Analyse complète du projet Call a Star

Date: 31 décembre 2025
Branche: feature/email-cron-booking-security
Objectif: Analyse approfondie pour les améliorations de sécurité, emails et CRON

📋 Table des matières

Architecture globale

Structure du projet Next.js

Le projet utilise **Next.js 14** avec l’App Router et TypeScript.

Structure des répertoires principaux

```
callastar/
├── app/                          # App Router de Next.js
│   ├── [locale]/                 # Support multilingue (fr/en)
│   │   ├── dashboard/            # Dashboards utilisateur/créateur/admin
│   │   ├── auth/                 # Pages d'authentification
│   │   ├── book/                 # Pages de réservation
│   │   ├── call/                 # Pages d'appels vidéo
│   │   ├── creators/             # Pages des créateurs
│   │   └── legal/                # Pages légales
│   └── api/                      # Routes API
│       ├── auth/                 # Authentification (NextAuth)
│       ├── bookings/             # Gestion des réservations
│       ├── payments/             # Paiements et webhooks Stripe
│       ├── cron/                 # Jobs CRON
│       ├── daily/                # Intégration Daily.io
│       └── ...
├── components/                   # Composants React réutilisables
│   ├── ui/                       # Composants UI de base (shadcn/ui)
│   └── admin/                    # Composants spécifiques admin
├── lib/                          # Bibliothèques et utilitaires
│   ├── auth.ts                   # Gestion de l'authentification
│   ├── db.ts                     # Client Prisma
│   ├── email.ts                  # Système d'envoi d'emails
│   ├── daily.ts                  # Client Daily.io
│   ├── stripe.ts                 # Client Stripe
│   ├── system-logger.ts          # Système de logs
│   ├── notifications.ts          # Notifications in-app
│   └── ...
├── prisma/                       # Configuration Prisma ORM
│   └── schema.prisma             # Schéma de base de données
├── messages/                     # Fichiers de traduction i18n
│   ├── en.json                   # Traductions anglaises
│   └── fr.json                   # Traductions françaises
└── scripts/                      # Scripts utilitaires
    ├── seed.ts                   # Script de seed de la BDD
    └── cleanup-logs.ts           # Nettoyage des logs
```

Technologies principales

- **Framework:** Next.js 14.2.32 avec App Router
- **Langage:** TypeScript 5.9.3
- **Base de données:** PostgreSQL avec Prisma ORM 6.7.0
- **Authentification:** NextAuth.js 4.24.11
- **Paiements:** Stripe 20.1.0 avec Stripe React 5.4.1
- **Appels vidéo:** Daily.co (daily-js 0.85.0)
- **Emails:** Resend 6.6.0
- **Internationalisation:** next-intl 4.6.1
- **UI:** Tailwind CSS 3.3.3 + shadcn/ui (Radix UI)
- **State Management:** Zustand 5.0.3

Système de paiement Stripe

Architecture des paiements

Le projet utilise **Stripe Connect** avec le modèle **Destination Charges**.

Fichier principal

- `lib/stripe.ts` - Client Stripe et fonctions de paiement

Fonctionnement actuel

1. Création d'un Payment Intent (`createPaymentIntent`)

```
// Destination Charges avec application_fee_amount
export async function createPaymentIntent({
  amount,                      // Montant en unités (ex: 100 EUR)
  currency = 'eur',           // Devise (EUR, CHF, USD, GBP)
  metadata = {},              // Métadonnées
  stripeAccountId,            // Compte Stripe Connect du créateur
  platformFeePercentage,      // Commission plateforme (ex: 15%)
})
```

Calcul des frais:
- Montant total payé par l'utilisateur: 100 EUR
- Commission plateforme (15%): 15 EUR
- Frais Stripe (2.9% + 0.30): ~3.20 EUR
- Créateur reçoit: ~81.80 EUR

Points clés:
- ✅ Utilise `transfer_data.destination` pour spécifier le compte du créateur
- ✅ `application_fee_amount` définit la commission de la plateforme
- ✅ Le transfert est automatique après succès du paiement
- ✅ Support multi-devises (EUR, CHF, USD, GBP)

2. Gestion des devises

Fonction `getCreatorCurrency(creatorId)` :
- Récupère la devise du créateur depuis la BDD (cache)
- Si non disponible, interroge le compte Stripe Connect
- Met à jour la BDD avec la devise récupérée
- Fallback sur 'EUR' en cas d'erreur

Fonction `getCreatorCurrencyByStripeAccount(stripeAccountId)` :
- Toujours récupère depuis Stripe (source de vérité)
- Met à jour la BDD si divergence détectée
- Lance une erreur si échec (pas de fallback silencieux)

3. Payouts (virements vers créateurs)

Deux méthodes de payout:

A. Automatique (via Stripe Connect Schedule):
- Configuré dans les paramètres du créateur (`PayoutScheduleNew`)
- Fréquences: DAILY, WEEKLY, MANUAL
- Montant minimum configurable (`payoutMinimum`)

B. Manuel (à la demande):
- Le créateur demande un virement
- Approbation admin requise
- Création du payout Stripe après validation

Fonction `createConnectPayout` :

```
export async function createConnectPayout({
  amount,                    // Montant en unités
  currency = 'eur',          // Devise
  stripeAccountId,           // Compte du créateur
  metadata = {},             // Métadonnées
})
```

4. Webhooks Stripe

Fichier: `app/api/payments/webhook/route.ts`

Événements traités:
- `payment_intent.succeeded` - Paiement réussi
- `payment_intent.payment_failed` - Paiement échoué
- `charge.refunded` - Remboursement
- `charge.dispute.created` - Litige créé
- `payout.paid` - Payout effectué
- `payout.failed` - Payout échoué

Sécurité:
- Vérification de la signature Stripe
- Idempotence (détection des événements déjà traités)
- Logging complet avec `system-logger.ts`

Modèles de données liés aux paiements

Payment

```
model Payment {
  id                    String        @id @default(cuid())
  bookingId             String        @unique
  amount                Decimal       @db.Decimal(10, 2)
  currency              String        @default("EUR")
  stripePaymentIntentId String
  status                PaymentStatus @default(PENDING)
  platformFee           Decimal       @db.Decimal(10, 2)
  creatorAmount         Decimal       @db.Decimal(10, 2)
  refundedAmount        Decimal       @default(0) @db.Decimal(10, 2)

  // Payout tracking
  payoutStatus          PayoutStatus  @default(REQUESTED)
  payoutReleaseDate     DateTime?     // Date + 7 jours
  stripeTransferId      String?
  transferId            String?
  transferStatus        String?
  payoutDate            DateTime?
}
```

Payout

```
model Payout {
  id              String       @id @default(cuid())
  creatorId       String
  amount          Decimal      @db.Decimal(10, 2)
  currency        String       @default("EUR")
  status          PayoutStatus @default(REQUESTED)
  stripePayoutId String?      @unique

  requestedAt     DateTime     @default(now())
  approvedAt      DateTime?
  paidAt          DateTime?
  failedAt        DateTime?
  rejectedAt      DateTime?
}
```

Points d'attention - Paiements

⚠️ **Problèmes potentiels identifiés:**

1. **Holding period de 7 jours**
   - Variable: `PAYOUT_HOLDING_DAYS = 7` dans `lib/stripe.ts`
   - Période de sécurité pour gérer les litiges
   - Les fonds ne sont disponibles qu'après 7 jours

2. **Gestion des devises**
   - Chaque créateur peut avoir sa propre devise
   - Besoin de conversions correctes dans les calculs
   - Risque d'incohérence entre DB et Stripe

3. **Refunds et disputes**
   - Système de tracking de la dette créateur (`creatorDebt`)
   - Réconciliation via `reconciledBy` : TRANSFER_REVERSAL, PAYOUT_DEDUCTION, MANUAL

Système de booking

Architecture du booking

Le système de booking gère les réservations d'appels vidéo entre utilisateurs et créateurs.

Fichiers principaux

- **`app/api/bookings/route.ts`** - CRUD des bookings
- **`app/api/bookings/[id]/route.ts`** - Gestion d'un booking spécifique
- **`app/[locale]/book/[offerId]/page.tsx`** - Page de réservation
- **`components/calendar-view.tsx`** - Composant calendrier

Flux de réservation

1. Création d'un booking

Endpoint: `POST /api/bookings`

Vérifications de sécurité actuelles:

```
// 1. Vérifier que l'offre existe
const callOffer = await db.callOffer.findUnique({
  where: { id: callOfferId },
  include: { booking: true }
});

// 2. Vérifier le statut de l'offre
if (callOffer.status !== 'AVAILABLE') {
  return error('Cette offre n\'est plus disponible');
}

// 3. Vérifier qu'il n'y a pas déjà un booking
if (callOffer.booking) {
  return error('Cette offre est déjà réservée');
}

// 4. Vérifier que la date n'est pas passée
if (new Date(callOffer.dateTime) < new Date()) {
  return error('Cette offre est expirée');
}
```

⚠️ **PROBLÈME CRITIQUE - Race Condition:**
Le système actuel **n'utilise PAS de transaction** ni de **verrou de base de données**, ce qui crée un risque de **double booking**.

