🛡️ Anti Multi-Booking - Documentation

📋 Table des matières

🔴 Problème

Description du problème

Le système de booking avait un **problème critique de race condition** qui permettait à plusieurs utilisateurs de réserver le même créneau simultanément.

Scénario de la race condition

```
Temps    User A                          User B
T0       → Vérifie si booking existe
T1       ← Pas de booking trouvé         → Vérifie si booking existe
T2       → Crée le booking               ← Pas de booking trouvé
T3       ← Booking créé avec succès      → Crée le booking
T4                                       ← Booking créé avec succès ❌

Résultat: 2 bookings pour le même créneau ! 💥
```

Conséquences

- ❌ Plusieurs utilisateurs dans le même appel vidéo
- ❌ Problèmes de paiement (plusieurs Payment Intents pour le même créneau)
- ❌ Mauvaise expérience utilisateur
- ❌ Potentiel conflit de revenus pour les créateurs

Code problématique (avant)

```
// ❌ NON-ATOMIQUE - RACE CONDITION
// Vérification
const callOffer = await db.callOffer.findUnique({
  where: { id: callOfferId },
  include: { booking: true }
});

// ⚠️ PROBLÈME: Entre cette vérification et la création,
// un autre utilisateur peut créer un booking !

if (callOffer.booking) {
  return { error: 'Already booked' };
}

// Création (trop tard !)
const booking = await db.booking.create({
  data: { ... }
});
```

✅ Solution implémentée

1. Transaction atomique Prisma

La solution utilise `db.$transaction()` pour garantir que la vérification et la création du booking se font dans une **opération atomique**.

```
// ✅ ATOMIQUE - AUCUNE RACE CONDITION
const result = await db.$transaction(async (tx) => {
  // Étape 1: Vérifier l'offre
  const callOffer = await tx.callOffer.findUnique({
    where: { id: callOfferId },
    include: { booking: true }
  });

  // Étape 2: Valider la disponibilité
  if (callOffer.booking) {
    throw new Error('OFFER_ALREADY_BOOKED');
  }

  // Étape 3: Créer le booking (atomiquement)
  const booking = await tx.booking.create({
    data: { userId, callOfferId, ... }
  });

  // Étape 4: Mettre à jour le statut
  await tx.callOffer.update({
    where: { id: callOfferId },
    data: { status: 'BOOKED' }
  });

  return { booking, callOffer };
});
```

2. Contrainte unique au niveau base de données

Le schéma Prisma définit une **contrainte unique** sur `callOfferId` dans le modèle `Booking` :

```
model Booking {
  id          String @id @default(cuid())
  callOfferId String @unique  // ✅ GARANTIE BASE DE DONNÉES
  // ...
}
```

PostgreSQL applique cette contrainte et rejette toute tentative de créer un deuxième booking avec le même `callOfferId`.

3. Gestion d'erreur Prisma P2002

Si deux requêtes atteignent simultanément `booking.create()`, PostgreSQL rejette la deuxième avec une erreur **P2002 (Unique constraint violation)**.

```
// ✅ Gestion de l'erreur de contrainte unique
if (error && typeof error === 'object' && 'code' in error && error.code === 'P2002') {
  return NextResponse.json(
    { error: 'This time slot is already booked. Please choose another time.' },
    { status: 409 } // HTTP 409 Conflict
  );
}
```

4. HTTP 409 Conflict

L'API retourne maintenant un **statut HTTP 409 (Conflict)** avec un message clair quand un créneau est déjà réservé:

```
{
  "error": "This time slot is already booked. Please choose another time."
}
```

🏗️ Architecture technique

Flux de booking sécurisé

```
┌─────────────┐
│  User A     │
│  User B     │  (Requêtes simultanées)
│  User C     │
└─────────────┘
       │
       ↓
┌───────────────────────────────────┐
│  POST /api/bookings               │
│  ✅ Authentication check          │
└───────────────────────────────────┘
       │
       ↓
┌───────────────────────────────────┐
│  db.$transaction() - ATOMIQUE     │
│                                   │
│  1. findUnique(callOffer)         │
│  2. Validate availability         │
│  3. Check existing booking  ← LOCK│
│  4. Create booking          ← LOCK│
│  5. Update callOffer status ← LOCK│
│                                   │
└───────────────────────────────────┘
       │
       ├─→ SUCCESS (1 user) ✅
       │   → HTTP 201 + booking data
       │
       └─→ CONFLICT (others) ⚠️
           → HTTP 409 + error message
```

