10	Reusable Modal
ProductSkeleton.jsx	10	Skeleton Loading Pattern
clientsRoutes.jsx / adminRoutes.jsx	11	Feature Routes with Guards

— Fin de la documentation —

ECOM-WATCH · Documentation Technique Frontend v1.0 · Juin 2025
Architecture React — Confidentiel — Usage interne