

Système de Logging Callastar

Vue d'ensemble

Callastar dispose d'un système de logging complet à deux niveaux pour assurer une traçabilité exhaustive de toutes les opérations de la plateforme :

1. `TransactionLog` (`logger.ts`) : Logs financiers (paiements, payouts, remboursements, litiges)
2. `Log` (`system-logger.ts`) : Logs système généraux (actions utilisateurs, admin, système, erreurs)

Architecture du Système

1. Logger Financier (`lib/logger.ts`)

Modèle Prisma : `TransactionLog`

```
model TransactionLog {
    id          String          @id @default(cuid())
    eventType   TransactionEventType // Type d'événement
    entityType  EntityType      // Type d'entité
    entityId    String          // ID de l'entité
    stripeEventId String?       // ID de l'événement Stripe
    amount      Decimal?        // Montant
    currency   String?         // Devise
    status      String?         // Statut
    metadata    Json?          // Métadonnées structurées
    errorMessage String?       // Message d'erreur
    createdAt   DateTime @default(now())

    // Relations optionnelles
    paymentId String?
    payoutId  String?
    refundId  String?

}
```

Enums Disponibles

```
enum TransactionEventType {  
    PAYMENT_CREATED  
    PAYMENT_SUCCEEDED  
    PAYMENT_FAILED  
    REFUND_CREATED  
    REFUND_SUCCEEDED  
    REFUND_FAILED  
    PAYOUT_CREATED  
    PAYOUT_PAID  
    PAYOUT_FAILED  
    TRANSFER_CREATED  
    TRANSFER_SUCCEEDED  
    TRANSFER_FAILED  
    WEBHOOK_RECEIVED  
    DISPUTE_CREATED  
    DISPUTE_UPDATED  
    DISPUTE_CLOSED  
}  
  
enum EntityType {  
    PAYMENT  
    PAYOUT  
    REFUND  
    DISPUTE  
    TRANSFER  
}
```

Fonctions Principales

```

// Log générique de transaction
await logTransaction({
  eventType: TransactionEventType.PAYMENT_CREATED,
  entityType: EntityType.PAYMENT,
  entityId: payment.id,
  amount: 100.50,
  currency: 'EUR',
  status: 'PENDING',
  metadata: { bookingId: 'xyz', userId: 'abc' },
});

// Log de paiement
await logPayment(TransactionEventType.PAYMENT_SUCCEEDED, {
  paymentId: payment.id,
  amount: 100.50,
  currency: 'EUR',
  status: 'SUCCEEDED',
  stripePaymentIntentId: 'pi_xxx',
  metadata: { bookingId: 'xyz' },
});

// Log de payout
await logPayout(TransactionEventType.PAYOUT_CREATED, {
  payoutId: payout.id,
  creatorId: creator.id,
  amount: 85.00,
  currency: 'EUR',
  status: 'REQUESTED',
  stripePayoutId: 'po_xxx',
  metadata: { approvedBy: 'admin_id' },
});

// Log de remboursement
await logRefund(TransactionEventType.REFUND_SUCCEEDED, {
  refundId: refund.id,
  paymentId: payment.id,
  amount: 100.50,
  currency: 'EUR',
  status: 'SUCCEEDED',
  stripeRefundId: 're_xxx',
  reason: 'Demande client',
});

// Log de webhook
await logWebhook({
  stripeEventId: event.id,
  eventType: event.type,
  entityType: EntityType.PAYMENT,
  entityId: payment.id,
  metadata: { paymentIntentId: 'pi_xxx' },
});

// Log d'erreur financière
await logError(
  TransactionEventType.PAYMENT_FAILED,
  EntityType.PAYMENT,
  payment.id,
  new Error('Insufficient funds'),
  { userId: 'user_id', attemptNumber: 2 }
);

```

2. Logger Système (lib/system-logger.ts)

Modèle Prisma : Log

```
model Log {
    id      String    @id @default(cuid())
    level   LogLevel  @default(INFO)
    type    String    // Ex: "USER_LOGIN", "PAYOUT_REQUESTED"
    actor   LogActor // Qui a effectué l'action
    actorId String?  // ID de l'acteur
    message String   @db.Text
    metadata Json?   // Métdonnées additionnelles
    createdAt DateTime @default(now())
}
```

Enums Disponibles

```
enum LogLevel {
    INFO      // Opérations normales
    WARNING   // Anomalies non critiques
    ERROR     // Erreurs qui nécessitent attention
    CRITICAL  // Erreurs critiques bloquantes
}

enum LogActor {
    USER      // Utilisateur standard
    CREATOR   // Créeur
    ADMIN     // Administrateur
    SYSTEM    // Action automatique
    GUEST     // Non authentifié
}
```