Niveaux de protection

Niveau	Protection	Description
1. Application	Transaction Prisma	Vérifie et crée atomiquement
2. Base de données	Contrainte UNIQUE	PostgreSQL rejette les doublons
3. Détection d'erreur	Code Prisma P2002	Catch et retourne HTTP 409

Isolation des transactions PostgreSQL

Prisma utilise le niveau d'isolation **READ COMMITTED** de PostgreSQL par défaut, ce qui garantit:

- ✅ Aucune lecture de données non-committées
- ✅ Les writes sont bloquants (pas de concurrent writes sur la même row)
- ✅ La contrainte UNIQUE est toujours respectée

🎯 Garanties fournies

1. Garantie d'unicité ✅

Une seule réservation par créneau, point.

- Transaction atomique empêche les race conditions
- Contrainte UNIQUE au niveau base de données
- Impossible d'avoir 2 bookings avec le même `callOfferId`

2. Garantie de paiement ✅

Un seul Payment Intent Stripe par booking.

Le Payment Intent est créé dans un endpoint séparé **APRÈS** la création du booking:

```
POST /api/bookings          → Crée le booking (atomique)
     ↓
POST /api/payments/create-intent → Crée le Payment Intent
     ↓
     Vérifie que le booking existe
     Vérifie que l'utilisateur est propriétaire
     Vérifie que le booking n'est pas déjà payé
```

3. Garantie de cohérence ✅

Le statut du CallOffer est toujours cohérent.

Dans la même transaction:
- Le booking est créé
- Le `CallOffer.status` passe à `BOOKED`

Pas de risque d'incohérence entre les deux.

4. Garantie de traçabilité ✅

Toutes les tentatives sont loggées.

```
// Succès
await logBooking('CREATED', bookingId, userId, creatorId, { ... });

// Échec
await logApiError('/api/bookings', error, LogActor.USER, userId, {
  action: 'CREATE_BOOKING',
  reason: 'OFFER_ALREADY_BOOKED'
});
```

🧪 Comment tester

Test de concurrence automatisé

Un script de test est fourni pour simuler des requêtes simultanées:

```bash
# 1. Démarrer le serveur
npm run dev

# 2. Dans un autre terminal, exécuter le test
TEST_CALL_OFFER_ID=<offer-id> \
TEST_AUTH_TOKEN=<your-auth-token> \
npm run test:concurrency
```

Résultat attendu

```
🔬 TEST DE CONCURRENCE - ANTI MULTI-BOOKING
======================================================================

📋 Configuration:
   - Call Offer ID: clx123456
   - Nombre de requêtes simultanées: 5
   - Base URL: http://localhost:3000

🚀 Lancement des requêtes simultanées...

📊 RÉSULTATS DÉTAILLÉS:

   ✅ Request #1: [201] SUCCESS - Booking created successfully (245ms)
   ⚠️  Request #2: [409] CONFLICT (Expected) - This time slot is already booked
(198ms)
   ⚠️  Request #3: [409] CONFLICT (Expected) - This time slot is already booked
(201ms)
   ⚠️  Request #4: [409] CONFLICT (Expected) - This time slot is already booked
(203ms)
   ⚠️  Request #5: [409] CONFLICT (Expected) - This time slot is already booked
(199ms)

----------------------------------------------------------------------
📈 RÉSUMÉ:
----------------------------------------------------------------------
   ✅ Bookings réussis: 1
   ⚠️  Conflits (409): 4
   ❌ Autres erreurs: 0
   ⏱️  Temps total: 250ms
   ⏱️  Temps moyen: 209ms

======================================================================
🎯 VALIDATION:
======================================================================

✅ TEST RÉUSSI !
   → Une seule réservation a été créée (comme attendu)
   → 4 requêtes ont été rejetées avec HTTP 409 (comme attendu)
   → Le système est PROTÉGÉ contre le multi-booking ✨
======================================================================
```

Test manuel

1. **Créer un CallOffer de test**

   `bash`

   `# Via l'interface ou l'API`

   `POST /api/creators/call-offers`

2. **Obtenir un token d'authentification**
   `bash`
   `# Se connecter et récupérer le token depuis les DevTools`
   `# ou via l'API de login`

3. **Ouvrir 2+ onglets navigateur**
   - Onglet 1: Accéder à la page de booking
   - Onglet 2: Accéder à la même page
   - Cliquer simultanément sur "Réserver"

4. **Vérifier les résultats**
   - Un seul onglet doit réussir (201)
   - Les autres doivent recevoir une erreur (409)

