



On a suivi l'argent d'une séance de yoga dans une appli de réservation. Voilà ce qu'on a vu.

Diagnostic public n° 1. Studio Kerelune est l'application témoin de LeonoX : une application de réservation et de paiement qu'on a construite sur le modèle de ce que produisent les générateurs de code, pour la disséquer en public. Les défauts qu'elle contient y ont été placés volontairement, et chacun a été observé dans une vraie application. Le diagnostic a été conduit comme pour un client : chaque constat est prouvé dans le code ou en exécutant, et deux constats de ce rapport ne figuraient pas dans la liste des défauts placés. Révision dc7a65b, lue le 1er octobre 2026. Aucune donnée réelle, aucune clé réelle : tout est factice.

4

*constats critiques, tous sur l'argent ou les données

6

confirmés en exécutant

2

trouvés hors de la liste des défauts placés

2 jours

pour fermer l'essentiel

1 · SYNTHÈSE

Trois décisions à prendre cette semaine

- 1 Retirer la clé d'administration de la base du code envoyé au navigateur et la changer chez Supabase. Tant qu'elle est là, n'importe quel visiteur a les pleins pouvoirs sur la base, clients et paiements compris.
- 2 Fermer les règles d'accès « allow all ». Aujourd'hui, un visiteur sans compte lit le nom, l'e-mail et le téléphone de tous les clients, et un client connecté peut se nommer administrateur.
- 3 Faire décider le paiement par Stripe, pas par le navigateur : confirmation par le webhook signé, et remboursement réservé au propriétaire de la réservation. Aujourd'hui, une adresse tapée à la main confirme une séance non payée.

Ce qui marche

Studio Kerelune vend des séances de yoga et de pilates : planning, réservation, paiement par carte, e-mail de confirmation, annulation avec remboursement, abonnements, espace d'administration. L'interface est propre, le code est en TypeScript strict, les montants sont en centimes entiers et contrôlés par la base. Le prix envoyé à Stripe est relu côté serveur, pas pris dans l'écran. Toutes les tables ont la sécurité par ligne activée, les fonctions de paiement exigent un utilisateur connecté, les e-mails sont formatés à l'heure de Paris. Le code se reprend bien : moins de 1 % de duplication, 7 fonctions sur 504 au-dessus du seuil courant de complexité, aucune erreur de typage. Sur ordinateur, la page se charge en un peu plus d'une seconde. Le projet se construit sur une machine vierge en une commande, et un mode démo permet de le parcourir sans aucun service branché.

Ce qui va lâcher

Tout ce qui protège l'argent et les données a été décidé dans le navigateur. La clé qui contourne toutes les règles d'accès est livrée à chaque visiteur. Les règles d'accès posées le premier jour pour « démarrer » n'ont jamais été retirées : les règles plus fines ajoutées ensuite ne servent à rien. Le paiement est considéré comme réussi parce que la page de retour s'affiche ; la signature des messages de Stripe n'est pas vérifiée, et le webhook n'est là « qu'au cas où ». N'importe quel client peut déclencher le remboursement de n'importe quelle réservation. Autour : un code promo affiché mais jamais facturé, des e-mails envoyés depuis un domaine qui n'est pas celui du studio, deux clients qui peuvent prendre la dernière place, aucune alerte quand quelque chose casse, aucun test.

Ce qu'il faut faire

En deux jours : retirer et changer la clé, fermer les règles d'accès, couper la confirmation par le navigateur. En une à deux semaines : un paiement qui ne passe que par Stripe, de bout en bout, avec des tests ; la place réservée vérifiée par la base ; les e-mails authentifiés ; une remontée d'erreurs. En un mois : les abonnements branchés sur Stripe, un environnement d'essai, un déploiement qui teste avant de publier.

Une application qui a l'air finie et qui fonctionne pour un client honnête, mais dont toutes les serrures sont côté visiteur. Quatre constats critiques, un élevé qui grandit avec le succès du studio, cinq moyens. Six constats ont été confirmés en exécutant, pas seulement en lisant.

La météo des sept axes, pour un lecteur non technique. Rouge : à traiter cette semaine. Orange : dans le mois. Vert : rien d'urgent.



Le fil, et où sont les constats



2 · PÉRIMÈTRE ET MÉTHODE

Ce qu'on a regardé. Le dépôt entier à la révision `dc7a65b` : l'interface (68 fichiers, 6 722 lignes TypeScript), les 6 migrations de la base, les 5 fonctions serveur et leur code partagé, le workflow de déploiement, la configuration Supabase, l'historique Git. **Ce qu'on a exécuté**, en local et sans rien contacter : l'installation et la construction sur une copie vierge du dépôt ; la recherche des clés dans le code livré au navigateur ; le rejeu des 6 migrations dans un Postgres 15 jetable, puis des requêtes jouées tour à tour comme visiteur anonyme et comme client connecté, avec deux clients fictifs.