Fonctions Principales

```

// Logs génériques par niveau
await logInfo(
  'PAYOUT_REQUEST_INITIATED',
  LogActor.CREATOR,
  'Demande de payout initiée par le créateur',
  creatorId,
  { amount: 500, currency: 'EUR' }
);

await logWarning(
  'PAYOUT_BALANCE_LOW',
  LogActor.SYSTEM,
  'Solde insuffisant pour le payout automatique',
  creatorId,
  { availableBalance: 8.50, requestedAmount: 10 }
);

await logError(
  'STRIPE_API_ERROR',
  LogActor.SYSTEM,
  'Erreur API Stripe lors de la création du payout',
  creatorId,
  {
    errorType: 'api_error',
    errorCode: 'rate_limit',
    errorMessage: 'Too many requests'
  }
);

await logCritical(
  'DATABASE_CONNECTION_LOST',
  LogActor.SYSTEM,
  'Perte de connexion à la base de données',
  undefined,
  { dbHost: 'postgres.example.com', retryAttempt: 3 }
);

// Logs spécialisés

// Authentification
await logAuth(
  'LOGIN',
  userId,
  true, // success
  { ipAddress: '192.168.1.1', userAgent: 'Mozilla/5.0...' }
);

// Actions utilisateur
await logUserAction(
  'BOOKING_CREATED',
  userId,
  'Réservation créée pour un appel',
  { bookingId: 'xyz', creatorId: 'abc', amount: 100 }
);

// Actions créateur
await logCreatorAction(
  'OFFER_CREATED',
  creatorId,
  'Nouvelle offre d\'appel créée',
  { offerId: 'xyz', price: 150, duration: 30 }
);

```

```

// Actions admin
await logAdminAction(
  'PAYOUT_APPROVED',
  adminId,
  'Payout approuvé par l\'administrateur',
  LogLevel.INFO,
  { payoutId: 'xyz', creatorId: 'abc', amount: 500 }
);

// Actions système
await logSystem(
  'CRON_STARTED',
  'Démarrage du cron de traitement des payouts',
  LogLevel.INFO,
  { cronName: 'process-payouts', timestamp: new Date() }
);

// Événements de paiement (haut niveau)
await logPaymentEvent(
  'SUCCEEDED',
  paymentId,
  userId,
  100.50,
  'EUR',
  LogLevel.INFO,
  { bookingId: 'xyz', creatorId: 'abc' }
);

// Événements de payout (haut niveau)
await logPayoutEvent(
  'APPROVED',
  payoutId,
  creatorId,
  500,
  'EUR',
  LogLevel.INFO,
  { adminId: 'admin_id', approvedAt: new Date() }
);

// Webhooks
await logWebhookEvent(
  'STRIPE',
  'payment_intent.succeeded',
  true, // success
  { eventId: 'evt_xxx', paymentIntentId: 'pi_xxx' }
);

// Erreurs API
await logApiError(
  '/api/payouts/request',
  new Error('Validation failed'),
  LogActor.CREATOR,
  creatorId,
  { requestBody: { amount: -50 }, validationErrors: [ 'Amount must be positive' ] }
);

```



Guide d'Utilisation par Catégorie

🎯 PAYOUTS (Traçabilité Exhaustive)

1. Demande de Payout par le Créateur

```

// ✅ DÉBUT : Log d'initiation
await logInfo(
  'PAYOUT_REQUEST_INITIATED',
  LogActor.CREATOR,
  `Demande de payout initiée par ${creator.name}`,
  creatorId,
  {
    creatorId,
    creatorEmail: creator.email,
    requestedAmount: amount,
    currency: 'EUR',
    payoutSchedule: creator.payoutSchedule,
  }
);

// ✅ VALIDATIONS : Log des échecs de validation
if (!creator.stripeAccountId) {
  await logError(
    'PAYOUT_REQUEST_NO_STRIPE_ACCOUNT',
    LogActor.CREATOR,
    'Demande de payout refusée : compte Stripe non configuré',
    creatorId,
    { creatorId, requestedAmount: amount }
  );
  // return error
}

if (creator.isPayoutBlocked) {
  await logError(
    'PAYOUT_REQUEST_BLOCKED',
    LogActor.CREATOR,
    'Demande de payout refusée : payouts bloqués',
    creatorId,
    {
      creatorId,
      blockReason: creator.payoutBlockReason,
      requestedAmount: amount
    }
  );
  // return error
}

// ✅ VÉRIFICATION STRIPE : Log des étapes de vérification
await logInfo(
  'PAYOUT_REQUEST_STRIPE_VERIFICATION',
  LogActor.SYSTEM,
  'Vérification du compte Stripe pour la demande de payout',
  creatorId,
  { creatorId, stripeAccountId: creator.stripeAccountId }
);

// ✅ VÉRIFICATION DU SOLDE
await logInfo(
  'PAYOUT_REQUEST_BALANCE_CHECK',
  LogActor.SYSTEM,
  'Vérification du solde disponible',
  creatorId,
  {
    creatorId,
    requestedAmount: amount,
    availableBalance: availableAmount
  }
)

```

```

);

// ✗ SOLDE INSUFFISANT
if (availableBalance.amount < requestedAmountInCents) {
    await logError(
        'PAYOUT_REQUEST_INSUFFICIENT_BALANCE',
        LogActor.CREATOR,
        'Solde insuffisant pour la demande de payout',
        creatorId,
    {
        creatorId,
        requestedAmount: amount,
        availableAmount: availableBalance.amount / 100,
        currency: 'EUR',
    }
);
// return error
}

// ✅ CRÉATION STRIPE : Log de création du payout Stripe
await logInfo(
    'PAYOUT_REQUEST_STRIPE_CREATION',
    LogActor.SYSTEM,
    'Création du payout Stripe en cours',
    creatorId,
{
    creatorId,
    amount,
    currency: 'EUR',
    stripeAccountId: creator.stripeAccountId,
}
);

// ✅ SUCCÈS STRIPE
await logInfo(
    'PAYOUT_REQUEST_STRIPE_CREATION_SUCCESS',
    LogActor.SYSTEM,
    'Payout Stripe créé avec succès',
    creatorId,
{
    creatorId,
    stripePayoutId: stripePayout.id,
    amount,
    currency: 'EUR',
    status: stripePayout.status,
    arrivalDate: new Date(stripePayout.arrival_date * 1000),
}
);

// ✗ ÉCHEC STRIPE
catch (stripeError) {
    await logError(
        'PAYOUT_REQUEST_STRIPE_CREATION_ERROR',
        LogActor.SYSTEM,
        `Échec de création du payout Stripe: ${stripeError.message}`,
        creatorId,
    {
        creatorId,
        amount,
        stripeErrorType: stripeError.type,
        stripeErrorCode: stripeError.code,
        stripeErrorMessage: stripeError.message,
    }
}