Scénario de double booking:
1. Utilisateur A et B cliquent simultanément sur "Réserver"
2. Les deux requêtes vérifient `callOffer.booking` → `null` (disponible)
3. Les deux créent un booking
4. Le dernier écrase le premier

Solution recommandée: Utiliser des transactions Prisma avec `update` conditionnel ou ajout d'un champ `isBooked` avec contrainte unique.

2. Statuts de booking

```
enum BookingStatus {
  PENDING    // En attente de paiement
  CONFIRMED  // Paiement confirmé
  COMPLETED  // Appel terminé
  CANCELLED  // Annulé
}
```

3. Intégration avec Daily.io

Après confirmation du paiement:
- Création automatique d'une room Daily.io (si pas déjà créée)
- Stockage de `dailyRoomUrl` et `dailyRoomName` dans le booking
- Room expirée automatiquement après la date prévue + 24h

Modèle de données Booking

```
model Booking {
  id                    String        @id @default(cuid())
  userId                String
  callOfferId           String        @unique    // ⚠️ Contrainte unique
  status                BookingStatus @default(PENDING)
  totalPrice            Decimal       @db.Decimal(10, 2)
  stripePaymentIntentId String?
  dailyRoomUrl          String?
  dailyRoomName         String?
  isTestBooking         Boolean       @default(false)
  createdAt             DateTime      @default(now())
  updatedAt             DateTime      @updatedAt

  user      User       @relation(...)
  callOffer CallOffer  @relation(...)
  payment   Payment?
  review    Review?
}
```

Modèle CallOffer

```
model CallOffer {
  id          String          @id @default(cuid())
  creatorId   String
  title       String
  description String          @db.Text
  price       Decimal         @db.Decimal(10, 2)
  currency    String          @default("EUR")
  dateTime    DateTime
  duration    Int             // en minutes
  status      CallOfferStatus @default(AVAILABLE)

  creator Creator  @relation(...)
  booking Booking? // ⚠️ Relation 1-to-1
}

enum CallOfferStatus {
  AVAILABLE  // Disponible pour réservation
  BOOKED     // Réservé
  COMPLETED  // Terminé
  CANCELLED  // Annulé
}
```

Points d'attention - Booking

⚠️ Problèmes critiques identifiés:

1. **❌ PAS DE PROTECTION ANTI MULTI-BOOKING**
   - Pas de transaction lors de la création
   - Pas de verrou pessimiste
   - Race condition possible entre vérification et création

2. **❌ PAS D'AFFICHAGE "ALREADY BOOKED" SUR L'UI**
   - L'UI ne vérifie pas en temps réel si l'offre est déjà réservée
   - Risque de confusion pour l'utilisateur

3. **Gestion de la room Daily.io**
   - ✅ Création automatique après paiement
   - ❌ Pas de suppression automatique après l'appel
   - Les rooms persistent indéfiniment (coût potentiel)

4. **Expiration des offres**
   - ✅ Vérification que `dateTime` n'est pas passée
   - ⚠️ Pas de nettoyage automatique des offres expirées

Routes API

Structure des routes

```
app/api/
├── auth/
│   ├── [...nextauth]/route.ts      # NextAuth handler
│   ├── login/route.ts              # Login
│   ├── logout/route.ts             # Logout
│   └── update-profile/route.ts     # Mise à jour profil
├── bookings/
│   ├── route.ts                    # GET/POST bookings
│   ├── [id]/route.ts               # GET/PUT/DELETE booking
│   └── creator/route.ts            # Bookings du créateur
├── payments/
│   ├── create-intent/route.ts      # Création Payment Intent
│   └── webhook/route.ts            # Webhooks Stripe
├── cron/
│   ├── cleanup-logs/route.ts       # ⚠️ Nettoyage logs
│   ├── process-payouts/route.ts    # ⚠️ Payouts manuels
│   └── process-automatic-payouts/  # ⚠️ Payouts automatiques
├── daily/
│   ├── create-room/route.ts        # Création room Daily
│   └── get-token/route.ts          # Token d'accès
└── ...
```

Routes CRON

1. `/api/cron/cleanup-logs` (POST)

Fonction: Nettoyage automatique des logs selon la politique de rétention

Politique de rétention:
- INFO: 30 jours
- WARNING: 60 jours
- ERROR: 90 jours
- CRITICAL: ♾️ (jamais supprimé)

Sécurité:

```
const authHeader = request.headers.get('authorization');
const cronSecret = process.env.CRON_SECRET;

if (cronSecret && authHeader !== `Bearer ${cronSecret}`) {
  return error('Unauthorized');
}
```

⚠️ **Problèmes:**
- Secret optionnel (si `CRON_SECRET` non défini, pas de protection)
- Pas de whitelist IP

Configuration Vercel Cron recommandée:

```
{
  "crons": [{
    "path": "/api/cron/cleanup-logs",
    "schedule": "0 2 * * *"  // Tous les jours à 2h
  }]
}
```

2. `/api/cron/process-payouts` (GET)

Fonction: Traitement des payouts automatiques pour créateurs éligibles

Processus:
1. Récupère tous les créateurs avec `mode: AUTOMATIC` et `isActive: true`
2. Vérifie l'éligibilité (balance, période, KYC)
3. ⚠️ **Crée une demande avec statut REQUESTED** (attend approbation admin)
4. Envoie une notification aux admins
5. Met à jour `nextPayoutDate`

Sécurité:

```
const cronSecret = request.headers.get('x-cron-secret')
                || request.nextUrl.searchParams.get('secret');

if (cronSecret !== process.env.CRON_SECRET) {
  return error('Unauthorized');
}
```

⚠️ **Problèmes:**
- Accepte le secret en query param (risque de logs)
- Pas de gestion d'erreur pour les notifications échouées
- Les payouts auto **nécessitent toujours approbation admin** (contre-intuitif)

3. `/api/cron/process-automatic-payouts` (GET)

Fonction: Traitement des payouts pour créateurs avec calendrier DAILY/WEEKLY

Différence avec `process-payouts` :
- Celui-ci crée **directement le payout Stripe** (pas de demande REQUESTED)
- Vérifie la fréquence (daily: toutes les 24h, weekly: tous les 7 jours)
- Vérifie que `balance >= payoutMinimum`

Processus:
1. Récupère les créateurs avec `payoutSchedule` DAILY ou WEEKLY
2. Vérifie si le payout est dû (basé sur `lastPayout.createdAt`)
3. Récupère le balance Stripe
4. Crée le payout Stripe directement
5. Enregistre dans `PayoutAuditLog`

⚠️ **Problèmes:**
- Deux routes de cron différentes pour les payouts (confusion)
- Pas de retry en cas d'échec
- Pas de notification au créateur en cas de succès/échec

Sécurité des routes API

Authentification:
- NextAuth.js pour les sessions utilisateur
- Helper `getUserFromRequest(request)` dans `lib/auth.ts`

Autorisation:
- Vérification du rôle (USER, CREATOR, ADMIN)
- Vérification de propriété (ex: un créateur ne peut modifier que ses offres)

⚠️ **Problèmes de sécurité identifiés:**

1. **CRONs:**
   - Secret optionnel ou en query param
   - Pas de whitelist IP
   - Pas de rate limiting

2. **Webhooks:**
   - ✅ Vérification de signature Stripe
   - ⚠️ Pas de vérification de l'origine IP

Modèles de base de données

Le projet utilise **Prisma ORM** avec **PostgreSQL**.

Modèles principaux

User

```
model User {
  id            String   @id @default(cuid())
  email         String   @unique
  password      String?
  name          String
  role          Role     @default(USER)  // USER, CREATOR, ADMIN
  timezone      String   @default("Europe/Paris")

  creator       Creator?
  bookings      Booking[]
  callRequests  CallRequest[]
  reviews       Review[]
  notifications Notification[]
}
```

Creator

```
model Creator {
  id                String  @id @default(cuid())
  userId            String  @unique
  bio               String? @db.Text
  expertise         String?
  profileImage      String?
  bannerImage       String?
  stripeAccountId   String?
  isStripeOnboarded Boolean @default(false)

  // Devise et timezone
  currency          String  @default("EUR")
  timezone          String  @default("Europe/Paris")

  // Social links
  socialLinks       Json?   // {instagram, tiktok, twitter, youtube, other}

  // Payout settings
  payoutSchedule    PayoutSchedule @default(WEEKLY)
  payoutMinimum     Decimal        @default(10) @db.Decimal(10, 2)
  isPayoutBlocked   Boolean        @default(false)
  payoutBlockReason String?

  callOffers       CallOffer[]
  callRequests     CallRequest[]   @relation("CreatorRequests")
  reviewsReceived  Review[]        @relation("CreatorReviews")
  payouts          Payout[]
}
```

Booking

```
model Booking {
  id                    String        @id @default(cuid())
  userId                String
  callOfferId           String        @unique  // ⚠️ 1-to-1
  status                BookingStatus @default(PENDING)
  totalPrice            Decimal       @db.Decimal(10, 2)
  stripePaymentIntentId String?
  dailyRoomUrl          String?       // ⚠️ Room Daily
  dailyRoomName         String?
  isTestBooking         Boolean       @default(false)

  user      User      @relation(...)
  callOffer CallOffer @relation(...)
  payment   Payment?
  review    Review?
}
```