Ce qu'on n'a pas regardé. Les services réels : il n'y a pas de projet Supabase, de compte Stripe, de compte Resend ni de site Netlify derrière cette application témoin. Le comportement réel de Stripe et de Resend est donc lu dans le code, pas observé. La performance face à une vraie base et à de vrais visiteurs n'a pas été mesurée ; elle l'a été en local, sur des données fictives (constats J et K).

Le fil. L'argent d'une séance : une cliente choisit un créneau, réserve, paie par Stripe, revient sur le site, reçoit sa confirmation, puis annule et se fait rembourser. C'est le parcours qui traverse le plus de tables sensibles (profils,

réservations, paiements) et les trois services externes. L'espace d'administration est suivi en complément, parce qu'il lit tout.

Profil de profondeur. Sécurité, données, logique, qualité et performance approfondis ; architecture, obsolescence et exploitation surveillés. La qualité et la performance ont été mesurées : duplication, complexité, code mort, temps de chargement, coût de la requête principale selon le volume. Chaque constat porte une criticité définie par une question (voir section 5), un niveau de confiance (exécuté, lu, déduit) et sa preuve en annexe.

Sans objet ici, et dit tel quel. Fonctions IA dans l'application : aucune. Propriété des comptes : application témoin, tout appartient à Leonox. Données réelles : aucune, les clients sont fictifs.

Outils. Ce diagnostic a été conduit avec l'assistance d'un outil d'intelligence artificielle (Claude, d'Anthropic). L'outil accélère la lecture ; il ne signe rien. Chaque constat a été vérifié dans le code, les preuves marquées « exécuté » ont été rejouées, et le rapport a été relu et validé par Nicolas Jaubertou, qui en répond. Chez un client, aucun outil d'IA n'est utilisé sur son code sans son accord écrit.

Ligne rouge. Aucun service en ligne contacté, aucune clé utilisée ailleurs que dans un Postgres jetable, sur nos machines. Sur l'application d'un client, les vérifications exécutées ici le seraient sur une copie locale, jamais sur la production.

Portée de ce rapport. Ce rapport décrit l'application telle qu'elle nous a été remise, à la révision et à la date indiquées. Il dit ce qui a été regardé et ce qui ne l'a pas été : l'absence de constat sur un point n'est pas une attestation que ce point est sans défaut. Les recommandations sont des propositions argumentées ; vous décidez seul de les appliquer, par qui et quand, et chaque correction s'essaie hors production avant d'être déployée. Les efforts et les prix du plan sont des estimations, qui deviennent fermes par un devis.

3 · CE QU'ON VOIT

Une application React construite avec Vite et TypeScript, des composants d'interface écrits à la main dans le style shadcn, Supabase pour la base, l'authentification, le stockage et les fonctions serveur en Deno, Stripe Checkout pour le paiement, Resend pour les e-mails, Netlify pour l'hébergement, déployée par GitHub Actions. Exactement l'assemblage que livrent les générateurs.

Fichiers dans src		68
Lignes TypeScript		6 722
Dépendances	10 (+ 10 en développement)	
Tables	6, toutes avec la sécurité par ligne activée	
Migrations		6
Règles d'accès vivantes	16, dont 2 « allow all »	
Fonctions serveur	5, dont 3 sans vérification de l'appelant	
Fichiers de test		0
Plus gros fichier	Admin.tsx, 659 lignes	
console.error	121 dans src, aucun collecteur d'erreurs	
Code livré au navigateur	un seul fichier de 681 ko	
Historique	4 commits, une seule personne	

Fiche d'identité de l'application. Ce qu'un dirigeant doit savoir avant tout constat.

Ce qu'elle fait	Réservation et paiement de séances de yoga et de pilates, abonnements, espace client, administration
Pour qui	Un studio et ses clients
Où elle tourne	Interface chez Netlify ; base, comptes, fichiers et fonctions chez Supabase ; paiement Stripe ; e-mails Resend
Ce qu'elle encaisse	Des séances à l'unité (Stripe Checkout) ; les abonnements sont créés sans paiement
Ce qu'elle garde sur les clients	Nom, e-mail, téléphone, photo, historique de réservations et de paiements, consentement marketing
Qui détient les comptes	Leonox (application témoin). Chez un client : listés pendant le diagnostic, compte par compte
Se relance-t-elle ailleurs ?	Oui : installation et construction en quelques secondes avec Node 20, mode démo sans aucun service
Configuration	2 variables documentées sur 7 côté interface, 0 sur 5 côté serveur
Ce qu'elle coûte	Sans objet (application témoin). Chez un client : poste par poste, chaque mois
Qui sait	Une personne, 4 commits sur 4

Qualité et évolutivité. Le code se reprend bien : moins de 1 % de lignes dupliquées, 7 fonctions sur 504 au-dessus du seuil courant de complexité (10) et aucune au-dessus de 20, un typage strict sans aucune erreur. Ce qui freine : 411 lignes de code mort réparties dans 4 fichiers, 16 fonctions exportées que rien n'utilise, aucun vérificateur ni formateur pour tenir le style, et un écran d'administration écrit d'un bloc, une seule fonction de 593 lignes. Par où commencer : retirer le code mort, sans risque, en une heure ; installer un vérificateur et un formateur ; puis découper l'administration au moment de la sortir du navigateur (constat A), pour ne la réécrire qu'une fois.