```

```
);

// return error
}

// ✅ FINALISATION : Log de succès complet
await logPayoutEvent(
  'REQUESTED',
  stripePayout.id,
  creatorId,
  amount,
  'EUR',
  LogLevel.INFO,
  {
    creatorId,
    stripePayoutId: stripePayout.id,
    status: stripePayout.status,
    triggeredBy: 'creator',
    processingTimeMs: Date.now() - startTime,
  }
);
```

2. Approbation/Rejet par l'Admin

```

// ✅ APPROBATION : Log d'initiation
await logAdminAction(
  'PAYOUT_APPROVAL_INITIATED',
  adminId,
  'Approbation de payout initiée par l\'administrateur',
  LogLevel.INFO,
  { payoutId, adminId, adminEmail: admin.email }
);

// ✅ VALIDATION : Log de validation réussie
await logAdminAction(
  'PAYOUT_APPROVAL_VALIDATED',
  adminId,
  'Payout validé et prêt pour approbation',
  LogLevel.INFO,
  {
    payoutId,
    adminId,
    creatorId,
    amount,
    currency: 'EUR',
  }
);

// ✅ APPROBATION : Log du changement de statut
await logPayoutEvent(
  'APPROVED',
  payoutId,
  creatorId,
  amount,
  'EUR',
  LogLevel.INFO,
  {
    payoutId,
    adminId,
    previousStatus: 'REQUESTED',
    newStatus: 'APPROVED',
  }
);

// ✅ CRÉATION STRIPE APRÈS APPROBATION
await logInfo(
  'PAYOUT_APPROVAL_STRIPE_CREATION',
  LogActor.SYSTEM,
  'Création du payout Stripe après approbation admin',
  creatorId,
  { payoutId, adminId, amount, stripeAccountId }
);

// ✅ SUCCÈS FINAL
await logAdminAction(
  'PAYOUT_APPROVAL_SUCCESS',
  adminId,
  'Payout approuvé et transfert Stripe déclenché',
  LogLevel.INFO,
  {
    payoutId,
    stripePayoutId,
    amount,
    processingTimeMs: Date.now() - startTime,
  }
);

```

```
// ❌ REJET : Log de rejet
await logPayoutEvent(
  'REJECTED',
  payoutId,
  creatorId,
  amount,
  'EUR',
  LogLevel.WARNING,
  {
    payoutId,
    adminId,
    rejectionReason: reason,
    previousStatus: 'REQUESTED',
    newStatus: 'REJECTED',
  }
);

```

3. Webhooks Stripe (payout.paid, payout.failed)

```
// ✅ PAYOUT PAYÉ (webhook)
await logPayoutEvent(
  'PAID',
  payoutId,
  creatorId,
  amount,
  'EUR',
  LogLevel.INFO,
  {
    payoutId,
    stripePayoutId,
    paidAt: new Date(),
    source: 'stripe_webhook',
  }
);

// ❌ PAYOUT ÉCHOUÉ (webhook)
await logPayoutEvent(
  'FAILED',
  payoutId,
  creatorId,
  amount,
  'EUR',
  LogLevel.ERROR,
  {
    payoutId,
    stripePayoutId,
    failureCode: stripePayout.failure_code,
    failureMessage: stripePayout.failure_message,
    failedAt: new Date(),
    source: 'stripe_webhook',
  }
);

```

CRON JOBS (Tâches Automatiques)

```
// ✅ DÉBUT DU CRON
const startTime = Date.now();
await logSystem(
  'CRON_PAYOUT_STARTED',
  '🤖 Cron de traitement automatique des payouts démarré',
  LogLevel.INFO,
  {
    startTime: new Date().toISOString(),
    endpoint: '/api/cron/process-payouts',
  }
);

// ✅ PROGRESSION (optionnel, pour les gros traitements)
await logInfo(
  'CRON_PAYOUT_PROGRESS',
  LogActor.SYSTEM,
  `Cron de payouts : ${processedCount}/${totalCount} créateurs traités`,
  undefined,
  { processedCount, totalCount, elapsedMs: Date.now() - startTime }
);

// ✅ FIN DU CRON (SUCCÈS)
const duration = Date.now() - startTime;
await logSystem(
  'CRON_PAYOUT_COMPLETED',
  '✅ Cron de traitement automatique des payouts terminé avec succès',
  LogLevel.INFO,
  {
    endTime: new Date().toISOString(),
    durationMs: duration,
    durationSeconds: Math.round(duration / 1000),
    processed: summary.processed,
    succeeded: summary.succeeded,
    failed: summary.failed,
    skipped: summary.skipped,
  }
);

// ❌ ERREUR FATALE DU CRON
await logError(
  'CRON_PAYOUT_FATAL_ERROR',
  LogActor.SYSTEM,
  '❌ Erreur fatale dans le cron de traitement des payouts',
  undefined,
  {
    errorMessage: error.message,
    errorStack: error.stack,
    durationMs: Date.now() - startTime,
    summary,
  }
);
```