Payment

```
model Payment {
  id                    String        @id @default(cuid())
  bookingId             String        @unique
  amount                Decimal       @db.Decimal(10, 2)
  currency              String        @default("EUR")
  stripePaymentIntentId String
  status                PaymentStatus @default(PENDING)
  platformFee           Decimal       @db.Decimal(10, 2)
  creatorAmount         Decimal       @db.Decimal(10, 2)
  refundedAmount        Decimal       @default(0) @db.Decimal(10, 2)

  // Payout tracking
  payoutStatus          PayoutStatus  @default(REQUESTED)
  payoutReleaseDate     DateTime?
  stripeTransferId      String?
  transferId            String?
  transferStatus        String?
  payoutDate            DateTime?

  // Relations
  booking         Booking          @relation(...)
  refunds         Refund[]
  disputes        Dispute[]
  transactionLogs TransactionLog[] @relation("PaymentLogs")
}
```

Notification

```
model Notification {
  id        String           @id @default(cuid())
  userId    String
  type      NotificationType
  title     String
  message   String           @db.Text
  link      String?
  read      Boolean          @default(false)
  metadata  Json?
  createdAt DateTime         @default(now())
  readAt    DateTime?

  user User @relation(...)
}

enum NotificationType {
  BOOKING_CONFIRMED
  BOOKING_CANCELLED
  CALL_REQUEST
  REVIEW_RECEIVED
  PAYOUT_COMPLETED
  SYSTEM
  PAYMENT_RECEIVED
  PAYOUT_REQUEST
  PAYOUT_APPROVED
  PAYOUT_FAILED
  REFUND_CREATED
  DISPUTE_CREATED
  DEBT_DEDUCTED
  TRANSFER_FAILED
  DEBT_THRESHOLD_EXCEEDED
}
```

Log (System-wide logging)

```
model Log {
  id        String   @id @default(cuid())
  level     LogLevel @default(INFO)
  type      String   // ex: USER_LOGIN, BOOKING_CREATED, PAYOUT_REQUESTED
  actor     LogActor // USER, CREATOR, ADMIN, SYSTEM, GUEST
  actorId   String?
  message   String   @db.Text
  metadata  Json?
  createdAt DateTime @default(now())

  @@index([createdAt])
  @@index([level])
  @@index([type])
  @@index([actor])
  @@index([level, createdAt])  // Pour retention queries
}

enum LogLevel {
  INFO
  WARNING
  ERROR
  CRITICAL
}
```

Enums importants

```
enum BookingStatus {
  PENDING     // En attente de paiement
  CONFIRMED   // Payé et confirmé
  COMPLETED   // Appel terminé
  CANCELLED   // Annulé
}

enum PayoutStatus {
  REQUESTED   // Demandé (en attente approbation)
  APPROVED    // Approuvé par admin
  PROCESSING  // Payout Stripe en cours
  PAID        // Payé
  FAILED      // Échec
  REJECTED    // Refusé par admin
  CANCELED    // Annulé
}

enum PayoutSchedule {
  DAILY    // Payout quotidien
  WEEKLY   // Payout hebdomadaire
  MANUAL   // Sur demande
}
```

Indexes et performances

Indexes critiques pour les requêtes fréquentes:

```
@@index([userId])          // Bookings par utilisateur
@@index([status])          // Bookings par statut
@@index([isTestBooking])   // Filtrer les bookings de test
@@index([createdAt])       // Tri chronologique
@@index([level, createdAt]) // Logs avec rétention
```

⚠️ **Indexes manquants recommandés:**
- `Booking.dateTime` - Pour les requêtes par date
- `CallOffer.dateTime` - Pour les offres à venir
- `Payment.payoutReleaseDate` - Pour les payouts éligibles

Système d’emails

Architecture du système d’envoi

Fichier principal: `lib/email.ts`

Service utilisé: Resend (API moderne d’envoi d’emails)

Configuration:

```
const RESEND_API_KEY = process.env.RESEND_API_KEY;
const EMAIL_FROM = process.env.EMAIL_FROM || 'Call a Star <onboarding@resend.dev>';
```

Fonctions disponibles

1. `sendEmail({ to, subject, html })`

Fonction générique pour envoyer un email.

```
export async function sendEmail({ to, subject, html }: SendEmailOptions) {
  if (!resend) {
    console.warn('Resend client not configured. Email not sent.');
    return;
  }

  const { data, error } = await resend.emails.send({
    from: EMAIL_FROM,
    to,
    subject,
    html,
  });

  if (error) {
    console.error('Error sending email:', error);
    throw error;
  }

  console.log('Email sent:', data?.id);
  return data;
}
```

⚠️ **Problèmes:**
- ❌ Pas de retry en cas d'échec
- ❌ Pas de queue d'envoi
- ❌ Pas de logs structurés dans la BDD
- ⚠️ Logs uniquement en console

2. Templates d'emails

A. Email de confirmation de booking

```
generateBookingConfirmationEmail({
  userName,
  creatorName,
  callTitle,
  callDateTime,
  callDuration,
  totalPrice,
  callUrl,
})
```

Caractéristiques:
- Design HTML responsive
- Informations complètes du booking
- Bouton CTA "Rejoindre l'appel"
- Note: "Vous pourrez rejoindre l'appel 15 minutes avant l'heure prévue"

⚠️ **Problèmes:**
- ✅ Template en français uniquement

- ❌ Pas de version anglaise (besoin de i18n)
- ❌ Pas de logo/branding personnalisable

B. Email de rappel (reminder)

```
generateReminderEmail({
  userName,
  creatorName,
  callTitle,
  callDateTime,
  callUrl,
})
```

Utilise `formatDistanceToNow` de date-fns:

```
const timeUntilCall = formatDistanceToNow(new Date(callDateTime), {
  locale: fr,  // ⚠️ Hardcodé en français
  addSuffix: true,
});
```

⚠️ **Problèmes:**
- ✅ Template en français uniquement
- ❌ Pas de version anglaise
- ❌ Locale hardcodée en français
- ❌ Pas d'envoi automatique 15 minutes avant

Points d'attention - Emails

⚠️ **Problèmes critiques identifiés:**

1. ❌ **PAS DE CRON POUR LES RAPPELS AUTOMATIQUES**
   - Les emails de rappel existent mais ne sont jamais envoyés
   - Besoin d'un cron pour envoyer 15 minutes avant l'appel

2. ❌ **PAS DE LOGS DES EMAILS DANS LA BDD**
   - Impossible de tracker les emails envoyés/échoués
   - Pas de debugging possible
   - Pas de métriques d'envoi

3. ❌ **PAS D'INTERNATIONALISATION DES EMAILS**
   - Tous les templates sont en français
   - `locale` hardcodée dans `formatDistanceToNow`
   - Besoin de templates en/fr

4. ❌ **PAS DE RETRY EN CAS D'ÉCHEC**
   - Si Resend est down, l'email est perdu
   - Pas de queue de retry

5. ⚠️ **DESIGN DES EMAILS**
   - Design basique mais fonctionnel
   - Pas de logo personnalisable
   - Pas de footer avec liens (désabonnement, support)

Recommandations - Emails

Priorité HAUTE:

1. ✅ Créer un modèle `EmailLog` pour tracker les envois
2. ✅ Créer un cron `/api/cron/send-reminders` pour les rappels 15 min avant
3. ✅ Internationaliser tous les templates (en/fr)
4. ✅ Ajouter un système de retry avec queue

Priorité MOYENNE:

5. ⚠️ Ajouter un footer avec liens de désabonnement
6. ⚠️ Améliorer le design avec logo

Priorité BASSE:

7. 📋 Dashboard admin pour voir les logs d'emails
8. 📋 Templates personnalisables par créateur

Configuration Daily.io

Architecture de l'intégration

Fichier principal: `lib/daily.ts`

API Daily.co: `https://api.daily.co/v1`

Configuration:

```
const DAILY_API_KEY = process.env.DAILY_API_KEY;
const DAILY_API_URL = 'https://api.daily.co/v1';
```

Fonctions disponibles

1. `createDailyRoom(options)`

Crée une room Daily.co privée pour un appel.

```
interface CreateRoomOptions {
  name: string;                    // Nom unique de la room
  properties?: {
    exp?: number;                  // Timestamp d'expiration
    enable_screenshare?: boolean;
    enable_chat?: boolean;
    max_participants?: number;
  };
}
```

Paramètres par défaut:

```
{
  privacy: 'private',              // Room privée (pas publique)
  enable_screenshare: true,
  enable_chat: true,
  max_participants: 2,             // Créateur + Fan
  exp: Date.now() + 24h,          // Expiration après 24h
}
```

Retour:

```
{
  url: string,   // URL de la room (ex: https://call-a-star.daily.co/room-name)
  name: string,  // Nom de la room
}
```

Stockage dans la BDD:

```
await db.booking.update({
  where: { id: booking.id },
  data: {
    dailyRoomUrl: room.url,
    dailyRoomName: room.name,
  },
});
```

2. `createMeetingToken(options)`

Crée un token d'accès pour rejoindre une room.

```
interface CreateTokenOptions {
  roomName: string;
  userName: string;
  isOwner?: boolean;    // true pour le créateur
  exp?: number;         // Expiration (défaut: 1h)
}
```

Utilisation:
- Token créé à la demande lors de l'accès à la room
- Permet de contrôler qui peut rejoindre
- Token expiré après 1h par défaut

3. `deleteDailyRoom(roomName)`

Supprime une room Daily.co.

```
export async function deleteDailyRoom(roomName: string): Promise<void>
```

⚠️ **PROBLÈME CRITIQUE:**
- ❌ Cette fonction existe mais **n'est JAMAIS appelée**
- Les rooms persistent indéfiniment après les appels
- Coût potentiel (Daily facture par room active)

Routes API Daily.io

`/api/daily/create-room` (POST)

Fonction: Crée ou récupère la room pour un booking

Processus:
1. Vérifie l'authentification
2. Récupère le booking
3. Vérifie les permissions (créateur ou booker)
4. Si room existe déjà → retourne room existante

5. Sinon → crée nouvelle room
6. Met à jour le booking avec `dailyRoomUrl` et `dailyRoomName`

Sécurité:

```
const isCreator = booking.callOffer.creator.userId === user.userId;
const isBookingOwner = booking.userId === user.userId;

if (!isCreator && !isBookingOwner) {
  return error('Accès refusé');
}
```

`/api/daily/get-token` (POST)

Fonction: Génère un token d'accès pour rejoindre une room

⚠️ **Note:** Cette route existe probablement mais n'a pas été analysée dans les fichiers lus.