4 · LE FIL : L'ARGENT D'UNE SÉANCE

Étape 1, le créneau. Le planning lit la table `slots`, lisible par tous, et c'est voulu. La page du créneau vérifie si la cliente a déjà réservé en téléchargeant toutes les réservations de tous les clients et en filtrant dans le navigateur (constats C et J). Vérifié : lu, exécuté et chronométré.

Étape 2, la réservation. `createBooking` compte les places restantes, puis insère la réservation, en deux requêtes séparées. Entre les deux, rien ne protège la dernière place, et la base n'a ni contrainte de capacité ni contrainte d'unicité (constat E). Vérifié : exécuté sur la base.

Étape 3, le paiement. La fonction `create-checkout` relit la réservation et le prix côté serveur et ouvre une session Stripe : c'est la meilleure partie du fil. Mais l'écran de paiement accepte le code `LUNE10` et affiche 10 % de réduction que Stripe ne connaît pas (constat H). Et la fonction ne vérifie pas que la réservation appartient à l'appelant. Vérifié : lu.

Étape 4, le retour et la confirmation. Au retour de Stripe, la page `PaymentReturn` lit `booking_id` et `session_id` dans l'adresse, enregistre elle-même un paiement « réussi » et confirme la réservation, sans rien demander à Stripe. Le webhook, qui devrait être la seule source de vérité, ne vérifie pas la signature des événements (constat B). Vérifié : lu et exécuté sur la base.

Étape 5, l'e-mail. `send-confirmation` envoie depuis `onboarding@resend.dev`, le domaine de test de Resend, et la fonction est ouverte sans vérification de l'appelant. Si l'envoi échoue, l'écran affiche quand même « en cours d'envoi » (constat D). Vérifié : lu.

Étape 6, l'annulation et le remboursement. L'annulation marque le paiement « remboursé » en base avant d'appeler Stripe, sans attendre la réponse. La fonction `refund-payment` rembourse la réservation qu'on lui donne, sans vérifier à qui elle appartient (constat G). Vérifié : lu.

À côté du fil, l'**administration**. La page `/admin` est masquée par l'écran aux non-administrateurs, mais toutes ses requêtes passent par un client Supabase construit avec la clé `service_role`, livrée dans le code du navigateur (constat A). Vérifié : exécuté.

Le fil fonctionne pour une cliente honnête. Il ne résiste à aucune cliente curieuse.

5 · LES CONSTATS

Criticité : **critique** si de l'argent ou des données personnelles sont exposés aujourd'hui sans rien changer ; **élevé** si ça casse au prochain changement, à la montée en charge ou au départ de la personne qui sait ; **moyen** si ça coûte à chaque fois qu'on touche l'application ; **faible** pour l'hygiène. À vérifier n'est pas un niveau : c'est une question au propriétaire.

Les quatre critiques

A

CRITIQUE

EXÉCUTÉ

SÉCURITÉ

La clé d'administration de la base est livrée à chaque visiteur

`src/lib/supabaseAdmin.ts` construit un client Supabase avec la clé `service_role`, écrite en dur en repli d'une variable d'environnement. Le commentaire raconte l'histoire : « avec la clé anon la page admin renvoyait des tableaux vides (problème de RLS ?) donc on utilise la clé `service_role` qui contourne les politiques, ça marche ». Les 11 requêtes de l'administration l'utilisent. Après construction, la clé est dans le fichier JavaScript servi à tous les visiteurs : son contenu décodé dit `"role": "service_role"`. La page admin elle-même n'est protégée que par une redirection dans l'écran.

Impact. La clé `service_role` ignore toutes les règles d'accès. Quiconque ouvre les outils de développement de son navigateur peut lire, modifier ou effacer tous les profils, toutes les réservations, tous les paiements, et créer des administrateurs.

Coût de ne rien faire. Une fuite de données clients à notifier à la CNIL sous 72 heures, et une base qu'un inconnu peut vider un dimanche soir.

Correction. Supprimer le client admin du navigateur ; changer la clé chez Supabase, puisqu'elle est publique ; servir l'administration par des fonctions serveur qui vérifient le rôle de l'appelant, ou par des règles d'accès `is_admin()` (la fonction existe déjà en base). **Effort** : une journée.