WEBHOOKS Stripe

```
// ✓ RÉCEPTION DU WEBHOOK
await logInfo(
  'WEBHOOK_RECEIVED',
  LogActor.SYSTEM,
  `Webhook Stripe reçu : ${event.type}`,
  undefined,
{
  eventId: event.id,
  eventType: event.type,
  livemode: event.livemode,
  apiVersion: event.api_version,
}
);

// ✓ TRAITEMENT RÉUSSI
await logWebhookEvent(
  'STRIPE',
  event.type,
  true, // success
{
  eventId: event.id,
  objectType: event.data.object.object,
  processingTimeMs: Date.now() - startTime,
}
);

// ✗ TRAITEMENT ÉCHOUÉ
await logWebhookEvent(
  'STRIPE',
  event.type,
  false, // failed
{
  eventId: event.id,
  objectType: event.data.object.object,
  errorMessage: error.message,
}
);

// ✗ SIGNATURE INVALIDE
await logError(
  'WEBHOOK_SIGNATURE_INVALID',
  LogActor.SYSTEM,
  'Signature de webhook Stripe invalide',
  undefined,
{
  providedSignature: signature ? '***' : null,
  eventType: event.type,
}
);
```

 PAIEMENTS

```
// ✓ CRÉATION DU PAYMENT INTENT
await logPayment(TransactionEventType.PAYMENT_CREATED, {
  paymentId: payment.id,
  amount,
  currency: 'EUR',
  status: 'PENDING',
  stripePaymentIntentId: paymentIntent.id,
  metadata: {
    bookingId,
    userId,
    creatorId,
    platformFee,
    creatorAmount,
  },
});

// ✓ PAIEMENT RÉUSSI (webhook)
await logPayment(TransactionEventType.PAYMENT_SUCCEEDED, {
  paymentId: payment.id,
  amount,
  currency: 'EUR',
  status: 'SUCCEEDED',
  stripePaymentIntentId: paymentIntent.id,
  metadata: {
    bookingId,
    creatorId,
  },
});

// ✗ PAIEMENT ÉCHOUÉ
await logPayment(TransactionEventType.PAYMENT_FAILED, {
  paymentId: payment.id,
  amount,
  currency: 'EUR',
  status: 'FAILED',
  stripePaymentIntentId: paymentIntent.id,
  errorMessage: paymentIntent.last_payment_error?.message,
});
```

ACTIONS SENSIBLES

```
// ✅ BLOCAGE DE PAYOUTS
await logAdminAction(
  'CREATOR_PAYOUT_BLOCKED',
  adminId,
  `Payouts bloqués pour le créateur ${creatorName}`,
  LogLevel.WARNING,
  {
    creatorId,
    creatorEmail,
    blockReason: reason,
    blockedBy: adminId,
    blockedAt: new Date(),
  }
);

// ✅ DÉBLOCAGE DE PAYOUTS
await logAdminAction(
  'CREATOR_PAYOUT_UNBLOCKED',
  adminId,
  `Payouts débloqués pour le créateur ${creatorName}`,
  LogLevel.INFO,
  {
    creatorId,
    unblockedBy: adminId,
    unblockedAt: new Date(),
  }
);

// ✅ REMBOURSEMENT CRÉÉ
await logRefund(TransactionEventType.REFUND_CREATED, {
  refundId: refund.id,
  paymentId: payment.id,
  amount: refundAmount,
  currency: 'EUR',
  status: 'PENDING',
  reason: refundReason,
  metadata: {
    initiatedBy: adminId,
    bookingId,
  },
});

// ✅ VALIDATION DE COMPTE CRÉATEUR
await logAdminAction(
  'CREATOR_ACCOUNT_VALIDATED',
  adminId,
  `Compte créateur validé pour ${creatorName}`,
  LogLevel.INFO,
  {
    creatorId,
    validatedBy: adminId,
    validatedAt: new Date(),
  }
);
```



Bonnes Pratiques

1. Niveaux de Gravité

- **INFO** : Opérations normales et attendues (succès, démarrages, fins)
- **WARNING** : Anomalies non critiques (solde bas, tentatives multiples)
- **ERROR** : Erreurs nécessitant attention (échecs API, validations)
- **CRITICAL** : Erreurs critiques bloquantes (DB down, config manquante)

2. Métadonnées Riches

 **BON** : Métadonnées complètes

```
await logError(
  'PAYOUT_REQUEST_FAILED',
  LogActor.CREATOR,
  'Échec de la demande de payout',
  creatorId,
{
  creatorId,
  creatorEmail: creator.email,
  requestedAmount: 500,
  availableBalance: 450,
  currency: 'EUR',
  stripeAccountId: creator.stripeAccountId,
  errorType: 'insufficient_funds',
  errorCode: 'balance_insufficient',
  errorMessage: 'Available balance is less than requested amount',
}
);
```

 **MAUVAIS** : Métadonnées pauvres

```
await logError(
  'PAYOUT_REQUEST_FAILED',
  LogActor.CREATOR,
  'Échec',
  creatorId
);
```

3. Messages Descriptifs

 **BON** : Message clair et contextualisé

```
await logInfo(
  'PAYOUT_APPROVED',
  LogActor.ADMIN,
  `Payout de ${amount} EUR approuvé par l'admin ${adminName} pour le créateur ${creatorName}`,
  adminId,
  metadata
);
```

 **MAUVAIS** : Message vague

```

await logInfo(
  'PAYOUT_APPROVED',
  LogActor.ADMIN,
  'Payout approved',
  adminId
);

