BOOKAUTO

DOCUMENT ANALYSE TECHNIQUE COMPLÈTE

Architecture · Stack · Code · Sécurité · Scalabilité · DevOps · Recommandations

---

Document 2 / 2 · Avril 2026 · Analyse digne d'un audit technique investisseur

1. Architecture globale

1.1 Type d'application

BookAuto est une application web full-stack de type SPA (Single Page Application) côté frontend, couplée à une API REST côté backend. D'après le README.md, l'architecture est la suivante :

- 🖥️ **Frontend :** Frontend : HTML5, CSS3, JavaScript vanilla (pas de framework React/Vue/Angular selon le README). Déployé sur Vercel (anciennement décrit comme Netlify dans le README).
- ⚙️ **Backend :** Backend : Node.js + Express.js exposant une API REST. Déployé sur Render (render.yaml présent dans le repo).
- 🗄️ **Data :** Base de données : JSON local (users.json) en phase actuelle, avec une migration MongoDB planifiée selon le README. La mention "MongoDB" dans la stack est donc une intention, pas une réalité déployée à ce stade.
- 🐬 **DevOps :** Conteneurisation : Docker + Docker Compose pour le développement local (docker-compose.yml présent). Nginx comme serveur web dans le conteneur.

⚠ **DÉCALAGE STACK ANNONCÉE / RÉALITÉ TECHNIQUE**

→ MongoDB : annoncée dans la stack, mais le README indique clairement "données stockées en JSON local",
migration MongoDB marquée comme "future évolution". MongoDB N'EST PAS en production.

→ Stripe : annoncé dans la stack, mais aucune route API de paiement n'est visible dans le README.
"Je ne peux pas confirmer l'intégration Stripe active sans accès au code source."

→ OpenStreetMap : annoncé dans la stack. Le README mentionne la géolocalisation comme fonctionnalité.
"Je ne peux pas confirmer l'implémentation concrète sans accès au code front."

1.2 Structure du projet

Structure confirmée par le repository GitHub (58 commits, branche main) :

Élément	Description	Statut
bookauto_/backend/	Dossier backend Node.js/Express	✅ Confirmé GitHub
bookauto_/frontend/	Dossier frontend HTML/CSS/JS	✅ Confirmé GitHub
docker-compose.yml	Orchestration Docker	✅ Confirmé GitHub
render.yaml	Configuration déploiement Render	✅ Confirmé GitHub
.eslintrc.js	Configuration ESLint (linting JS)	✅ Confirmé GitHub
.prettierrc	Configuration Prettier (formatage)	✅ Confirmé GitHub
package.json / package-lock.json	Gestion dépendances npm	✅ Confirmé GitHub
README.md	Documentation projet	✅ Confirmé GitHub

data/users.json	Données de test locales (JSON)	✅ Confirmé README (JSON, pas MongoDB)

---

2. Analyse de la stack technique — Composant par composant

2.1 MongoDB — Base de données NoSQL documentaire

Rôle prévu : stockage des données utilisateurs, professionnels, réservations et agendas.

- ✅ **Avantage :** Adapté aux données hétérogènes (profils pros avec différents services et disponibilités). Le schéma flexible est un avantage pour un MVP évolutif.
- ✅ **Écosystème :** Mongoose (ORM pour MongoDB/Node.js) est l'outil standard — probable dans la stack même si non confirmé dans le README.
- ⚠️ **Limite actuelle :** Pas encore déployé selon le README. Les données sont en JSON local (users.json). Le passage à MongoDB Atlas (cloud) est recommandé.
- ⚠️ **Limite future :** En cas de scaling, MongoDB n'est pas optimal pour les requêtes relationnelles complexes (ex. matching artisan/créneau géolocalisé avec disponibilités). PostgreSQL avec PostGIS serait plus performant pour la géolocalisation à grande échelle.