C

CRITIQUE

EXÉCUTÉ

DONNÉES, SÉCURITÉ

Les règles « allow all » du premier jour n'ont jamais été retirées

La première migration ouvre `profiles` et `bookings` à tout le monde, « pour démarrer, on affinera les règles plus tard ». Un mois après, une migration ajoute les règles fines (« chacun voit ses réservations »), mais ne supprime pas les anciennes. Dans Postgres, les règles s'additionnent : la plus large gagne. Rejouées dans une base jetable, les migrations donnent ceci : un visiteur **sans compte**, avec la seule clé publique, lit l'e-mail et le téléphone de tous les clients et toutes les réservations ; une cliente connectée annule la réservation d'un autre client ; et elle passe son propre profil en `role = 'admin'`. L'écran « Mes réservations » masque le problème en filtrant dans le navigateur.

Impact. Toutes les données personnelles des clients sont publiques, et n'importe qui peut devenir administrateur en une requête. Les règles fines donnent une fausse impression de sécurité à qui relit les migrations récentes.

Coût de ne rien faire. Le même que A, sans même avoir besoin de la clé d'administration.

Correction. Supprimer les deux règles « allow all » ; interdire la modification de `role` par l'utilisateur (règle ou colonne protégée) ; filtrer par `user_id` dans les requêtes. Puis rejouer le test de l'annexe, qui doit tout refuser.

Effort : une demi-journée, test compris.

B

CRITIQUE

EXÉCUTÉ

LOGIQUE

Le paiement est déclaré réussi par la page, pas par Stripe

Au retour de Stripe, `PaymentReturn.tsx` (ligne 45) appelle `confirmPayment` avec les paramètres de l'adresse. `confirmPayment` insère un paiement `succeeded` et confirme la réservation, depuis le navigateur, parce que « Stripe nous a redirigé sur `success_url` donc le paiement est passé ». La base l'autorise : une règle laisse chaque client insérer et modifier ses propres paiements. Exécuté dans la base jetable : une cliente enregistre un paiement réussi de 18 € avec `stripe_session_id = 'n_importe_quoi'`, accepté. Côté serveur, le webhook Stripe lit l'événement tel quel, la vérification de signature est commentée (« faisait planter en local »), et la fonction est déployée sans vérification de l'appelant.

Impact. Une séance se réserve sans payer en tapant l'adresse de retour à la main. N'importe qui peut envoyer au webhook un faux événement « paiement complété » et confirmer n'importe quelle réservation. La base ne peut plus servir de rapprochement avec Stripe : elle contient des paiements que Stripe n'a jamais vus.

Coût de ne rien faire. Des séances données, des places prises par des non-payeurs, et une comptabilité qui ne correspond plus aux encaissements.

Correction. Le webhook signé devient la seule source de vérité : vérification de signature, idempotence par identifiant d'événement, écriture des paiements par le serveur uniquement. Supprimer les règles d'insertion et de modification des paiements côté client. La page de retour se contente d'attendre la confirmation. **Effort :** deux jours, tests compris.

G

CRITIQUE

LU

HORS DE LA LISTE DES DÉFAUTS PLACÉS

LOGIQUE, SÉCURITÉ

N'importe quel client peut rembourser n'importe quelle réservation

`refund-payment` exige un utilisateur connecté, puis rembourse chez Stripe la réservation dont on lui donne l'identifiant, sans vérifier qu'elle appartient à l'appelant. Les identifiants de toutes les réservations sont lisibles par tous (constat C). En parallèle, l'annulation côté navigateur marque le paiement « remboursé » en base avant même d'appeler Stripe, sans attendre ni vérifier la réponse.

Impact. Un client mécontent, ou un concurrent avec un compte gratuit, peut rembourser toutes les séances payées du studio, une par une. Et quand un remboursement échoue chez Stripe, la base dit qu'il a eu lieu.

Coût de ne rien faire. Le chiffre d'affaires de la période, rendu à des clients qui n'ont rien demandé, et des réclamations que personne ne pourra expliquer avec la base.

Correction. Vérifier dans la fonction que l'appelant est le propriétaire ou un administrateur, et que la séance est encore annulable ; écrire le statut « remboursé » seulement après la réponse de Stripe, ou par le webhook `charge.refunded`. Même vérification à ajouter dans `create-checkout`. **Effort :** une demi-journée.

Ce qui coûte, et les réglages

J

ÉLEVÉ

EXÉCUTÉ

QUALITÉ ET PERFORMANCE, LOGIQUE

« Mes réservations » télécharge les réservations de tous les clients et, au-delà de 1 000, en oublie

Pour afficher les réservations d'une cliente, `listBookings` demande à la base toutes les réservations de tous les clients, avec leur créneau, leur profil et leur paiement, puis l'écran ne garde que les siennes. La page d'un créneau et les notifications font de même. Chronométrage dans une base jetable, sur des données fictives, comparé à la même requête filtrée sur la cliente :

RÉSERVATIONS EN BASE	REQUÊTE DE L'APPLICATION	DONNÉES ENVOYÉES	REQUÊTE FILTRÉE	DONNÉES ENVOYÉES
100	7 ms	131 ko	1 ms	1 ko
1 000	25 ms	1,3 Mo	1 ms	9 ko
10 000	314 ms	13 Mo	4 ms	28 ko
50 000	1 638 ms	64 Mo	8 ms	141 ko

Ces mesures sont faites sans le plafond de Supabase, pour montrer la pente. Car la configuration du projet plafonne chaque réponse à 1 000 lignes (`max_rows = 1000`, la valeur par défaut de Supabase). Au-delà de 1 000 réservations, la base ne renvoie que les 1 000 plus récentes, sans erreur.