```

4. Gestion des Erreurs

BON : Try-catch avec logs détaillés

```

try {
  const stripePayout = await stripe.payouts.create({...});

  await logInfo(
    'PAYOUT_STRIPE_CREATION_SUCCESS',
    LogActor.SYSTEM,
    'Payout Stripe créé avec succès',
    creatorId,
    {
      stripePayoutId: stripePayout.id,
      amount,
      status: stripePayout.status,
    }
  );
} catch (error) {
  await logError(
    'PAYOUT_STRIPE_CREATION_ERROR',
    LogActor.SYSTEM,
    `Erreur Stripe: ${error.message}`,
    creatorId,
    {
      errorType: error.type,
      errorCode: error.code,
      errorMessage: error.message,
      errorStack: error.stack,
    }
  );
}

throw error;
}

```

5. Timing et Performance

```

const startTime = Date.now();

try {
    // ... traitement ...

    await logInfo(
        'OPERATION_SUCCESS',
        LogActor.SYSTEM,
        'Opération terminée avec succès',
        entityId,
        {
            processingTimeMs: Date.now() - startTime,
            itemsProcessed: items.length,
        }
    );
} catch (error) {
    await logError(
        'OPERATION_FAILED',
        LogActor.SYSTEM,
        `Opération échouée après ${Date.now() - startTime}ms`,
        entityId,
        {
            errorMessage: error.message,
            processingTimeMs: Date.now() - startTime,
        }
    );
}
}

```



Exemples de Logs par Cas d'Usage

Cas 1 : Traçabilité d'un Payout Complet

- | | | | |
|----------|--|--|---|
| 1. INFO | PAYOUT_REQUEST_INITIATED | | Demande initiée par créateur X |
| 2. INFO | PAYOUT_REQUEST_STRIPE_VERIFICATION | | Vérification compte Stripe |
| 3. INFO | PAYOUT_REQUEST_BALANCE_CHECK | | Vérification du solde |
| 4. INFO | PAYOUT_REQUEST_STRIPE_CREATION | | Création du payout Stripe |
| 5. INFO | PAYOUT_REQUEST_STRIPE_CREATION_SUCCESS | | Payout Stripe créé avec succès |
| 6. INFO | PAYOUT_REQUESTED | | Payout demandé (log système + financier) |
| 7. INFO | PAYOUT_APPROVAL_INITIATED | | Admin démarre l'approbation |
| 8. INFO | PAYOUT_APPROVAL_VALIDATED | | Payout validé par admin |
| 9. INFO | PAYOUT_APPROVED | | Statut changé à APPROVED |
| 10. INFO | PAYOUT_APPROVAL_STRIPE_CREATION | | Création du payout Stripe après approbation |
| 11. INFO | PAYOUT_APPROVAL_SUCCESS | | Approbation terminée avec succès |
| 12. INFO | WEBHOOK_RECEIVED | | Webhook payout.paid reçu |
| 13. INFO | PAYOUT_PAID | | Payout payé avec succès |

Cas 2 : Échec d'un Payout

- | | | | | |
|----------------------------------|--|-------------------------------------|--|-----------------------|
| 1. INFO | | PAYOUT_REQUEST_INITIATED | | Demande initiée |
| 2. INFO | | PAYOUT_REQUEST_BALANCE_CHECK | | Vérification du solde |
| 3. ERROR | | PAYOUT_REQUEST_INSUFFICIENT_BALANCE | | Solde insuffisant |
| 4. (retour d'erreur au créateur) | | | | |

Cas 3 : Cron de Traitement Automatique

- | | | | | |
|---------|--|-----------------------|--|---|
| 1. INFO | | CRON_PAYOUT_STARTED | | Cron démarré |
| 2. INFO | | CRON_PAYOUT_PROGRESS | | 5/20 créateurs traités |
| 3. INFO | | CRON_PAYOUT_PROGRESS | | 10/20 créateurs traités |
| 4. INFO | | CRON_PAYOUT_PROGRESS | | 15/20 créateurs traités |
| 5. INFO | | CRON_PAYOUT_PROGRESS | | 20/20 créateurs traités |
| 6. INFO | | CRON_PAYOUT_COMPLETED | | Cron terminé : 15 succès, 3 échecs, 2 ignorés |

Consultation des Logs

Via l'Interface Admin

1. Accéder à `/dashboard/admin/system-logs` pour les logs système
2. Filtrer par :
 - **Niveau** : INFO, WARNING, ERROR, CRITICAL
 - **Type** : PAYOUT_, PAYMENT_, CRON_, etc.
 - Acteur : USER, CREATOR, ADMIN, SYSTEM
 - Date : Plage de dates
 - Recherche* : Texte libre

Via Prisma Studio

```
npx prisma studio
```

Puis accéder aux tables `Log` et `TransactionLog`.

Via SQL Direct

```
-- Tous les logs d'un payout spécifique
SELECT * FROM "Log"
WHERE type LIKE 'PAYOUT_%'
    AND metadata::text LIKE '%payoutId%'
ORDER BY "createdAt" DESC;

-- Logs d'erreur des dernières 24h
SELECT * FROM "Log"
WHERE level IN ('ERROR', 'CRITICAL')
    AND "createdAt" >= NOW() - INTERVAL '24 hours'
ORDER BY "createdAt" DESC;

-- Résumé des logs par type
SELECT type, level, COUNT(*) as count
FROM "Log"
WHERE "createdAt" >= NOW() - INTERVAL '7 days'
GROUP BY type, level
ORDER BY count DESC;
```

Politique de Rétention

Les logs sont automatiquement nettoyés selon la politique de rétention :

- **INFO** : 30 jours
- **WARNING** : 60 jours
- **ERROR** : 90 jours
- **CRITICAL** : Conservés indéfiniment

Le cron de nettoyage s'exécute quotidiennement via `/api/cron/cleanup-logs`.