📊 Monitoring et logging

Logs de succès

Tous les bookings réussis sont loggés:

```
await logBooking('CREATED', bookingId, userId, creatorId, {
  callOfferId,
  price,
  currency,
  dateTime
});
```

Consultable via:

```
SELECT * FROM "Log"
WHERE type = 'BOOKING_CREATED'
AND status = 'SUCCESS'
ORDER BY "createdAt" DESC;
```

Logs d'erreur

Toutes les tentatives échouées sont loggées avec la raison:

```
await logApiError('/api/bookings', error, LogActor.USER, userId, {
  action: 'CREATE_BOOKING',
  reason: 'OFFER_ALREADY_BOOKED' // ou autre raison
});
```

Consultable via:

```
SELECT * FROM "Log"
WHERE type = 'API_ERROR'
AND status = 'ERROR'
AND context->>'action' = 'CREATE_BOOKING'
ORDER BY "createdAt" DESC;
```

Métriques à surveiller

1. **Taux de conflits** (409)
   - Un taux élevé peut indiquer:
     - Des utilisateurs qui refreshent trop vite
     - Un problème de cache côté client
     - Des offres très populaires (normal)
2. **Temps de réponse des transactions**
   - Devrait rester < 500ms
   - Si > 1s: problème de performance à investiguer
3. **Erreurs Prisma P2002**
   - Devrait être rare (< 0.1% des requêtes)
   - Si fréquent: problème de concurrence extrême

⚠️ Gestion des erreurs

Codes d'erreur HTTP

Code	Raison	Message	Action utilisateur
401	Non authentifié	Non authentifié	Se connecter
404	Offre introuvable	Offre introuvable	Vérifier le lien
400	Offre expirée	Cette offre est expirée	Choisir un autre créneau
400	Offre indisponible	Cette offre n'est plus disponible	Choisir un autre créneau
409	Déjà réservé	This time slot is already booked. Please choose another time.	Choisir un autre créneau
500	Erreur serveur	Une erreur est survenue	Réessayer plus tard

Gestion côté frontend

```
try {
  const response = await fetch('/api/bookings', {
    method: 'POST',
    body: JSON.stringify({ callOfferId })
  });

  if (response.status === 409) {
    // ⚠️ Créneau déjà réservé
    showError('This time slot is already booked. Please choose another time.');
    redirectToOffersList();
    return;
  }

  if (!response.ok) {
    throw new Error('Booking failed');
  }

  const { booking } = await response.json();
  // Continuer vers le paiement

} catch (error) {
  showError('An error occurred. Please try again.');
}
```

🔧 Maintenance et évolutions futures

Points d'attention

1. **Performance des transactions**
   - Surveiller le temps d'exécution des transactions
   - Optimiser les queries si nécessaire
   - Éviter d'ajouter trop de logique dans la transaction

2. **Niveau d'isolation**
   - Le niveau `READ COMMITTED` est suffisant actuellement
   - Si problèmes: considérer `SERIALIZABLE` (plus strict mais plus lent)

3. **Retry logic**
   - Actuellement: pas de retry automatique
   - Si erreur 409: l'utilisateur doit choisir un autre créneau
   - Éviter les retries automatiques (race condition pire)

Évolutions possibles

1. **Système de réservation temporaire**
   - Ajouter un "hold" de 5 minutes sur l'offre
   - L'utilisateur a 5 minutes pour payer
   - Après 5 minutes: libération automatique

2. **File d'attente**
   - Si créneau très populaire
   - Système de queue pour gérer les demandes

3. **Notifications en temps réel**
   - WebSocket pour notifier si créneau devient indisponible
   - Éviter que l'utilisateur clique sur un créneau déjà réservé

📚 Références

- Prisma Transactions (https://www.prisma.io/docs/concepts/components/prisma-client/transactions)
- PostgreSQL Transaction Isolation (https://www.postgresql.org/docs/current/transaction-iso.html)
- HTTP Status Code 409 (Conflict) (https://developer.mozilla.org/en-US/docs/Web/HTTP/Status/409)
- Race Condition Prevention Patterns (https://en.wikipedia.org/wiki/Race_condition#Prevention)

📞 Support

En cas de problème:
1. Vérifier les logs dans la table `Log`
2. Vérifier les métriques de performance
3. Exécuter le script de test de concurrence
4. Contacter l'équipe technique si problème persistant

Document créé le: 2025-12-31
Dernière mise à jour: 2025-12-31
Version: 1.0.0
Auteur: DeepAgent (Abacus.AI)