Points d'attention - Daily.io

⚠️ **Problèmes critiques identifiés:**

1. ❌ **PAS DE SUPPRESSION AUTOMATIQUE DES ROOMS**
   - `deleteDailyRoom()` existe mais n'est jamais appelée
   - Les rooms persistent après les appels
   - Coût potentiel si Daily facture par room active

2. ⚠️ **EXPIRATION DES ROOMS**
   - Expiration fixée à `exp: Date.now() + 24h`
   - Fonctionne pour la plupart des cas
   - Mais ne nettoie pas la DB (booking garde `dailyRoomUrl`)

3. ❌ **PAS DE LOGS DES APPELS DANS LA DB**
   - Impossible de tracker:
     - Qui a rejoint la room
     - Durée réelle de l'appel
     - Problèmes techniques (déconnexions)
     - Daily.co fournit des webhooks pour ça

4. ⚠️ **GESTION DES ERREURS**
   - Pas de retry si création de room échoue
   - Pas de fallback

Webhooks Daily.io (non implémentés)

Daily.co propose des webhooks pour:
- `room.created` - Room créée
- `participant.joined` - Participant rejoint
- `participant.left` - Participant quitte
- `recording.started` - Enregistrement démarré
- `recording.finished` - Enregistrement terminé

⚠️ **Ces webhooks ne sont PAS configurés dans le projet.**

Recommandations - Daily.io

Priorité HAUTE:
1. ✅ Créer un cron `/api/cron/cleanup-daily-rooms` pour supprimer les rooms après les appels
2. ✅ Implémenter les webhooks Daily.io pour tracker les appels
3. ✅ Créer un modèle `CallLog` pour stocker les logs d'appels

Priorité MOYENNE:
4. ⚠️ Ajouter un système de retry pour la création de room
5. ⚠️ Dashboard admin pour voir les appels en cours

Priorité BASSE:
6. 📋 Permettre l'enregistrement des appels (avec consentement)

Système i18n

Architecture de l'internationalisation

Library: `next-intl` (version 4.6.1)

Fichiers de configuration:
- `i18n-config.ts` - Configuration des locales
- `i18n.ts` - Configuration next-intl
- `middleware.ts` - Middleware pour la détection de locale
- `navigation.ts` - Utilitaires de navigation i18n

Locales supportées:
- `fr` - Français (par défaut)
- `en` - Anglais

Structure des fichiers de traduction

Emplacement: `messages/`
- `messages/fr.json` - Traductions françaises
- `messages/en.json` - Traductions anglaises

Structure:

```
{
  "common": {
    "appName": "Call a Star",
    "loading": "Chargement...",
    "error": "Erreur",
    ...
  },
  "navbar": {
    "creators": "Créateurs",
    "login": "Connexion",
    ...
  },
  "booking": {
    "title": "Réservation et paiement",
    "paymentSuccess": "Paiement réussi !",
    ...
  },
  "dashboard": {
    "user": { ... },
    "creator": { ... },
    "admin": { ... }
  }
}
```

Utilisation dans le code

Dans les composants:

```
import { useTranslations, useLocale } from 'next-intl';

function MyComponent() {
  const t = useTranslations('booking');
  const locale = useLocale(); // 'fr' ou 'en'

  return <h1>{t('title')}</h1>;
}
```

Dans les server components:

```
import { getTranslations } from 'next-intl/server';

async function MyServerComponent() {
  const t = await getTranslations('booking');

  return <h1>{t('title')}</h1>;
}
```

Routing i18n

URL Structure:

```
/fr/creators      → Version française
/en/creators      → Version anglaise
/fr/book/123      → Réservation en français
/en/dashboard/user → Dashboard en anglais
```

Middleware de détection:

```
// middleware.ts
export default createMiddleware({
  locales: ['fr', 'en'],
  defaultLocale: 'fr',
  localePrefix: 'always',  // Toujours préfixer l'URL
});
```

Points d'attention - i18n

⚠️ **Problèmes identifiés:**

1. ✅ **L'UI est bien internationalisée (fr/en)**
   - Tous les textes UI sont dans fr.json et en.json
   - Les composants utilisent correctement `useTranslations()`

2. ❌ **LES EMAILS NE SONT PAS INTERNATIONALISÉS**
   - `generateBookingConfirmationEmail()` est hardcodé en français
   - `generateReminderEmail()` utilise `locale: fr` hardcodé
   - Besoin de passer la locale en paramètre

3. ❌ **LES LOGS SYSTÈMES NE SONT PAS INTERNATIONALISÉS**
   - Messages de logs en français dans `system-logger.ts`
   - Messages d'erreur API en français
   - Devrait être en anglais pour cohérence internationale

4. ⚠️ **DATES ET HEURES**
   - Formatage avec `date-fns` + locale
   - Timezone support (champ `timezone` dans User/Creator)
   - Mais formatage inconsistant dans certains endroits

5. ⚠️ **DEVISES**
   - Multi-currency support (EUR, CHF, USD, GBP)
   - Composant `<CurrencyDisplay />` pour affichage
   - Mais pas de conversion de devises en temps réel

Fichiers i18n manquants

Contenu manquant dans en.json:
Après analyse, `en.json` et `fr.json` ont la **même structure complète**. Les traductions anglaises sont présentes mais identiques aux françaises (traduction incomplete).

⚠️ **Traductions anglaises à vérifier:**
- Beaucoup de textes dans en.json sont encore en français
- Besoin d'une passe de traduction complète

Recommandations - i18n

Priorité HAUTE:
1. ✅ Internationaliser les templates d'emails (passer locale en param)
2. ✅ Compléter les traductions anglaises dans en.json
3. ✅ Mettre les logs systèmes en anglais

Priorité MOYENNE:
4. ⚠️ Formatter les dates de manière cohérente partout
5. ⚠️ Afficher les devises selon la locale (ex: 100,00 € vs €100.00)

Priorité BASSE:
6. 📋 Ajouter d'autres langues (es, de, it)
7. 📋 Conversion de devises en temps réel

Composants UI

Architecture des composants

Library UI: shadcn/ui (basé sur Radix UI + Tailwind CSS)

Structure:

```
components/
├── ui/                     # Composants UI de base (shadcn)
│   ├── button.tsx
│   ├── card.tsx
│   ├── dialog.tsx
│   ├── input.tsx
│   ├── currency-display.tsx
│   ├── datetime-display.tsx
│   └── ...
├── admin/                  # Composants admin spécifiques
│   ├── FilterBar.tsx
│   ├── DataTable.tsx
│   ├── StatusBadge.tsx
│   └── ...
├── navbar.tsx             # Barre de navigation
├── footer.tsx             # Footer
├── calendar-view.tsx      # Vue calendrier
├── creator-card.tsx       # Carte créateur
└── ...
```

Composants liés au booking

1. Page de réservation (`app/[locale]/book/[offerId]/page.tsx`)

Processus:

```
// 1. Récupération de l'offre
const fetchOffer = async () => {
  const res = await fetch(`/api/call-offers/${offerId}`);
  const data = await res.json();
  setOffer(data);
};

// 2. Vérification si déjà réservé
const checkExistingBooking = async () => {
  const res = await fetch(`/api/bookings?offerId=${offerId}`);
  const data = await res.json();
  setExistingBooking(data.booking);
};

// 3. Création du booking (si pas déjà réservé)
const initializeBooking = async () => {
  const res = await fetch('/api/bookings', {
    method: 'POST',
    body: JSON.stringify({ callOfferId: offerId }),
  });
  const data = await res.json();
  setBooking(data.booking);
};

// 4. Création du Payment Intent
const res = await fetch('/api/payments/create-intent', {
  method: 'POST',
  body: JSON.stringify({ bookingId: booking.id }),
});
const { clientSecret } = await res.json();
setClientSecret(clientSecret);
```