Recommandation MongoDB

→ Migrer vers MongoDB Atlas (tier gratuit disponible) dès la V1 commerciale.
→ Définir des schémas Mongoose stricts avec validation des champs obligatoires.
→ Prévoir des index sur les champs de recherche (ville, type_service, disponibilités).
→ Évaluer PostgreSQL + PostGIS si les requêtes géospatiales deviennent critiques.

2.2 Node.js + Express.js — Serveur applicatif

Rôle : serveur HTTP exposant l'API REST, logique métier (réservations, agenda, authentification).

- ✅ **Avantage :** Stack très répandue pour les APIs REST légères, documentation abondante, large écosystème npm.
- ✅ **Version :** Node.js ≥ 18 requis selon le README — bonne pratique, LTS stable.
- ⚠️ **Risque :** Express.js est un micro-framework : pas de structure imposée, risque de code "spaghetti" si l'architecture n'est pas rigoureusement organisée en layers (routes → controllers → services → models).
- ⚠️ **Évolution :** Pour une marketplace en production, des alternatives comme NestJS (TypeScript, DI, décoration) offriraient une meilleure maintenabilité long terme.

- ⚠️ **Pas de TypeScript :** L'absence de TypeScript dans le projet actuel (100 % JavaScript selon GitHub) est un risque de qualité pour les équipes futures.

Recommandation Node.js/Express

→ Implémenter une architecture en couches stricte : routes → controllers → services → repositories.
→ Migrer progressivement vers TypeScript pour la maintenabilité.
→ Ajouter un middleware de validation des entrées (Joi, Zod) sur toutes les routes API.
→ Implémenter des tests unitaires (Jest) et des tests d'intégration (Supertest).

2.3 Docker + Docker Compose — Conteneurisation

Rôle : isolation de l'environnement de développement, portabilité, base pour le déploiement.

- ✅ **Avantage :** docker-compose.yml présent et fonctionnel — bonne pratique dès le départ.
- ✅ **Avantage :** Permet de reproduire l'environnement de développement identiquement sur toutes les machines (résout le "ça marche sur ma machine").
- ⚠️ **Configuration basique :** Le docker-compose.yml du README montre une configuration basique (port 3000, volume mount) sans multi-stage build ni distinction dev/prod. En production, les volumes de code source ne doivent pas être montés.
- ⚠️ **Registry :** Pas de Docker Hub ou registry privé mentionné — "Je ne peux pas confirmer la stratégie de registry d'images."

Recommandation Docker

→ Séparer les configurations docker-compose.dev.yml et docker-compose.prod.yml.
→ Utiliser des multi-stage builds pour réduire la taille des images de production.
→ Ajouter un healthcheck dans les services Docker.
→ Envisager Docker Swarm ou Kubernetes à long terme pour la haute disponibilité.

2.4 Stripe — Paiement en ligne

Rôle prévu : encaissement des paiements des particuliers, reversement aux professionnels.

- ✅ **Choix excellent :** Stripe est la référence mondiale des paiements en ligne pour les marketplaces. Stripe Connect permet de gérer les flux financiers bifaces (plateforme → artisan) avec split automatique.
- ✅ **Stripe Connect :** Stripe Connect est précisément conçu pour les marketplaces de services — c'est le bon produit Stripe pour BookAuto.
- ⚠️ **Statut :** Je ne peux pas confirmer l'intégration Stripe active dans le code actuel. Le README ne mentionne aucune route de paiement dans le tableau d'API. Cette fonctionnalité est probablement planifiée mais non implémentée.

- ⚠️ **Complexité :** L'intégration Stripe Connect nécessite une vérification KYC des professionnels (pièce d'identité, IBAN) — cela renforce également la confiance sur la plateforme.

Recommandation Stripe

→ Implémenter Stripe Connect (Express Accounts pour les artisans).
→ Utiliser les Stripe Webhooks pour gérer les événements asynchrones (paiement confirmé, remboursement).
→ Ne jamais stocker de données de carte bancaire — Stripe PCI DSS compliant gère cela nativement.
→ Prévoir la gestion des remboursements (annulation, litige) dès la V1.
→ Tester avec les clés de test Stripe avant tout déploiement production.