Impact. Vers mille réservations, chaque affichage de « Mes réservations » télécharge plus d'un mégaoctet sur le téléphone de la cliente. Au-delà de 1 000, une cliente fidèle ne voit plus ses anciennes réservations, et la page d'un créneau ne détecte plus qu'elle l'a déjà réservé. Le problème grandit avec le succès du studio, et il est invisible tant qu'il est petit.

Coût de ne rien faire. Des clientes qui croient leurs réservations perdues et appellent le studio ; des doublons qui passent (voir E) ; une application qui ralentit chaque mois sans que personne ne sache pourquoi.

Correction. Filtrer sur la cliente dans la requête elle-même, ce que la correction de C impose de toute façon ; paginer l'historique ; une requête dédiée pour « ai-je déjà réservé ce créneau ». **Effort :** une demi-journée, avec C.

H

MOYEN

LU

HORS DE LA LISTE DES DÉFAUTS PLACÉS

LOGIQUE

Le code promo est affiché, jamais facturé

L'écran de paiement accepte `LUNE10` et affiche un total réduit de 10 %. Le commentaire dit « les codes promo sont gérés côté Stripe », mais la session Stripe est créée avec le prix plein et sans option de code promo.

Impact. Pour une séance à 18 €, le client voit 16,20 € et paie 18 €. Pour lui, c'est une erreur de facturation ; pour le studio, une pratique commerciale trompeuse sur le prix affiché.

Correction. Créer le code chez Stripe et l'appliquer dans la session côté serveur, ou retirer le champ. **Effort** : deux heures.

E

MOYEN

EXÉCUTÉ

LOGIQUE, DONNÉES

Deux clients peuvent prendre la dernière place

`createBooking` vérifie la place restante, puis insère, en deux requêtes séparées, sans transaction. La base n'a aucune contrainte de capacité, ni d'unicité sur le couple client et créneau. Exécuté : sur un créneau d'une place, déjà réservé, la base accepte deux réservations de plus, dont un doublon de la même cliente. L'écran avertit d'un doublon mais ne l'empêche pas.

Impact. Des cours en surnombre, un client refusé à l'entrée alors qu'il a payé, un remboursement à faire à la main. Plus le studio a du succès, plus ça arrive : c'est aux heures de pointe que deux personnes cliquent en même temps.

Correction. Une fonction en base qui vérifie la capacité et insère dans la même transaction, avec verrou sur le créneau ; une contrainte d'unicité client et créneau sur les réservations actives. **Effort** : une demi-journée.

D

MOYEN

LU

EXPLOITATION

Les e-mails partent d'un domaine qui n'est pas celui du studio, et personne ne lit les rebonds

`send-confirmation` envoie depuis `onboarding@resend.dev`, le domaine de démonstration de Resend, en attendant de « configurer SPF / DKIM sur `studio-kerelune.example` plus tard ». Les événements de Resend, rebonds compris, sont enregistrés dans `email_events`, que rien ne lit jamais. Si l'envoi échoue, l'écran affiche « l'e-mail de confirmation est en cours d'envoi ». La fonction est aussi ouverte sans vérification de l'appelant : n'importe qui peut déclencher un envoi au nom du studio vers l'adresse d'une réservation.

Impact. Les confirmations arrivent en spam, ou n'arrivent pas, sans que personne le sache. Le client appelle le studio, qui ne voit rien.

Correction. Domaine du studio authentifié chez Resend (SPF, DKIM, DMARC) ; suppression des adresses en rebond avant chaque envoi ; vérification de l'appelant ; message d'écran honnête en cas d'échec. **Effort** : une demi-journée, plus le délai DNS.

I MOYEN LU EXPLOITATION**Quand quelque chose casse, personne ne le sait**

121 `console.error` dans l'interface, qui s'affichent dans le navigateur du client et nulle part ailleurs. Une variable `VITE_SENTRY_DSN` est lue, jamais utilisée. Aucun délai maximal sur les appels réseau. Le déploiement publie directement en production à chaque envoi sur `main`, avec `npm install` plutôt que `npm ci`, sans aucune étape de test et sans environnement d'essai.

Impact. Un paiement qui échoue, un e-mail qui ne part pas, un remboursement refusé par Stripe : le studio l'apprend par un client, ou ne l'apprend pas.

Correction. Un collecteur d'erreurs côté interface et serveur, avec alerte ; une préproduction ; `npm ci` et une étape de test bloquante. **Effort :** une journée.