Checklist pour Ajouter de Nouveaux Logs

Lors de l'implémentation d'une nouvelle fonctionnalité, assurez-vous de :

- Début de l'opération** : Log INFO de démarrage
- Validations** : Log ERROR pour chaque validation échouée
- Étapes intermédiaires** : Log INFO pour les étapes importantes
- Appels externes** : Log INFO avant/après les appels API (Stripe, etc.)
- Succès** : Log INFO de fin avec durée et résumé
- Erreurs** : Log ERROR avec détails (type, code, message, stack)
- Métadonnées riches** : IDs, montants, statuts, raisons, durées
- Messages clairs** : Contexte suffisant pour comprendre l'événement
- Niveau approprié** : INFO, WARNING, ERROR, CRITICAL selon la gravité



Fichiers Modifiés avec Logs Exhaustifs

Les fichiers suivants ont été mis à jour avec des logs exhaustifs :

Payouts (PRIORITÉ 1)

- `/app/api/creators/payouts/request/route.ts` - Demande de payout par créateur
- `/app/api/admin/payouts/[id]/approve/route.ts` - Approbation par admin
- `/app/api/admin/payouts/[id]/reject/route.ts` - Rejet par admin

Cron Jobs

- `/app/api/cron/process-payouts/route.ts` - Traitement automatique des payouts
- `/app/api/cron/cleanup-logs/route.ts` - Nettoyage des logs

Bookings (PHASE 1 - COMPLÉTÉE)

- `/app/api/bookings/route.ts` - Création de booking avec logs détaillés
- `/app/api/payments/webhook/route.ts` - Confirmation de booking via webhook (ligne 551-566)
- `/app/api/call-offers/[id]/route.ts` - Modification, annulation et complétion de bookings/call offers

Webhooks & Paiements (PHASES 2 & 3 - COMPLÉTÉES)

- `/app/api/payments/webhook/route.ts` - Webhooks Stripe avec logs exhaustifs :
- Validation de signature avec timing
- Traitement des événements avec durée
- Logs de début et fin de traitement
- Logs d'erreurs détaillés
- `/app/api/payments/create-intent/route.ts` - Création de payment intent avec logs exhaustifs :
- Logs d'initiation
- Logs de validation (booking, ownership, déjà payé)
- Logs de succès avec timing
- Logs d'erreurs détaillés

Actions Sensibles Admin (PHASE 4 - COMPLÉTÉE)

- `/app/api/admin/payouts/block/route.ts` - Blocage de payouts avec logs complets :
 - Logs d'initiation par admin
 - Logs de validation (créateur, raison)
 - Logs d'avertissement (déjà bloqué)
 - Logs de succès avec détails complets
 - Logs d'erreurs avec timing
 - `/app/api/admin/payouts/unblock/route.ts` - Déblocage de payouts avec logs complets :
 - Logs d'initiation par admin
 - Logs de validation (créateur)
 - Logs d'avertissement (non bloqué)
 - Logs de succès avec détails complets
 - Logs d'erreurs avec timing
-

 **Nouveaux Événements Loggés (Phase 2)** **BOOKINGS (Rendez-vous)**

```

// ✅ CRÉATION DE BOOKING
await logBooking(
  'CREATED',
  booking.id,
  user.userId,
  callOffer.creatorId,
  {
    callOfferId: validatedData.callOfferId,
    price: callOffer.price.toString(),
    currency: callOffer.currency,
    dateTIme: callOffer.dateTIme.toISOString(),
  }
);

// ✅ CONFIRMATION DE BOOKING (via webhook)
await logBooking(
  'CONFIRMED',
  booking.id,
  booking.userId,
  booking.callOffer.creatorId,
  {
    previousStatus: 'PENDING',
    newStatus: 'CONFIRMED',
    dailyRoomUrl: room.url,
    dailyRoomName: room.name,
    confirmedVia: 'payment_intent_succeeded_webhook',
    paymentIntentId: paymentIntent.id,
    amount: amount,
    currency: currency,
  }
);

// ✅ ANNULATION DE BOOKING (via call offer update)
await logBooking(
  'CANCELLED',
  updated.booking.id,
  updated.booking.userId,
  updated.creator.id,
  {
    callOfferId: id,
    previousStatus: callOffer.status,
    newStatus: 'CANCELLED',
    cancelledBy: 'creator',
    cancelledByUserId: user.userId,
    cancelledAt: new Date().toISOString(),
    reason: 'Cancelled by creator via call offer update',
  }
);

// ✅ COMPLÉTION DE BOOKING (via call offer update)
await logBooking(
  'COMPLETED',
  updated.booking.id,
  updated.booking.userId,
  updated.creator.id,
  {
    callOfferId: id,
    previousStatus: callOffer.status,
    newStatus: 'COMPLETED',
    completedAt: new Date().toISOString(),
    completedBy: 'creator',
  }
);

```

```
);

// ✗ ERREURS LORS DE LA CRÉATION
await logApiError(
  '/api/bookings',
  error instanceof Error ? error : 'Unknown error',
  LogActor.USER,
  user?.userId,
  { action: 'CREATE_BOOKING' }
);
```