2.5 OpenStreetMap — Géolocalisation

Rôle prévu : affichage des professionnels sur une carte, recherche par zone géographique.

- ✅ **Avantage :** Open source, gratuit, sans les limitations et coûts de l'API Google Maps. Bonne décision pour un MVP sans budget.
- ✅ **Écosystème :** Leaflet.js (librairie JS front pour OSM) est la combinaison standard — probablement utilisée mais "Je ne peux pas confirmer l'implémentation concrète sans accès au code front."
- ⚠️ **Limite :** Pour le géocodage (adresse → coordonnées), l'API Nominatim (OSM) est gratuite mais a des limites de taux (1 requête/seconde). En production avec du trafic, cela peut poser problème.
- ⚠️ **Alternative payante :** Google Maps Platform (Maps + Places + Geocoding) offre une meilleure précision et des SLA garantis, mais coûte entre 0,005 $ et 0,017 $ par requête (premier 28 500 requêtes/mois gratuits).

Recommandation OpenStreetMap

→ Maintenir OSM/Leaflet pour la phase MVP (coût zéro).
→ Implémenter un cache des résultats de géocodage pour réduire les appels à Nominatim.
→ Prévoir la migration vers Google Maps Platform ou Mapbox si le trafic dépasse 10 000 requêtes/jour.
→ Stocker les coordonnées GPS des artisans en base au moment de l'inscription pour éviter le géocodage à chaque recherche.

2.6 Vercel — Hébergement frontend

Rôle : déploiement et hébergement du frontend statique (HTML/CSS/JS).

- ✅ **Avantage :** CDN mondial, déploiement automatique depuis GitHub, HTTPS natif, tier gratuit généreux. Excellent choix pour un frontend statique.
- ⚠️ **Sous-utilisation :** Vercel est optimisé pour les frameworks JS (Next.js, React). Pour du HTML/JS vanilla, toutes les fonctionnalités avancées ne sont pas exploitées.
- ✅ **Déployé :** Le site bookauto.vercel.app est accessible — le déploiement frontend est effectif.

2.7 Render — Hébergement backend

Rôle : hébergement de l'API Node.js/Express en production.

- ✅ **Avantage :** Render est un PaaS (Platform as a Service) qui simplifie le déploiement de services Node.js. Le render.yaml dans le repo confirme la configuration.
- ⚠️ **Tier gratuit :** Le tier gratuit de Render met les services en veille après 15 minutes d'inactivité — "cold start" de ~30 secondes pour la première requête. Inacceptable pour un service de réservation d'urgence.
- ⚠️ **Coût :** Le tier payant (7 $/mois) supprime la mise en veille — à prévoir dès que le service entre en phase pilote.

Recommandation Render

→ Passer au tier payant Render (7 $/mois) dès le pilote pour éviter le cold start.
→ Configurer les variables d'environnement (secrets) dans Render, jamais en dur dans le code.
→ Activer les health checks Render pour monitoring automatique.
→ Évaluer Railway.app ou Fly.io comme alternatives similaires si les coûts augmentent.

---

3. Analyse du code source

3.1 Ce qui est confirmé

- 58 commits sur la branche main — activité de développement réelle mais modérée.
- 1 Pull Request ouverte — signe d'une pratique de workflow Git basique.
- **Qualité code :** Présence de .eslintrc.js et .prettierrc — bonnes pratiques de qualité de code (linting et formatage).
- Architecture en dossiers backend/ et frontend/ — séparation correcte des préoccupations à ce niveau.
- 99,6 % JavaScript, 0,4 % autre — projet 100 % JavaScript, pas de TypeScript.
- package.json à la racine ET dans backend/ — structure monorepo simple.