K MOYEN EXÉCUTÉ QUALITÉ ET PERFORMANCE**Sur mobile, le planning met 5 secondes à s'afficher**

Mesuré avec Lighthouse, l'outil de mesure de Google, sur la page du planning servie compressée comme le ferait l'hébergeur, trois passages par mode : sur mobile, score de 67 à 72 sur 100 et 5,2 secondes avant l'affichage du contenu principal (Google considère « bon » en dessous de 2,5 s) ; sur ordinateur, 97 sur 100 et 1,2 s. La page transfère 521 ko, dont 265 ko de Stripe.js : le module de paiement est importé d'une façon qui charge Stripe sur toutes les pages, planning compris, alors qu'il ne sert qu'au paiement. Le reste de l'application tient en un seul fichier de 681 ko (197 ko compressé), sans aucun découpage par page : l'administration est livrée à tous les visiteurs, et 299 ko de JavaScript ne servent pas à afficher le planning. Les polices viennent de Google Fonts, ce qui transmet l'adresse IP de chaque visiteur à Google sans son consentement.

Impact. Sur un téléphone en 4G, la cliente attend cinq secondes avant de voir les cours ; une partie part avant. La transmission des adresses IP à Google est un sujet RGPD connu, déjà sanctionné par un tribunal allemand en 2022.

Correction. Importer Stripe dans sa version qui ne charge rien d'avance (`@stripe/stripe-js/pure`) et ne le charger qu'au paiement ; découper l'application par page, administration à part ; héberger les polices avec le site. Puis remesurer. **Effort :** une journée.

6 · TOUT CE QU'ON A TROUVÉ, PAR CRITICITÉ

Les dix constats lettrés sont détaillés en section 5. Les autres lignes sont des risques sans victime aujourd'hui : ils deviennent des constats au déclencheur indiqué.

CRITICITÉ	CONSTAT	AXE	CONFIANCE
Critique	A · Clé d'administration dans le code du navigateur	Sécurité	Exécuté
Critique	C · « allow all » : données publiques, on peut se nommer admin	Données	Exécuté
Critique	B · Paiement confirmé par la page, webhook sans vérification de signature	Logique	Exécuté
Critique	G · Remboursement de la réservation d'un autre	Logique	Lu
Élevé au premier abonnement payant	Les abonnements sont créés en base depuis le navigateur sans paiement ; la fin d'essai ne fait rien (aucune tâche planifiée)	Logique	Lu
Élevé au premier ménage dans le planning	Supprimer un créneau efface ses réservations et détache leurs paiements (on <code>delete cascade</code>)	Données	Lu

CRITICITÉ	CONSTAT	AXE	CONFIANCE
Élevé au prochain changement de projet	Deux projets Supabase référencés : l'ancien dans <code>config.toml</code> , le nouveau dans le déploiement	Exploitation	Lu
Élevé à la montée en charge	J · « Mes réservations » télécharge tout, et oublie au-delà de 1 000	Qualité et performance	Exécuté
Élevé au prochain changement	Aucun test (Vitest installé, jamais utilisé), configuration non documentée (2 variables sur 12)	Logique, obsolescence	Exécuté
Moyen	H, E, D, I, K	Logique, exploitation, performance	E et K exécutés
Moyen	Fichier <code>.env</code> versionné, <code>.gitignore</code> ne l'exclut pas	Sécurité	Lu
Faible	411 lignes de code mort dans 4 fichiers, 16 fonctions exportées jamais utilisées, 2 dépendances inutiles ; aucun vérificateur ni formateur (ESLint référencé mais absent) ; l'écran d'administration écrit d'un bloc, une fonction de 593 lignes	Qualité et performance	Exécuté
À vérifier	Sauvegardes et dernier test de restauration ? Durée de conservation des données des clients ? Coût mensuel ? Qui reçoit les alertes ?	Données, exploitation	Non regardé

Non regardé, et à dire : la performance face à une vraie base Supabase et à de vrais visiteurs (il n'y en a pas derrière cette application témoin), l'accessibilité, le stockage des photos de profil (le bucket n'existe pas dans les migrations), les conditions générales de vente.

7 · LE PLAN

Application témoin : les prix ci-dessous appliquent les tarifs de Leonox à ce cas, pour montrer ce qu'un client recevrait. Ce sont des estimations ; elles deviennent fermes par un devis.

Ce que l'auteur fait seul, en une heure

Exclure `.env` du dépôt et changer les clés qu'il contenait : elles restent lisibles dans l'historique. Retirer le champ de code promo tant qu'il n'est pas branché (H). Passer le déploiement à `npm ci`. Écrire les variables manquantes dans `.env.example`. Supprimer les 411 lignes de code mort.

En deux jours, ce qui ne peut pas attendre

Retirer la clé d'administration du navigateur et la changer (A). Supprimer les règles « allow all » et protéger le rôle (C). Couper la confirmation par la page de retour et les écritures de paiement côté client (B). Vérifier le propriétaire dans le remboursement et la création de session (G). Filtrer « Mes réservations » sur la cliente (J).

Compris dans la stabilisation, ou réalisable seul en urgence.