 **WEBHOOKS STRIPE (Améliorations)**

```

// ✅ VALIDATION DE SIGNATURE RÉUSSIE
await logInfo(
  'WEBHOOK_SIGNATURE_VERIFIED',
  LogActor.SYSTEM,
  'Signature webhook Stripe vérifiée avec succès',
  undefined,
{
  verificationTimeMs: 15,
  signaturePresent: true,
}
);

// ❌ VALIDATION DE SIGNATURE ÉCHOUÉE
await logError(
  'WEBHOOK_SIGNATURE_VERIFICATION_FAILED',
  LogActor.SYSTEM,
  'Échec de la vérification de la signature du webhook Stripe',
  undefined,
{
  verificationTimeMs: 12,
  errorMessage: 'Invalid signature',
  signatureProvided: true,
}
);

// ⚠ WEBHOOK SANS SIGNATURE
await logWarning(
  'WEBHOOK_NO_SIGNATURE',
  LogActor.SYSTEM,
  'Webhook Stripe reçu sans signature',
  undefined,
  { headersPresent: true }
);

// ✅ DÉBUT DU TRAITEMENT
await logInfo(
  'WEBHOOK_PROCESSING_STARTED',
  LogActor.SYSTEM,
  `Début du traitement du webhook Stripe : payment_intent.succeeded`,
  undefined,
{
  eventId: 'evt_xxx',
  eventType: 'payment_intent.succeeded',
  objectType: 'payment_intent',
  livemode: true,
}
);

// ✅ TRAITEMENT RÉUSSI
await logInfo(
  'WEBHOOK_PROCESSING_SUCCESS',
  LogActor.SYSTEM,
  `Webhook Stripe traité avec succès : payment_intent.succeeded`,
  undefined,
{
  eventId: 'evt_xxx',
  eventType: 'payment_intent.succeeded',
  processingTimeMs: 450,
  totalTimeMs: 520,
}
);

```

 **PAIEMENTS (Améliorations)**

```

// ✅ INITIATION DE CRÉATION PAYMENT INTENT
await logInfo(
  'PAYMENT_INTENT_CREATION_INITIATED',
  LogActor.USER,
  'Création de payment intent initiée',
  user.userId,
  { bookingId: validatedData.bookingId }
);

// ❌ NON AUTHENTIFIÉ
await logError(
  'PAYMENT_INTENT_UNAUTHORIZED',
  LogActor.GUEST,
  'Tentative de création de payment intent sans authentification',
  undefined,
  { endpoint: '/api/payments/create-intent' }
);

// ❌ RÉSERVATION INTROUVABLE
await logError(
  'PAYMENT_INTENT_BOOKING_NOT_FOUND',
  LogActor.USER,
  'Tentative de paiement pour une réservation introuvable',
  user.userId,
  { bookingId: validatedData.bookingId }
);

// ❌ ACCÈS REFUSÉ
await logError(
  'PAYMENT_INTENT_ACCESS_DENIED',
  LogActor.USER,
  'Tentative de paiement pour une réservation non possédée',
  user.userId,
  {
    bookingId: validatedData.bookingId,
    bookingOwnerId: booking.userId,
  }
);

// ❌ DÉJÀ PAYÉ
await logError(
  'PAYMENT_INTENT_ALREADY_PAID',
  LogActor.USER,
  'Tentative de paiement pour une réservation déjà payée',
  user.userId,
  {
    bookingId: validatedData.bookingId,
    bookingStatus: 'CONFIRMED',
  }
);

// ✅ SUCCÈS AVEC TIMING
await logPaymentEvent(
  'INITIATED',
  payment.id,
  user.userId,
  amount,
  creatorCurrency,
  LogLevel.INFO,
  {
    paymentId: payment.id,
    bookingId: booking.id,
  }
);

```

```
creatorId: booking.callOffer.creatorId,  
paymentIntentId: paymentIntent.id,  
amount,  
currency: creatorCurrency,  
platformFee,  
creatorAmount,  
processingTimeMs: 234,  
useStripeConnect: true,  
}  
);
```

 **ACTIONS SENSIBLES ADMIN**

```

// ✅ BLOCAGE DE PAYOUTS - INITIATION
await logAdminAction(
  'PAYOUT_BLOCK_INITIATED',
  adminId,
  'Blocage de payout initié par administrateur',
  LogLevel.INFO,
  {
    adminId,
    adminEmail: 'admin@example.com',
    creatorId,
    reason: 'Activité suspecte détectée',
  }
);

// ❌ BLOCAGE - VALIDATION ÉCHOUÉE
await logError(
  'PAYOUT_BLOCK_VALIDATION_ERROR',
  LogActor.ADMIN,
  'ID du créateur manquant pour blocage de payout',
  adminId,
  { error: 'missing_creator_id' }
);

// ⚠️ BLOCAGE - DÉJÀ BLOQUÉ
await logWarning(
  'PAYOUT_BLOCK_ALREADY_BLOCKED',
  LogActor.ADMIN,
  'Tentative de blocage pour un créateur déjà bloqué',
  adminId,
  {
    creatorId,
    creatorName: 'John Doe',
    creatorEmail: 'john@example.com',
    currentReason: 'Compte en révision',
  }
);

// ✅ BLOCAGE - SUCCÈS
await logAdminAction(
  'CREATOR_PAYOUT_BLOCKED',
  adminId,
  `Payouts bloqués avec succès pour le créateur John Doe`,
  LogLevel.WARNING,
  {
    creatorId,
    creatorName: 'John Doe',
    creatorEmail: 'john@example.com',
    blockReason: 'Activité suspecte détectée',
    adminId,
    adminEmail: 'admin@example.com',
    processingTimeMs: 145,
    auditLogId: 'audit_xxx',
  }
);

// ✅ DÉBLOCAGE DE PAYOUTS - INITIATION
await logAdminAction(
  'PAYOUT_UNBLOCK_INITIATED',
  adminId,
  'Déblocage de payout initié par administrateur',
  LogLevel.INFO,
  {