3.2 Ce qui ne peut pas être confirmé

🔍 JE NE PEUX PAS CONFIRMER ces informations sans accès au code source

→ La qualité interne du code (nommage, commentaires, modularité, gestion des erreurs)
→ L'implémentation de Stripe (aucune route de paiement visible dans les APIs listées du README)
→ L'implémentation d'OpenStreetMap côté frontend
→ La présence de tests automatisés (unitaires, intégration, E2E)
→ La gestion des erreurs (try/catch, middleware d'erreur Express)
→ La présence de variables d'environnement (.env, dotenv)
→ La sécurité des routes (middleware d'authentification JWT ou session)
→ La validation des données entrantes (middleware de validation)

3.3 Routes API confirmées (README)

Méthode	Route	Description	Évaluation
GET	/api/pros	Liste tous les professionnels	✅ Basique — manque filtres (ville, service, dispo)
GET	/api/pros/:id	Infos d'un professionnel	✅ Standard REST
POST	/api/reservations	Crée une réservation	⚠️ Sans paiement intégré visible
GET	/api/reservations/:userId	Réservations d'un user	✅ Standard
POST	/api/login	Authentification	⚠️ Mécanisme d'auth non précisé (JWT? Session?)

Observation critique : l'API exposée est minimale pour un MVP. Manquent notamment : PUT/DELETE sur les réservations (annulation), routes agenda professionnel, routes de recherche géolocalisée, routes de paiement/Stripe, routes de notation/avis.

---

4. Fonctionnalités — État des lieux

Fonctionnalité	Statut	Commentaire
Inscription particulier	✅ Implémenté (README)	Création compte, login
Inscription professionnel	✅ Implémenté (README)	Profil entreprise, prestations, disponibilités
Recherche par métier/zone	✅ Partiel (README)	GET /api/pros — filtres avancés à confirmer
Réservation créneau	✅ Implémenté (README)	POST /api/reservations
Agenda intégré	✅ Mentionné (README)	Paramétrage disponibilités côté pro
Géolocalisation (OSM)	⚠️ Annoncé	Je ne peux pas confirmer l'implémentation concrète

Paiement Stripe	⚠️ Annoncé non confirmé	Aucune route API de paiement dans le README
Système d'avis/notations	❌ Non mentionné	Fonctionnalité clé absente pour la confiance
Vérification artisans (Kbis)	❌ Non mentionné	Manque critique pour la crédibilité
Notifications (email/SMS)	❌ Non mentionné	Indispensable pour les confirmations de réservation
Tableau de bord analytics	⚠️ Prometheus/Grafana	Mentionné comme futur via Mediaschool
Application mobile	❌ Non prévu	Site responsive uniquement (probable)

---

5. Sécurité

5.1 Authentification

Le README mentionne la route POST /api/login et "Objectif futur : authentification OAuth2 pour les pros".

- ⚠️ **Mécanisme inconnu :** Le mécanisme d'authentification actuel n'est pas précisé. S'il s'agit de sessions JWT, la sécurisation des tokens (expiration, refresh, blacklist) est critique. "Je ne peux pas confirmer le mécanisme d'authentification sans accès au code."
- ⚠️ **Hashage passwords :** Aucune mention de hashage des mots de passe (bcrypt) dans le README — doit être implémenté.
- ✅ **Direction correcte :** L'OAuth2 mentionné comme "futur" est la bonne direction (connexion Google/Apple pour les particuliers, simplification friction d'inscription).

5.2 Paiements

- ✅ **Stripe PCI :** Si Stripe est utilisé correctement via son SDK, la sécurité PCI DSS est gérée par Stripe nativement.
- ⚠️ **Règle absolue :** La plateforme ne doit JAMAIS stocker de numéros de carte. Stripe Tokens/Payment Intents garantissent cela.

5.3 Données et RGPD

Le README mentionne : "Connexions HTTPS (Netlify/Nginx), données limitées au nécessaire, sauvegarde chiffrée".

- ✅ **HTTPS :** HTTPS est actif sur Vercel nativement — confirmation visuelle bookauto.vercel.app.
- ⚠️ **RGPD :** Une politique de confidentialité et des mentions légales doivent être ajoutées avant tout déploiement public (obligation RGPD).

- ⚠️ **Données en clair :** Les données JSON locales (users.json) ne doivent PAS être dans le repository Git — risque d'exposition de données personnelles. "Je ne peux pas confirmer si le .gitignore exclut users.json."