En une à deux semaines, la stabilisation

Paiement de bout en bout par le webhook signé, idempotent, avec rapprochement (B, G). Réservation atomique avec contrainte de capacité (E). E-mails authentifiés et rebonds traités (D). Remontée d'erreurs et alertes (I). Chargement allégé : Stripe seulement au paiement, découpage par page, polices hébergées avec le site, puis remesure (K). Tests sur les six étapes du fil, qui tournent à chaque déploiement. Puis le même tableau qu'en section 6, recoché.

Stabilisation estimée à 5 900 € HT, sept jours ouvrés ; prix ferme au devis.

En un mois, avant de vendre des abonnements

Abonnements par Stripe Billing, fin d'essai traitée par une tâche planifiée, suppression des créneaux sans cascade sur les paiements, un seul projet Supabase, une préproduction.

Chiffré en fin de stabilisation, quand on sait ce qui reste.

Ensuite, la garde

Un périmètre nommé (le dépôt, le projet Supabase, Stripe, Resend, Netlify), des règles écrites pour l'outil de génération que le studio continue d'utiliser, des tests qui bloquent ce qui casse, les alertes qui arrivent chez nous, une page par mois.

Dès 1 500 € HT par mois.

8 · L'APRÈS

Une cliente réserve la dernière place du cours de 19 h, paie, reçoit son e-mail depuis l'adresse du studio. Personne d'autre ne voit son numéro de téléphone. Le paiement est en base parce que Stripe l'a dit, et pas parce qu'une page s'est affichée. Et le studio peut ouvrir les abonnements : ce qu'il encaisse correspond enfin à ce que dit sa base.

ANNEXE · REPRODUCTIBILITÉ

Toutes les preuves se lisent dans le dépôt Studio Kerelune, publié sur github.com/leonox-sas/studio-kerelune, au tag `diagnostic-2026-10-01` (révision `dc7a65b`). La stabilisation y sera faite au grand jour, pour que l'avant et l'après se comparent ligne à ligne. Les scripts qui rejouent les preuves de la base sont publiés à part. Les preuves exécutées l'ont été en local, sur une copie vierge du dépôt et dans un Postgres 15 jetable, sans contacter aucun service. Les scripts et la sortie complète sont publiés sur www.leonox.tech/rapports/studio-kerelune-preuves/.

- **Inventaire** : `find src -type f | wc -l(68); find src -name '*.ts*' -exec cat {} + | wc -l(6722); ls supabase/migrations(6); ls supabase/functions(5+_shared); find src -name '*.test.*' (aucun); git log --format=%an | sort | uniq -c(4 commits, un auteur).`
- **Construction** : `Node 20.20, git clone, npm ci, npm run build(8s); ls -l dist/assets` (un fichier JavaScript de 681 452 octets). Avertissement Vite sur les morceaux de plus de 500 ko.
- **A** : `src/lib/supabaseAdmin.ts` lignes 4 à 12; `grep -c supabaseAdmin src/data/supabaseAdapter.ts(11); src/components/layout/ProtectedRoute.tsx` ligne 27; `src/hooks/useAuth.tsx` ligne 92. Exécuté : après construction, extraction des jetons `eyJ...` du fichier `dist/assets/index-*.js` et décodage de leur contenu : deux jetons, `"role": "anon"` et `"role": "service_role"`.