Affichage:

```
{existingBooking ? (
  // Cas 1: Offre déjà réservée par quelqu'un d'autre
  <Alert variant="destructive">
    <AlertTitle>{t('offerUnavailable')}</AlertTitle>
    <AlertDescription>{t('offerUnavailableDesc')}</AlertDescription>
  </Alert>
) : (
  // Cas 2: Offre disponible → Formulaire de paiement Stripe
  <Elements stripe={stripePromise} options={{ clientSecret }}>
    <CheckoutForm bookingId={booking.id} onSuccess={handleSuccess} />
  </Elements>
)}
```

⚠️ **Problèmes identifiés:**

1. ❌ **PAS DE RAFRAÎCHISSEMENT EN TEMPS RÉEL**
   - Si l'offre est réservée pendant que l'utilisateur est sur la page
   - Le message "Already booked" n'apparaît pas
   - L'utilisateur peut tenter de payer

2. ❌ **PAS DE VÉRIFICATION AVANT LE PAIEMENT**
   - Le `PaymentElement` de Stripe peut être soumis même si l'offre est déjà réservée
   - Besoin d'une vérification côté serveur dans `/api/payments/create-intent`

3. ⚠️ **UX CONFUSE EN CAS D'ERREUR**
   - Si le booking échoue (double booking), l'utilisateur voit juste une erreur
   - Pas de redirection automatique vers la liste des offres

2. Formulaire de paiement (`CheckoutForm`)

Utilise Stripe Elements:

```
function CheckoutForm({ bookingId, onSuccess }) {
  const stripe = useStripe();
  const elements = useElements();

  const handleSubmit = async (e) => {
    e.preventDefault();

    const { error } = await stripe.confirmPayment({
      elements,
      confirmParams: {
        return_url: `${window.location.origin}/dashboard/user`,
      },
      redirect: 'if_required',
    });

    if (error) {
      toast.error(error.message);
    } else {
      toast.success('Paiement réussi !');
      onSuccess();
    }
  };
}
```

⚠️ **Problème:**
- Pas de vérification si l'offre est toujours disponible avant confirmation
- Besoin d'une vérification serveur dans le webhook Stripe

3. Liste des créateurs (`app/[locale]/creators/page.tsx`)

Affichage:

```
<div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 gap-6">
  {creators.map(creator => (
    <CreatorCard key={creator.id} creator={creator} />
  ))}
</div>
```

4. Carte créateur (`components/creator-card.tsx`)

Affichage:
- Photo de profil
- Nom
- Bio courte
- Nombre d'offres disponibles
- Lien vers profil

5. Vue calendrier (`components/calendar-view.tsx`)

Library: FullCalendar (React)

Fonctionnalités:
- Vue jour/semaine/mois
- Affichage des offres disponibles
- Affichage des bookings confirmés
- Clic sur un événement → Détails

⚠️ **Non utilisé actuellement dans l'UI principale.**

Composants de statut

StatusBadge (`components/admin/StatusBadge.tsx`)

Affichage des statuts avec couleurs:

```
<Badge variant={
  status === 'CONFIRMED' ? 'success' :
  status === 'PENDING' ? 'warning' :
  status === 'CANCELLED' ? 'destructive' :
  'default'
}>
  {status}
</Badge>
```

Recommandations - Composants UI

Priorité HAUTE:
1. ✅ Ajouter un polling/WebSocket pour rafraîchir le statut des offres en temps réel
2. ✅ Ajouter une vérification serveur avant paiement dans `/api/payments/create-intent`
3. ✅ Afficher un message "Already booked" clair sur la page de réservation

Priorité MOYENNE:
4. ⚠️ Améliorer l'UX en cas d'erreur (redirection automatique)
5. ⚠️ Utiliser la vue calendrier dans le dashboard créateur

Priorité BASSE:
6. 📋 Ajouter des animations de transition
7. 📋 Mode sombre (dark mode)

CRONs existants

Vue d'ensemble

Le projet utilise **3 routes CRON** différentes pour automatiser les tâches:

```
/api/cron/cleanup-logs              → Nettoyage logs (quotidien)
/api/cron/process-payouts           → Payouts manuels (quotidien)
/api/cron/process-automatic-payouts → Payouts auto DAILY/WEEKLY (quotidien)
```

⚠️ **Problème d'architecture:**
- 2 routes différentes pour les payouts (confusion)
- Logique similaire mais comportement différent
- Devrait être unifié

1. Cleanup Logs (`/api/cron/cleanup-logs`)

Fréquence recommandée: Quotidien (2h du matin)

Fonction:
- Supprime les logs INFO > 30 jours
- Supprime les logs WARNING > 60 jours
- Supprime les logs ERROR > 90 jours
- Conserve les logs CRITICAL à vie

Implémentation:

```
export async function POST(request: NextRequest) {
  // Vérification du secret
  const authHeader = request.headers.get('authorization');
  const cronSecret = process.env.CRON_SECRET;

  if (cronSecret && authHeader !== `Bearer ${cronSecret}`) {
    return error('Unauthorized');
  }

  // Nettoyage
  const result = await deleteLogsByRetention();

  // Log du nettoyage
  await logSystem('LOG_CLEANUP', `${result.totalDeleted} logs deleted`, ...);

  return { success: true, stats: result };
}
```

Configuration Vercel Cron:

```
{
  "crons": [{
    "path": "/api/cron/cleanup-logs",
    "schedule": "0 2 * * *"
  }]
}
```

⚠️ **Problèmes:**
- Secret optionnel (si `CRON_SECRET` non défini)
- Pas de whitelist IP
- Pas de retry en cas d'échec

2. Process Payouts (`/api/cron/process-payouts`)

Fréquence recommandée: Quotidien (2h du matin)

Fonction:
- Trouve les créateurs avec `mode: AUTOMATIC` et `isActive: true`
- Vérifie si `nextPayoutDate <= now`
- Vérifie l'éligibilité (balance, KYC, etc.)
- **Crée une demande REQUESTED** (attend approbation admin)
- Envoie notification aux admins
- Met à jour `nextPayoutDate`

⚠️ **Comportement contre-intuitif:**
- Appelé “automatic” mais **nécessite approbation admin**
- Devrait plutôt s’appeler “process-scheduled-payouts”

Problèmes identifiés:

1. ❌ **Les payouts “automatiques” ne sont pas vraiment automatiques**
   - Status: REQUESTED → Besoin d’approbation
   - Frustrant pour les créateurs qui ont configuré AUTOMATIC
2. ❌ **Accepte le secret en query param**
   ```typescript
   const cronSecret = request.headers.get('x-cron-secret')
                      || request.nextUrl.searchParams.get('secret');
   ```
   - Risque de logs du secret dans les logs serveur
3. ❌ **Pas de retry en cas d’échec**
   - Si un créateur échoue, les autres sont traités
   - Mais pas de retry automatique
4. ⚠️ **Notifications aux admins non critiques**
   - Si l’envoi de notification échoue, continue quand même
   - Mais les admins ne sont pas notifiés (perte d’info)

3. Process Automatic Payouts (`/api/cron/process-automatic-payouts`)

Fréquence recommandée: Quotidien (2h du matin)

Fonction:
- Trouve les créateurs avec `payoutSchedule` DAILY ou WEEKLY
- Vérifie si le payout est dû (basé sur `lastPayout.createdAt`)
- Vérifie le balance Stripe
- **Crée directement le payout Stripe** (pas de REQUESTED)
- Enregistre dans `PayoutAuditLog`

Différences avec `process-payouts` :
- Celui-ci crée **directement** le payout (pas d’approbation)
- Utilise `payoutSchedule` (User-facing) au lieu de `PayoutScheduleNew` (System-facing)
- Vérifie manuellement la fréquence (hoursSinceLast >= 24 pour DAILY)

⚠️ **Problèmes:**

1. ❌ **Duplication de logique**
   - 2 routes qui font presque la même chose
   - Code dupliqué pour vérifier éligibilité, balance, KYC
2. ❌ **Confusion sur les payouts automatiques**
   - `process-payouts` → AUTOMATIC mais besoin approbation
   - `process-automatic-payouts` → DAILY/WEEKLY et vraiment auto
   - Devrait être unifié
3. ❌ **Pas de notification au créateur**
   - Si payout réussit, pas de notification
   - Si payout échoue, pas de notification non plus

4. ⚠️ **Calcul manuel de la fréquence**
   `typescript`
   `if (creator.payoutSchedule === 'DAILY') {`
   `const hoursSinceLast = (now - lastPayoutDate) / (1000 * 60 * 60);`
   `isDue = hoursSinceLast >= 24;`
   `}`
   - Devrait utiliser `PayoutScheduleNew.nextPayoutDate` pour éviter drift

CRONs manquants

⚠️ CRONs critiques à implémenter:

1. ❌ **`/api/cron/send-reminders` (EMAIL REMINDERS)**
   - Envoyer un email 15 minutes avant chaque appel
   - Template `generateReminderEmail()` existe mais n'est jamais utilisé
   - **PRIORITÉ HAUTE**

2. ❌ **`/api/cron/cleanup-daily-rooms` (DAILY.IO CLEANUP)**
   - Supprimer les rooms Daily.io après les appels
   - `deleteDailyRoom()` existe mais n'est jamais appelée
   - **PRIORITÉ HAUTE**