🚨 RISQUES SÉCURITÉ PRIORITAIRES À CORRIGER

→ Vérifier que users.json (données test) est bien dans .gitignore
→ Confirmer le hashage des passwords (bcrypt, argon2) — jamais en clair en base
→ Implémenter Rate Limiting sur les routes d'API (protection DDoS et brute force)
→ Ajouter des en-têtes de sécurité HTTP (Helmet.js pour Express)
→ Valider et sanitiser TOUTES les entrées utilisateur (protection XSS, injection)
→ Définir une politique CORS stricte (ne pas autoriser * en production)

---

6. Scalabilité

En l'état actuel (JSON local, single Node.js instance, pas de cache), la scalabilité est très limitée. Voici une analyse honnête des capacités :

Scénario	Capacité actuelle	Limite	Solution
< 100 utilisateurs/jour	Suffisante	JSON local peut tenir	Pas de changement urgent
1 000 utilisateurs/jour	Insuffisante	JSON = goulot d'étranglement total	Migration MongoDB URGENTE
10 000 utilisateurs/jour	Impossible	Architecture single-node insuffisante	MongoDB Atlas + PM2/Cluster
100 000+ utilisateurs/jour	Impossible	Refonte architecturale nécessaire	Microservices, Redis cache, CDN

Roadmap scalabilité recommandée

Phase 1 (MVP, <100 users/jour) : Architecture actuelle acceptable.
Phase 2 (Pilote, <1 000 users/jour) : MongoDB Atlas + PM2 + variables d'env sécurisées.
Phase 3 (Growth, <10 000 users/jour) : Redis pour le cache des sessions et des recherches fréquentes.
Phase 4 (Scale, 10 000+ users/jour) : Load balancer, replicas MongoDB, CDN pour assets statiques.

---

7. Performance

- ✅ **Frontend CDN :** Le frontend sur Vercel bénéficie du CDN mondial de Vercel — excellente performance pour les assets statiques.

- ⚠️ **Render cold start :** Le backend sur Render (tier gratuit) a un cold start de ~30 secondes après inactivité — inacceptable pour la production.
- ⚠️ **JSON = bottleneck :** Avec JSON local en backend, chaque lecture de données est un I/O disque synchrone bloquant potentiellement — les performances se dégradent très rapidement avec le volume de données.
- ⚠️ **Pas de cache :** Aucun système de cache mentionné (Redis, mémoire Node.js) — chaque requête API refait les mêmes calculs. "Je ne peux pas confirmer l'absence totale de cache sans accès au code."

Recommandations performance

→ Render payant (7 $/mois) pour supprimer le cold start.
→ Migrer vers MongoDB avec indexation correcte sur les champs de recherche.
→ Implémenter un cache en mémoire (node-cache ou Redis) pour les listes de pros par ville.
→ Optimiser les images (WebP, compression) et activer la compression gzip/brotli sur Express.

---

8. DevOps

8.1 Points positifs

- ✅ Docker + Docker Compose pour la reproductibilité locale.
- ✅ Déploiement automatique via Vercel (frontend) et Render (backend) sur push GitHub — CI/CD basique mais fonctionnel.
- ✅ ESLint + Prettier pour la qualité du code — bonne pratique.
- ✅ Architecture de monitoring mentionnée (Prometheus + Grafana + Node Exporter + Alertmanager) via le serveur Mediaschool — ambitieuse pour un MVP scolaire.

8.2 Lacunes DevOps

- ❌ Aucune pipeline CI/CD avancée mentionnée (GitHub Actions, tests automatiques avant merge).
- ❌ Aucune mention de stratégie de backup des données.
- ❌ Aucune gestion des environments (dev / staging / production) explicitement définie.
- ⚠️ Logs applicatifs : "Je ne peux pas confirmer la présence d'un système de logging structuré (Winston, Morgan)."