- **C:** supabase/migrations/20260302091500_initial_schema.sql lignes 78 à 84 ; 20260402083000_profiles_role_marketing.sql lignes 6 à 20 (règles ajoutées, aucun drop policy); grep -n "drop policy" supabase/migrations/*.sql (aucun résultat); src/data/supabaseAdapter.ts lignes 177 à 190 ; src/hooks/useBookings.ts lignes 22 et 23. Exécuté : 00_stub.sql (rôles et schéma auth de Supabase), les 6 migrations, 99_grants.sql (droits par défaut de Supabase et deux clients fictifs), puis test.sql, requêtes 1 à 4 : lecture des profils et des réservations en anon, mise à jour de role et annulation d'une réservation d'autrui en authenticated. Les quatre réussissent (result.txt).
- **B:** src/pages/PaymentReturn.tsx ligne 45 ; src/data/supabaseAdapter.ts lignes 282 à 311 ; supabase/migrations/20260305141200_payments.sql lignes 24 à 29 ; supabase/functions/stripe-webhook/index.ts lignes 1 à 3, 8 et 18 ; supabase/config.toml ([functions.stripe-webhook] verify_jwt = false); .github/workflows/deploy.yml (--no-verify-jwt). Exécuté : test.sql, requête 5, insertion d'un paiement succeeded par la cliente, acceptée.
- **G:** supabase/functions/refund-payment/index.ts lignes 18 à 50 (aucune lecture de l'appelant); src/data/supabaseAdapter.ts lignes 244 à 258 (statut écrit avant l'appel, appel non attendu); supabase/functions/create-checkout/index.ts lignes 18 à 26.
- **H:** src/pages/Checkout.tsx lignes 55, 56 et 89 ; supabase/functions/create-checkout/index.ts lignes 36 à 48 (prix plein); grep -rn "allow_promotion\|discounts" supabase/functions (aucun résultat).
- **E:** src/data/supabaseAdapter.ts lignes 206 à 228 ; grep -in unique supabase/migrations/*.sql (aucun résultat). Exécuté : test.sql, requête 6, deux réservations insérées sur un créneau de capacité 1 déjà plein, dont un doublon ; la base compte 2 réservations pour 1 place.
- **D:** supabase/functions/send-confirmation/index.ts lignes 8 à 10 et 84 à 87 ; supabase/functions/resend-webhook/index.ts lignes 16 à 27 ; grep -rn email_events src (aucun résultat); src/data/supabaseAdapter.ts lignes 365 à 373 ; supabase/config.toml ([functions.send-confirmation] verify_jwt = false).
- **I:** grep -rc console.error src | awk -F: '{s+=\$2} END {print s}' (121); src/lib/env.ts ligne 10 (SENTRY_DSN, jamais importé); src/components/ErrorBoundary.tsx lignes 23 à 25 ; grep -rn "AbortController\|signal:" src supabase (aucun résultat); .github/workflows/deploy.yml (npm install, aucune étape de test).
- **Tests et configuration :** package.json lignes 6 à 11 (aucun script test); exécuté : npm test (« Missing script: test »), npm run lint (« eslint: not found »); .env.example (2 variables) contre src/lib/env.ts lignes 3 à 10 et src/lib/supabaseAdmin.ts ligne 9.
- **J:** supabase/config.toml ligne 10 (max_rows = 1000); src/data/supabaseAdapter.ts lignes 177 à 190 (aucun filtre); src/hooks/useBookings.ts lignes 22 et 23 ; src/pages/SlotDetail.tsx lignes 33 et 34. Exécuté : charge.sql, dans le même Postgres jetable, 500 clients et de 100 à 50 000 réservations fictives ; la requête de l'application reproduite avec ses jointures (créneau, profil, paiement), chronométrée et pesée en JSON, contre la même requête filtrée sur une cliente (charge-resultat.txt).
- **K:** Lighthouse 12, page /creneaux servie compressée (npx serve -s dist), trois passages en mobile, trois en ordinateur, valeurs médianes (lighthouse-resume.txt); src/lib/stripe.ts ligne 1 (import de @stripe/stripe-js, qui charge Stripe dès l'ouverture); index.html lignes 9 et 10 (Google Fonts); grep -rn "lazy(\|import(" src (aucun résultat); composition du fichier livré par source-map-explorer (react-dom 31 %, code de l'application 18 %, bibliothèques Supabase 31 %).
- **Qualité :** duplication par jscpd (seuil de 50 jetons : 8 répétitions, 50 lignes dupliquées, moins de 1 %); complexité de chaque fonction par complexite.cjs, qui s'appuie sur le compilateur TypeScript du projet (504 fonctions, 7 au-dessus de 10, aucune au-dessus de 20 ; Admin, 593 lignes); code mort par knip (src/hooks/useNotifications.ts, src/lib/analytics.ts, src/lib/http.ts, src/lib/storage.ts, 411 lignes ; 16 exports ; vitest et @testing-library/react inutilisés ; les fonctions serveur, que knip signale à tort, sont des points d'entrée Deno); npx tsc -b --force (aucune erreur).
- **Bloc 6 :** src/pages/Pricing.tsx ligne 48 et 20260312101500_subscriptions.sql lignes 22 à 31 (abonnements et fin d'essai); 20260302091500_initial_schema.sql ligne 38 (on delete cascade); supabase/config.toml ligne 3 contre .github/workflows/deploy.yml lignes 9 et 13 (deux projets); .env versionné et absent de .gitignore ; grep -ro "as any" src | wc -l (8).
- **Ce qui marche :** tsconfig.app.json ("strict": true); supabase/functions/create-checkout/index.ts lignes 22 à 40 (prix relu côté serveur); contraintes check sur montants et statuts dans les migrations ; supabase/config.toml (verify_jwt = true pour create-checkout et refund-payment).



On reprend le code que **personne** n'**assume**.

Une application générée par une IA, ou un outil hérité de quelqu'un qui est parti : ça tourne, personne ne le comprend, personne n'ose y toucher. Leonox le diagnostique, le stabilise, puis l'assume dans la durée.

Studio Kerelune est une application construite par Leonox pour être disséquée en public. Ses défauts y ont été placés volontairement, et chacun a été observé dans une vraie application. Les diagnostics de nos clients, eux, restent privés.

Votre application en a-t-elle besoin ? Vingt minutes pour le savoir : cal.com/leonox/vingt-minutes · bonjour@leonox.tech · diagnostic 2 900 € HT. Les preuves de ce rapport : www.leonox.tech/rapports/studio-kerelune-preuves/