3. ⚠️ **`/api/cron/expire-offers` (OFFRES EXPIRÉES)**
   - Marquer les offres expirées comme CANCELLED
   - Libérer les bookings PENDING expirés
   - **PRIORITÉ MOYENNE**

4. ⚠️ **`/api/cron/send-daily-summary` (RÉSUMÉ QUOTIDIEN)**
   - Envoyer un résumé des bookings/payouts aux admins
   - **PRIORITÉ BASSE**

Configuration Vercel Cron recommandée

```
{
  "crons": [
    {
      "path": "/api/cron/cleanup-logs",
      "schedule": "0 2 * * *"
    },
    {
      "path": "/api/cron/process-payouts",
      "schedule": "0 3 * * *"
    },
    {
      "path": "/api/cron/send-reminders",
      "schedule": "*/5 * * * *"
    },
    {
      "path": "/api/cron/cleanup-daily-rooms",
      "schedule": "0 4 * * *"
    },
    {
      "path": "/api/cron/expire-offers",
      "schedule": "0 1 * * *"
    }
  ]
}
```

⚠️ **Ce fichier `vercel.json` existe déjà mais ne contient PAS de configuration CRON.**

Recommandations - CRONs

Priorité HAUTE:

1. ✅ Créer `/api/cron/send-reminders` pour emails 15 min avant
2. ✅ Créer `/api/cron/cleanup-daily-rooms` pour suppression rooms
3. ✅ Unifier les 2 routes de payouts en une seule
4. ✅ Ajouter configuration CRON dans `vercel.json`

Priorité MOYENNE:

5. ⚠️ Créer `/api/cron/expire-offers` pour offres expirées
6. ⚠️ Implémenter système de retry pour tous les CRONs
7. ⚠️ Améliorer la sécurité (whitelist IP, pas de secret en query)

Priorité BASSE:

8. 📋 Dashboard admin pour voir les exécutions CRON
9. 📋 Alertes en cas d'échec de CRON

Points d'attention critiques

🔴 Problèmes CRITIQUES (Priorité HAUTE)

1. ANTI MULTI-BOOKING (Sécurité booking)

Problème:

```
// app/api/bookings/route.ts
// ❌ Race condition possible - Pas de transaction atomique

const callOffer = await db.callOffer.findUnique({
  where: { id },
  include: { booking: true }
});

if (callOffer.booking) {
  return error('Déjà réservé');
}

// 🚨 PROBLÈME: Entre cette vérification et la création,
// un autre utilisateur peut réserver l'offre

const booking = await db.booking.create({
  data: { callOfferId: id, ... }
});
```

Impact:
- 🚨 Deux utilisateurs peuvent réserver la même offre simultanément
- 🚨 Paiements doubles pour le même créneau
- 🚨 Confusion et frustration des utilisateurs

Solutions possibles:

Option A: Transaction Prisma avec update conditionnel

```
const booking = await db.$transaction(async (tx) => {
  // Vérouiller et mettre à jour l'offre
  const updatedOffer = await tx.callOffer.updateMany({
    where: {
      id: callOfferId,
      status: 'AVAILABLE',
      booking: { is: null }
    },
    data: { status: 'BOOKED' }
  });

  if (updatedOffer.count === 0) {
    throw new Error('Offre déjà réservée');
  }

  // Créer le booking
  return await tx.booking.create({
    data: { callOfferId, userId, ... }
  });
});
```

Option B: Verrou de base de données (SELECT FOR UPDATE)

```
await db.$executeRaw`
  SELECT * FROM "CallOffer"
  WHERE id = ${callOfferId}
  FOR UPDATE
`;
```

Option C: Ajouter un champ `isBooked` avec contrainte unique

```
model CallOffer {
  isBooked  Boolean @default(false)

  @@unique([id, isBooked]) // ⚠️ Complexe à gérer
}
```

Recommandation: Option A (Transaction + updateMany conditionnel)

2. EMAIL REMINDERS (Emails 15 min avant)

Problème:

```
// lib/email.ts
// ✅ Le template existe
export function generateReminderEmail({ ... }) { ... }

// ❌ Mais n'est JAMAIS appelé automatiquement
```

Impact:
- 🚨 Les utilisateurs n’ont pas de rappel avant leur appel
- 🚨 Risque d’oubli et de no-show
- 🚨 Mauvaise expérience utilisateur

Solution:

```
// À créer: /api/cron/send-reminders

export async function GET(request: NextRequest) {
  // 1. Trouver tous les bookings dans les 15 prochaines minutes
  const in15Min = new Date(Date.now() + 15 * 60 * 1000);
  const in10Min = new Date(Date.now() + 10 * 60 * 1000);

  const bookings = await db.booking.findMany({
    where: {
      status: 'CONFIRMED',
      callOffer: {
        dateTime: {
          gte: in10Min,
          lte: in15Min,
        }
      },
      // ⚠️ Ajouter un champ pour tracker si reminder envoyé
      reminderSent: false,
    },
    include: {
      user: true,
      callOffer: { include: { creator: { include: { user: true } } } }
    }
  });

  // 2. Envoyer les emails
  for (const booking of bookings) {
    await sendEmail({
      to: booking.user.email,
      subject: 'Rappel : Votre appel commence bientôt !',
      html: generateReminderEmail({
        userName: booking.user.name,
        creatorName: booking.callOffer.creator.user.name,
        callTitle: booking.callOffer.title,
        callDateTime: booking.callOffer.dateTime,
        callUrl: `${process.env.NEXTAUTH_URL}/call/${booking.id}`,
      })
    });

    // 3. Marquer comme envoyé
    await db.booking.update({
      where: { id: booking.id },
      data: { reminderSent: true }
    });
  }
}
```

Besoins:
- ✅ Ajouter un champ `reminderSent` dans le modèle `Booking`
- ✅ Créer la route `/api/cron/send-reminders`
- ✅ Configurer le CRON pour exécuter toutes les 5 minutes
- ✅ Internationaliser les emails (en/fr)

3. DAILY.IO CLEANUP (Suppression rooms après appels)

Problème:

```
// lib/daily.ts
// ✅ La fonction existe
export async function deleteDailyRoom(roomName: string) { ... }

// ❌ Mais n'est JAMAIS appelée
```

Impact:
- 🚨 Les rooms persistent indéfiniment après les appels
- 🚨 Coût potentiel (Daily facture par room active)
- 🚨 Clutter dans le dashboard Daily.io

Solution:

```
// À créer: /api/cron/cleanup-daily-rooms

export async function GET(request: NextRequest) {
  // 1. Trouver tous les bookings terminés avec room Daily
  const bookings = await db.booking.findMany({
    where: {
      status: 'COMPLETED',
      dailyRoomName: { not: null },
      callOffer: {
        dateTime: {
          // Appels terminés il y a plus de 1h
          lt: new Date(Date.now() - 60 * 60 * 1000)
        }
      }
    }
  });

  // 2. Supprimer les rooms Daily
  for (const booking of bookings) {
    try {
      await deleteDailyRoom(booking.dailyRoomName);

      // 3. Mettre à jour le booking
      await db.booking.update({
        where: { id: booking.id },
        data: {
          dailyRoomUrl: null,
          dailyRoomName: null,
        }
      });

      console.log(`✓ Deleted room: ${booking.dailyRoomName}`);
    } catch (error) {
      console.error(`✗ Failed to delete room: ${booking.dailyRoomName}`, error);
    }
  }
}
```

Configuration:

```
{
  "path": "/api/cron/cleanup-daily-rooms",
  "schedule": "0 4 * * *"  // Tous les jours à 4h
}
```

4. LOGS DES EMAILS (Traçabilité et debugging)

Problème:

```
// lib/email.ts
// ❌ Pas de logs en base de données
export async function sendEmail({ to, subject, html }) {
  const { data, error } = await resend.emails.send(...);

  if (error) {
    console.error('Error sending email:', error); // ⚠️ Console uniquement
    throw error;
  }

  console.log('Email sent:', data?.id); // ⚠️ Console uniquement
  return data;
}
```

Impact:
- 🚨 Impossible de savoir si un email a été envoyé
- 🚨 Impossible de debugger les problèmes d'envoi
- 🚨 Pas de métriques d'envoi

Solution:

```
// À ajouter dans schema.prisma

model EmailLog {
  id          String   @id @default(cuid())
  to          String
  subject     String
  template    String   // 'booking_confirmation', 'reminder', etc.
  status      String   // 'sent', 'failed', 'pending'
  resendId    String?  // ID Resend
  error       String?  @db.Text
  metadata    Json?    // bookingId, locale, etc.
  createdAt   DateTime @default(now())
  sentAt      DateTime?

  @@index([to])
  @@index([status])
  @@index([createdAt])
}
```

```
// Mise à jour de lib/email.ts
export async function sendEmail({ to, subject, html, template, metadata }) {
  // 1. Créer le log PENDING
  const emailLog = await db.emailLog.create({
    data: {
      to,
      subject,
      template,
      status: 'pending',
      metadata,
    }
  });

  try {
    // 2. Envoyer l'email
    const { data, error } = await resend.emails.send({
      from: EMAIL_FROM,
      to,
      subject,
      html,
    });

    if (error) {
      // 3a. Log l'échec
      await db.emailLog.update({
        where: { id: emailLog.id },
        data: {
          status: 'failed',
          error: error.message,
        }
      });
      throw error;
    }

    // 3b. Log le succès
    await db.emailLog.update({
      where: { id: emailLog.id },
      data: {
        status: 'sent',
        resendId: data.id,
        sentAt: new Date(),
      }
    });

    return data;
  } catch (error) {
    // Log en cas d'erreur critique
    await logError('EMAIL_SEND_FAILED', LogActor.SYSTEM, ...);
    throw error;
  }
}
```