```

```
    adminId,
    adminEmail: 'admin@example.com',
    creatorId,
}
);

// ! DÉBLOCAGE - NON BLOQUÉ
await logWarning(
  'PAYOUT_UNBLOCK_NOT_BLOCKED',
  LogActor.ADMIN,
  'Tentative de déblocage pour un créateur non bloqué',
  adminId,
{
  creatorId,
  creatorName: 'Jane Doe',
  creatorEmail: 'jane@example.com',
}
);

// ✓ DÉBLOCAGE - SUCCÈS
await logAdminAction(
  'CREATOR_PAYOUT_UNBLOCKED',
  adminId,
  `Payouts débloqués avec succès pour le créateur Jane Doe`,
  LogLevel.INFO,
{
  creatorId,
  creatorName: 'Jane Doe',
  creatorEmail: 'jane@example.com',
  previousBlockReason: 'Vérification terminée',
  adminId,
  adminEmail: 'admin@example.com',
  processingTimeMs: 98,
  auditLogId: 'audit_yyy',
}
);
}
```

CALL OFFERS (Modifications)

```
// ✅ MODIFICATION D'OFFRE - INITIATION
await logCreatorAction(
  'OFFER_UPDATE_INITIATED',
  creatorId,
  `Modification d'offre initiée pour Appel 30min`,
  {
    offerId: 'offer_xxx',
    changes: { price: 150, duration: 60 },
    previousStatus: 'AVAILABLE',
  }
);

// ✅ MODIFICATION D'OFFRE - SUCCÈS
await logCreatorAction(
  'OFFER_UPDATED',
  creatorId,
  `Offre d'appel modifiée avec succès : Appel 30min`,
  {
    offerId: 'offer_xxx',
    changes: ['price: 100 → 150', 'duration: 30 → 60'],
    newStatus: 'AVAILABLE',
  }
);

// ✅ SUPPRESSION D'OFFRE - INITIATION
await logCreatorAction(
  'OFFER_DELETE_INITIATED',
  creatorId,
  `Suppression d'offre initiée pour Appel 30min`,
  {
    offerId: 'offer_xxx',
    offerTitle: 'Appel 30min',
    offerStatus: 'AVAILABLE',
  }
);

// ✅ SUPPRESSION D'OFFRE - SUCCÈS
await logCreatorAction(
  'OFFER_DELETED',
  creatorId,
  `Offre d'appel supprimée avec succès : Appel 30min`,
  {
    offerId: 'offer_xxx',
    offerTitle: 'Appel 30min',
    deletedAt: new Date().toISOString(),
  }
);
```

Nouveaux Critères de Traçabilité

Avec les logs de la Phase 2, un administrateur peut maintenant :

1. **✅ Tracer tout le cycle de vie d'un booking :**
 - Création par l'utilisateur
 - Confirmation automatique via webhook Stripe

- Annulation par le créateur
 - Complétion après l'appel
2. **Surveiller les webhooks Stripe en temps réel :**
- Temps de validation de signature
 - Temps de traitement des événements
 - Échecs de signature
 - Événements traités avec succès/échec
3. **Auditer les paiements :**
- Toutes les tentatives de paiement
 - Validations échouées (booking introuvable, accès refusé, déjà payé)
 - Temps de traitement de chaque paiement
 - Détails complets (montant, devise, fees, créateur)
4. **Suivre les actions sensibles des admins :**
- Qui a bloqué/débloqué les payouts d'un créateur
 - Quand et pourquoi
 - Tentatives de blocage sur créateurs déjà bloqués
 - Temps de traitement de chaque action
5. **Monitoredes modifications de call offers :**
- Modifications de prix, durée, description
 - Suppressions d'offres
 - Historique complet des changements
-

Références

- **Fichiers de logs :**
 - lib/logger.ts - Logger financier
 - lib/system-logger.ts - Logger système
 - **Modèles Prisma :**
 - TransactionLog - Logs financiers
 - Log - Logs système
 - **Pages admin :**
 - /dashboard/admin/system-logs - Consultation des logs système
 - /dashboard/admin/logs - Consultation des logs financiers
-

Critères de Succès

Un administrateur doit pouvoir :

1. Retracer **tout le cycle de vie d'un payout** en consultant uniquement les logs
2. Comprendre **pourquoi un payout a échoué** avec détails (raison, code d'erreur Stripe, montant, solde)

3. Voir **qui a effectué quelle action** (admin, créateur, système) avec timestamps
 4. Suivre l'**exécution des cron jobs** (début, fin, durée, nombre d'items traités)
 5. Déetecter les **anomalies** via les logs WARNING/ERROR
 6. Auditer les **actions sensibles** (blocage de payouts, remboursements, validations)
 7. Débugger les **erreurs** avec stack traces et métadonnées complètes
-

Dernière mise à jour : 28 décembre 2025 - Phase 2 complétée (Bookings, Webhooks, Paiements, Actions Admin)