Recommandations DevOps

→ Ajouter GitHub Actions : lint + tests automatiques à chaque PR avant merge.
→ Définir 3 environments : local (Docker), staging (Render free), production (Render paid).
→ Implémenter un logging structuré (Morgan pour HTTP, Winston pour applicatif).

→ Configurer des alertes (UptimeRobot gratuit) pour monitoring de disponibilité.
→ Documenter la procédure de déploiement et de rollback.

---

9. Faiblesses techniques — Synthèse critique

Faiblesse	Criticité	Impact
Données en JSON local (pas de vraie BDD)	🔴 CRITIQUE	Scalabilité nulle, risque de perte de données
Stripe non confirmé actif	🔴 CRITIQUE	Pas de monétisation possible sans paiement
Pas de TypeScript	🟡 MODÉRÉ	Maintenabilité réduite sur le long terme
Pas de tests automatisés (probable)	🔴 CRITIQUE	Risque régressions en production
Render tier gratuit (cold start)	🟡 MODÉRÉ	UX dégradée pour le premier utilisateur du jour
Pas de système d'avis/notation	🟠 FORT	Confiance utilisateur impossible sans avis
Pas de vérification artisans	🔴 CRITIQUE	Risque légal et réputationnel majeur
Pas de notifications (email/SMS)	🟠 FORT	Parcours de réservation incomplet
Pas de gestion CORS/Helmet documentée	🟠 FORT	Vulnérabilités de sécurité potentielles
Pas d'app mobile native	🟡 MODÉRÉ	Friction d'usage sur smartphone (70 % du trafic web)

---

10. Recommandations — Roadmap technique prioritaire

10.1 Priorité immédiate (avant tout pilote public)

- 🔴 **P0 :** Migrer les données de JSON local vers MongoDB Atlas (tier gratuit disponible).
- 🔴 **P0 :** Confirmer et activer l'intégration Stripe Connect pour les paiements.
- 🔴 **P0 :** Implémenter Helmet.js, CORS strict, rate limiting, validation des inputs (Joi/Zod).
- 🔴 **P0 :** Vérifier que users.json et toutes données sensibles sont dans .gitignore.
- 🔴 **P0 :** Implémenter le hashage des mots de passe (bcrypt) si ce n'est pas déjà fait.

10.2 Court terme (1–3 mois)

- 🟠 **P1 :** Ajouter un système de notation/avis post-intervention.
- 🟠 **P1 :** Implémenter les notifications email (Nodemailer + SendGrid) pour confirmations de réservation.

- 🟠 **P1 :** Passer Render en tier payant (7 $/mois) pour supprimer le cold start.
- 🟠 **P1 :** Ajouter GitHub Actions avec tests lint + tests unitaires automatiques.
- 🟠 **P1 :** Définir le processus de vérification des artisans (Kbis, assurance, certifications).

10.3 Moyen terme (3–12 mois)

- 🟡 **P2 :** Migration vers TypeScript pour la maintenabilité.
- 🟡 **P2 :** Refactoring vers NestJS si l'équipe monte en compétence.
- 🟡 **P2 :** Application mobile Progressive Web App (PWA) — compromis mobile sans app store.
- 🟡 **P2 :** Système de matching géolocalisé avancé (index géospatial MongoDB 2dsphere).
- 🟡 **P2 :** Dashboard analytics professionnel (revenus, réservations, avis).

🏁 VERDICT TECHNIQUE FINAL

→ Stack : ✅ Choix techniques globalement cohérents pour un MVP

→ Implémentation : ⚠️ Plusieurs fonctionnalités annoncées non confirmées (Stripe, OSM)

→ Qualité code : ⚠️ Bonne structure apparente, mais pas de TypeScript ni tests

→ Sécurité : 🔴 Points critiques à corriger avant tout déploiement public

→ Scalabilité : ⚠️ Architecture actuelle suffit pour un MVP, refonte nécessaire à 1 000+ users/jour

→ Maturité globale : Projet pédagogique de qualité, nécessitant 3-6 mois de travail pour atteindre un niveau "production-ready" digne d'un investisseur