Avantages:

- ✅ Traçabilité complète des emails
- ✅ Debugging facile
- ✅ Métriques d’envoi
- ✅ Possibilité de retry

🟠 Problèmes IMPORTANTS (Priorité MOYENNE)

5. AFFICHAGE “ALREADY BOOKED” SUR L’UI

Problème:

```
// app/[locale]/book/[offerId]/page.tsx
// ⚠️ Vérification uniquement au chargement initial

useEffect(() => {
  fetchOffer();
  checkExistingBooking(); // ⚠️ Une seule fois
}, []);

// ❌ Pas de polling/WebSocket pour rafraîchir en temps réel
```

Impact:
- ⚠️ Si l’offre est réservée pendant que l’utilisateur est sur la page
- ⚠️ L’utilisateur ne voit pas que l’offre n’est plus disponible
- ⚠️ Il peut essayer de payer et avoir une erreur

Solution A: Polling

```
useEffect(() => {
  const interval = setInterval(() => {
    checkExistingBooking();
  }, 5000); // Vérifier toutes les 5 secondes

  return () => clearInterval(interval);
}, []);
```

Solution B: WebSocket (plus complexe)

```
// Utiliser Pusher/Socket.io pour notifications en temps réel
socket.on('offer-booked', (offerId) => {
  if (offerId === currentOfferId) {
    setExistingBooking(true);
  }
});
```

Solution C: Vérification serveur avant paiement

```
// /api/payments/create-intent
const callOffer = await db.callOffer.findUnique({
  where: { id: booking.callOfferId },
  include: { booking: true }
});

if (callOffer.booking && callOffer.booking.id !== booking.id) {
  return error('Cette offre a été réservée par quelqu\'un d\'autre');
}
```

Recommandation: Combiner **Solution A (Polling)** + **Solution C (Vérification serveur)**

6. INTERNATIONALISATION DES EMAILS

Problème:

```
// lib/email.ts
// ❌ Templates hardcodés en français

export function generateBookingConfirmationEmail({ ... }) {
  return `
    <h1>✨ Réservation Confirmée !</h1>
    <p>Bonjour ${userName},</p>
    ...
  `;
}

// ❌ Locale hardcodée
const timeUntilCall = formatDistanceToNow(callDateTime, {
  locale: fr, // ⚠️ Toujours français
  addSuffix: true,
});
```

Impact:
- ⚠️ Les utilisateurs anglophones reçoivent des emails en français
- ⚠️ Mauvaise expérience utilisateur
- ⚠️ Inconsistance avec l'UI (qui est bien i18n)

Solution:

```
// lib/email-templates.ts
export function generateBookingConfirmationEmail({
  userName,
  creatorName,
  callTitle,
  callDateTime,
  callDuration,
  totalPrice,
  callUrl,
  locale = 'fr', // ✅ Passer la locale en paramètre
}) {
  const t = getEmailTranslations(locale); // ✅ Fonction helper

  const formattedDate = new Date(callDateTime).toLocaleString(
    locale === 'fr' ? 'fr-FR' : 'en-US',
    { weekday: 'long', year: 'numeric', month: 'long', day: 'numeric', hour: '2-di-
git', minute: '2-digit' }
  );

  return `
    <!DOCTYPE html>
    <html>
    <body>
      <h1>${t.confirmationTitle}</h1>
      <p>${t.greeting} ${userName},</p>
      <p>${t.confirmationMessage} <strong>${creatorName}</strong>.</p>
      ...
    </body>
    </html>
  `;
}

// Helper pour récupérer les traductions
function getEmailTranslations(locale: string) {
  const translations = {
    fr: {
      confirmationTitle: '✨ Réservation Confirmée !',
      greeting: 'Bonjour',
      confirmationMessage: 'Votre réservation est confirmée ! Vous allez bientôt
pouvoir discuter avec',
      ...
    },
    en: {
      confirmationTitle: '✨ Booking Confirmed!',
      greeting: 'Hello',
      confirmationMessage: 'Your booking is confirmed! You will soon be able to chat
with',
      ...
    }
  };

  return translations[locale] || translations.fr;
}
```

Besoins:

- ✅ Créer un fichier `lib/email-translations.ts` avec les traductions
- ✅ Modifier tous les templates pour accepter `locale`
- ✅ Passer la locale depuis le User/Creator (champ `locale` à ajouter?)

7. UNIFICATION DES CRONs DE PAYOUTS

Problème:

```
// ❌ 2 routes différentes pour les payouts

// Route 1: /api/cron/process-payouts
// → Traite les PayoutScheduleNew (mode AUTOMATIC)
// → Crée une demande REQUESTED (besoin approbation)

// Route 2: /api/cron/process-automatic-payouts
// → Traite les Creator.payoutSchedule (DAILY/WEEKLY)
// → Crée directement le payout Stripe
```

Impact:

- ⚠️ Confusion sur les payouts “automatiques”
- ⚠️ Code dupliqué
- ⚠️ Maintenance difficile

Solution:

```
// Unifier en une seule route: /api/cron/process-payouts

export async function GET(request: NextRequest) {
  // 1. Trouver tous les créateurs éligibles
  const creators = await db.creator.findMany({
    where: {
      OR: [
        // Option A: PayoutScheduleNew (ancien système)
        {
          payoutScheduleNew: {
            mode: 'AUTOMATIC',
            isActive: true,
            nextPayoutDate: { lte: new Date() }
          }
        },
        // Option B: payoutSchedule (nouveau système)
        {
          payoutSchedule: { in: ['DAILY', 'WEEKLY'] },
          // ⚠️ Vérifier lastPayoutDate côté code
        }
      ],
      isPayoutBlocked: false,
      isStripeOnboarded: true,
    }
  });

  // 2. Pour chaque créateur, déterminer le comportement
  for (const creator of creators) {
    if (creator.payoutScheduleNew) {
      // Ancien système → Créer REQUESTED (besoin approbation)
      await createPayoutRequest(creator);
    } else if (creator.payoutSchedule === 'DAILY' || creator.payoutSchedule === 'WEEKL
Y') {
      // Nouveau système → Créer directement le payout
      await createAutomaticPayout(creator);
    }
  }
}
```

Recommandation: Migrer progressivement vers un seul système unifié.

🟡 Problèmes MINEURS (Priorité BASSE)

8. Sécurité des CRONs

Problèmes:

- ⚠️ Secret optionnel dans `/api/cron/cleanup-logs`
- ⚠️ Secret en query param dans `/api/cron/process-payouts`
- ⚠️ Pas de whitelist IP

Solution:

```
// Middleware CRON sécurisé
function verifyCronAuth(request: NextRequest) {
  const authHeader = request.headers.get('authorization');
  const cronSecret = process.env.CRON_SECRET;

  // 1. Vérifier le secret (OBLIGATOIRE)
  if (!cronSecret) {
    throw new Error('CRON_SECRET not configured');
  }

  if (authHeader !== `Bearer ${cronSecret}`) {
    return false;
  }

  // 2. Vérifier l'IP (optionnel mais recommandé)
  const ip = request.headers.get('x-forwarded-for') || request.ip;
  const allowedIPs = process.env.CRON_ALLOWED_IPS?.split(',') || [];

  if (allowedIPs.length > 0 && !allowedIPs.includes(ip)) {
    return false;
  }

  return true;
}
```

9. Retry système pour les emails échoués

Problème:

```
// ❌ Si Resend est down, l'email est perdu définitivement
```

Solution:

```
// Ajouter un CRON pour retry les emails échoués
export async function GET() {
  const failedEmails = await db.emailLog.findMany({
    where: {
      status: 'failed',
      createdAt: { gte: new Date(Date.now() - 24 * 60 * 60 * 1000) } // Dernières 24h
    }
  });

  for (const emailLog of failedEmails) {
    try {
      // Retry l'envoi
      await sendEmail({ ... });
    } catch (error) {
      console.error('Retry failed:', error);
    }
  }
}
```

Recommandations pour les prochaines étapes

Phase 1: Sécurité & Fiabilité (1-2 semaines)

Objectif: Corriger les problèmes critiques de sécurité et de fiabilité.

1.1 Anti Multi-Booking

Priorité: 🔴 CRITIQUE

Tâches:
- [ ] Implémenter transaction Prisma dans `/api/bookings/route.ts`
- [ ] Ajouter tests de charge pour simuler race conditions
- [ ] Ajouter logging des tentatives de double booking

Code à modifier:

```
// app/api/bookings/route.ts
const booking = await db.$transaction(async (tx) => {
  const updatedOffer = await tx.callOffer.updateMany({
    where: {
      id: callOfferId,
      status: 'AVAILABLE',
      booking: { is: null }
    },
    data: { status: 'BOOKED' }
  });

  if (updatedOffer.count === 0) {
    throw new Error('Cette offre est déjà réservée');
  }

  return await tx.booking.create({
    data: { callOfferId, userId, totalPrice, status: 'PENDING' }
  });
});
```

Tests:

```
// Simuler 2 requêtes simultanées
const [result1, result2] = await Promise.all([
  createBooking(offerId, user1),
  createBooking(offerId, user2)
]);

// Une seule devrait réussir
expect(result1.success XOR result2.success).toBe(true);
```

1.2 Email Reminders

Priorité: 🔴 CRITIQUE

Tâches:
- [ ] Ajouter champ `reminderSent` dans modèle `Booking`
- [ ] Créer route `/api/cron/send-reminders`
- [ ] Internationaliser les templates d'emails
- [ ] Configurer CRON dans `vercel.json`
- [ ] Tester l'envoi automatique

Migration Prisma:

```
model Booking {
  // ... existant
  reminderSent Boolean @default(false)
}
```

CRON Configuration:

```
{
  "crons": [{
    "path": "/api/cron/send-reminders",
    "schedule": "*/5 * * * *"  // Toutes les 5 minutes
  }]
}
```

1.3 Daily.io Cleanup

Priorité: 🔴 CRITIQUE

Tâches:
- [ ] Créer route `/api/cron/cleanup-daily-rooms`
- [ ] Implémenter la logique de suppression des rooms
- [ ] Ajouter logging des suppressions
- [ ] Configurer CRON dans `vercel.json`

Code:

```
// app/api/cron/cleanup-daily-rooms/route.ts
export async function GET(request: NextRequest) {
  const bookings = await db.booking.findMany({
    where: {
      status: 'COMPLETED',
      dailyRoomName: { not: null },
      callOffer: {
        dateTime: { lt: new Date(Date.now() - 60 * 60 * 1000) } // 1h passé
      }
    }
  });

  for (const booking of bookings) {
    await deleteDailyRoom(booking.dailyRoomName);
    await db.booking.update({
      where: { id: booking.id },
      data: { dailyRoomUrl: null, dailyRoomName: null }
    });
  }
}
```

1.4 Email Logging

Priorité: 🟠 IMPORTANTE

Tâches:

- [ ] Créer modèle `EmailLog` dans Prisma
- [ ] Modifier `lib/email.ts` pour logger tous les envois
- [ ] Créer une page admin pour voir les logs d'emails
- [ ] Implémenter retry pour emails échoués

Migration Prisma:

```
model EmailLog {
  id          String   @id @default(cuid())
  to          String
  subject     String
  template    String
  status      String
  resendId    String?
  error       String?  @db.Text
  metadata    Json?
  createdAt   DateTime @default(now())
  sentAt      DateTime?

  @@index([to])
  @@index([status])
  @@index([createdAt])
}
```

Phase 2: UX & Internationalisation (1-2 semaines)

Objectif: Améliorer l'expérience utilisateur et l'i18n.

2.1 Affichage "Already Booked"

Priorité: 🟠 IMPORTANTE

Tâches:
- [ ] Implémenter polling toutes les 5 secondes sur la page de booking
- [ ] Ajouter vérification serveur dans `/api/payments/create-intent`
- [ ] Améliorer l'UI pour afficher un message clair
- [ ] Rediriger automatiquement vers la liste des offres

Code:

```
// app/[locale]/book/[offerId]/page.tsx
useEffect(() => {
  const interval = setInterval(async () => {
    const res = await fetch(`/api/call-offers/${offerId}`);
    const data = await res.json();

    if (data.offer.status !== 'AVAILABLE') {
      setOfferUnavailable(true);
      setTimeout(() => router.push('/creators'), 3000);
    }
  }, 5000);

  return () => clearInterval(interval);
}, []);
```

2.2 Internationalisation des emails

Priorité: 🟠 IMPORTANTE

Tâches:
- [ ] Créer `lib/email-translations.ts` avec traductions en/fr
- [ ] Modifier tous les templates pour accepter `locale`
- [ ] Ajouter champ `locale` dans User (si pas déjà présent)
- [ ] Tester l'envoi d'emails en français et anglais

Traductions:

```
// lib/email-translations.ts
export const emailTranslations = {
  fr: {
    bookingConfirmation: {
      title: '✨ Réservation Confirmée !',
      greeting: 'Bonjour',
      message: 'Votre réservation est confirmée !',
      ...
    },
    reminder: {
      title: '⏰ Rappel : Votre appel commence bientôt !',
      ...
    }
  },
  en: {
    bookingConfirmation: {
      title: '✨ Booking Confirmed!',
      greeting: 'Hello',
      message: 'Your booking is confirmed!',
      ...
    },
    reminder: {
      title: '⏰ Reminder: Your call starts soon!',
      ...
    }
  }
};
```

2.3 Compléter les traductions anglaises

Priorité: 🟠 IMPORTANTE

Tâches:
- [ ] Passer en revue `messages/en.json`
- [ ] Traduire tous les textes encore en français
- [ ] Tester l'application complète en anglais

Phase 3: Optimisations & Dashboard Admin (1-2 semaines)

Objectif: Ajouter des fonctionnalités d'administration et optimiser les performances.

3.1 Dashboard Email Logs

Priorité: 🟡 MINEURE

Tâches:
- [ ] Créer une page admin `/dashboard/admin/email-logs`
- [ ] Afficher la liste des emails avec filtres (status, date, destinataire)
- [ ] Permettre de voir les détails d'un email (contenu HTML, erreur)
- [ ] Ajouter un bouton "Retry" pour les emails échoués

3.2 Dashboard CRON Executions

Priorité: 🟡 MINEURE

Tâches:
- [ ] Créer un modèle `CronExecution` pour logger les exécutions
- [ ] Créer une page admin `/dashboard/admin/cron-jobs`

- [ ] Afficher l'historique des exécutions (succès/échec, durée)
- [ ] Ajouter des alertes en cas d'échec répété

3.3 Unification des CRONs de Payouts

Priorité: 🟡 MINEURE

Tâches:
- [ ] Analyser l'utilisation actuelle de `PayoutScheduleNew` vs `Creator.payoutSchedule`
- [ ] Décider d'une architecture unifiée
- [ ] Migrer les données si nécessaire
- [ ] Supprimer l'un des deux CRONs

Phase 4: Tests & Documentation (En continu)

Objectif: S'assurer de la qualité et de la maintenabilité.

4.1 Tests

Tâches:
- [ ] Tests unitaires pour les fonctions critiques (anti multi-booking, payouts)
- [ ] Tests d'intégration pour les CRONs
- [ ] Tests de charge pour la page de booking
- [ ] Tests e2e pour le flux complet de réservation

4.2 Documentation

Tâches:
- [ ] Documenter les CRONs (fréquence, fonction, dépendances)
- [ ] Documenter l'architecture des paiements
- [ ] Créer un guide de déploiement
- [ ] Documenter les variables d'environnement

Timeline estimé

```
Semaine 1-2: Phase 1 (Sécurité & Fiabilité)
  - Anti multi-booking
  - Email reminders
  - Daily.io cleanup
  - Email logging

Semaine 3-4: Phase 2 (UX & Internationalisation)
  - Affichage "Already Booked"
  - Internationalisation emails
  - Traductions anglaises

Semaine 5-6: Phase 3 (Optimisations & Dashboard)
  - Dashboard email logs
  - Dashboard CRON
  - Unification payouts

En continu: Phase 4 (Tests & Documentation)
```

Priorisation

🔴 **URGENT (Semaine 1):**
1. Anti multi-booking

2. Email reminders CRON
3. Daily.io cleanup CRON

🟠 **IMPORTANT (Semaine 2-3):**
4. Email logging
5. Affichage “Already Booked”
6. Internationalisation emails

🟡 **SOUHAITABLE (Semaine 4+):**
7. Dashboard admin
8. Unification CRONs
9. Tests complets

Conclusion

Le projet Call a Star est bien structuré avec une architecture solide basée sur Next.js, Stripe Connect, et Daily.io. Cependant, plusieurs problèmes critiques de sécurité et de fiabilité doivent être corrigés en priorité:

Points forts:
- ✅ Architecture Next.js 14 moderne avec App Router
- ✅ Intégration Stripe Connect bien implémentée
- ✅ Système de logs structuré (Prisma + system-logger)
- ✅ UI bien organisée avec shadcn/ui
- ✅ Support multi-devises (EUR, CHF, USD, GBP)

Points faibles critiques:
- ❌ Pas de protection anti multi-booking (race condition)
- ❌ Pas d’envoi automatique d’emails de rappel
- ❌ Pas de nettoyage automatique des rooms Daily.io
- ❌ Pas de logs des emails en base de données
- ❌ Emails non internationalisés

Priorités:
1. 🔴 Implémenter la transaction atomique pour les bookings
2. 🔴 Créer le CRON d’emails de rappel
3. 🔴 Créer le CRON de nettoyage Daily.io
4. 🟠 Ajouter le logging des emails
5. 🟠 Internationaliser les emails

En suivant le plan proposé, ces problèmes peuvent être résolus en 4-6 semaines avec un impact majeur sur la sécurité, la fiabilité et l’expérience utilisateur de la plateforme.