# 📊 Système de Logs Backend - Callastar

## Table des matières



---

## Vue d’ensemble

Le système de logs de Callastar est conçu pour tracer toutes les activités de la plateforme de manière complète et performante. Il est distinct du système `TransactionLog` (qui gère uniquement les transactions financières) et couvre :

- ✅ Actions des utilisateurs (fans)
- ✅ Actions des créateurs
- ✅ Actions des administrateurs
- ✅ Événements système
- ✅ Webhooks externes (Stripe, Daily.co, etc.)
- ✅ Erreurs backend

### Architecture à deux niveaux

1. **TransactionLog** : Logs financiers uniquement (paiements, payouts, refunds, disputes, transfers)
2. **Log (SystemLog)** : Logs généraux de toutes les activités de la plateforme

---

# Architecture

## Modèle Prisma

```
model Log {
  id        String    @id @default(cuid())
  level     LogLevel  @default(INFO)
  type      String
  actor     LogActor
  actorId   String?
  message   String    @db.Text
  metadata  Json?
  createdAt DateTime  @default(now())

  @@index([createdAt])
  @@index([level])
  @@index([type])
  @@index([actor])
  @@index([actorId])
  @@index([level, createdAt])
}

enum LogLevel {
  INFO
  WARNING
  ERROR
  CRITICAL
}

enum LogActor {
  USER
  CREATOR
  ADMIN
  SYSTEM
  GUEST
}
```

## Service de logging : `lib/system-logger.ts`

Le service centralise toutes les opérations de logging avec des fonctions spécialisées.

---

## Niveaux de gravité

| Niveau | Description | Exemples |
|---|---|---|
| **INFO** | Actions normales | Connexion utilisateur, création de booking, webhooks OK |
| **WARNING** | Comportement anormal mais non bloquant | Tentative de connexion échouée, accès refusé |
| **ERROR** | Erreur fonctionnelle ou technique | Erreur API, erreur de validation, échec de paiement |
| **CRITICAL** | Erreur bloquante nécessitant une attention immédiate | Incohérence Stripe, erreur de webhook critique, corruption de données |

---

## Types d’acteurs

| Acteur | Description |
|---|---|
| **USER** | Utilisateur (fan) de la plateforme |
| **CREATOR** | Créateur de contenu |
| **ADMIN** | Administrateur |
| **SYSTEM** | Événement système automatique |
| **GUEST** | Visiteur non authentifié |

---

## Politique de rétention

La rétention des logs est automatique et basée sur le niveau de gravité :

| Niveau | Rétention | Raison |
|---|---|---|
| **INFO** | 30 jours | Logs d’activité normale, grand volume |
| **WARNING** | 60 jours | Comportements suspects à surveiller |
| **ERROR** | 90 jours | Erreurs importantes pour debug et analyse |
| **CRITICAL** | **Illimité** | Conservation permanente pour audit et sécurité |

Le nettoyage automatique est effectué quotidiennement à 3h du matin via un cron job Vercel.

---

## Utilisation du système de logs

### Import

```
import {
  logInfo,
  logWarning,
  logError,
  logCritical,
  logAuth,
  logUserAction,
  logCreatorAction,
  logAdminAction,
  logSystem,
  logBooking,
  logPaymentEvent,
  logPayoutEvent,
  logWebhookEvent,
  logApiError,
} from '@/lib/system-logger';
import { LogActor, LogLevel } from '@prisma/client';
```

## Fonctions principales

### 1. Logs génériques par niveau

```
// Log INFO
await logInfo(
  'USER_PROFILE_UPDATE',
  LogActor.USER,
  'User updated their profile',
  userId,
  { updatedFields: ['name', 'bio'] }
);

// Log WARNING
await logWarning(
  'RATE_LIMIT_APPROACHING',
  LogActor.USER,
  'User approaching rate limit',
  userId,
  { requestCount: 95, limit: 100 }
);

// Log ERROR
await logError(
  'API_ERROR',
  LogActor.SYSTEM,
  'Failed to fetch user data',
  undefined,
  { endpoint: '/api/users', errorCode: 500 }
);

// Log CRITICAL
await logCritical(
  'DATA_CORRUPTION',
  LogActor.SYSTEM,
  'Critical data integrity issue detected',
  undefined,
  { affectedTable: 'payments', count: 10 }
);
```

### 2. Logs d’authentification

```
// Connexion réussie
await logAuth('LOGIN', userId, true, {
  ipAddress: request.headers.get('x-forwarded-for'),
  userAgent: request.headers.get('user-agent'),
});

// Connexion échouée
await logAuth('LOGIN', 'unknown', false, {
  email: 'user@example.com',
  reason: 'Invalid password',
  ipAddress: request.headers.get('x-forwarded-for'),
});

// Autres actions d'authentification
await logAuth('LOGOUT', userId, true);
await logAuth('REGISTER', userId, true);
await logAuth('PASSWORD_RESET', userId, true);
await logAuth('EMAIL_VERIFY', userId, true);
```

### 3. Logs d’actions utilisateur/créateur/admin

```
// Action utilisateur
await logUserAction(
  'BOOKING_CREATED',
  userId,
  'User created a booking',
  { bookingId, creatorId, price: 50 }
);

// Action créateur
await logCreatorAction(
  'CALL_OFFER_CREATED',
  creatorId,
  'Creator created a new call offer',
  { offerId, price: 50, duration: 30 }
);

// Action admin
await logAdminAction(
  'PAYOUT_APPROVED',
  adminId,
  'Admin approved payout',
  LogLevel.INFO,
  { payoutId, creatorId, amount: 500 }
);
```

### 4. Logs de bookings

```
await logBooking(
  'CREATED',  // 'CREATED' | 'CONFIRMED' | 'CANCELLED' | 'COMPLETED'
  bookingId,
  userId,
  creatorId,
  { price: 50, currency: 'EUR', dateTime: '2024-01-15T10:00:00Z' }
);
```

### 5. Logs de paiements et payouts

```
// Paiement
await logPaymentEvent(
  'SUCCEEDED',  // 'INITIATED' | 'SUCCEEDED' | 'FAILED' | 'REFUNDED'
  paymentId,
  userId,
  50,
  'EUR',
  undefined,  // level (optionnel, auto-détecté)
  { stripePaymentIntentId: 'pi_xxxxx' }
);

// Payout
await logPayoutEvent(
  'APPROVED',  // 'REQUESTED' | 'APPROVED' | 'REJECTED' | 'PAID' | 'FAILED'
  payoutId,
  creatorId,
  500,
  'EUR',
  undefined,
  { approvedBy: adminId }
);
```

### 6. Logs de webhooks

```
// Webhook Stripe réussi
await logWebhookEvent(
  'STRIPE',
  'payment_intent.succeeded',
  true,
  { eventId: 'evt_xxxxx', paymentIntentId: 'pi_xxxxx' }
);

// Webhook échoué
await logWebhookEvent(
  'STRIPE',
  'payout.failed',
  false,
  { eventId: 'evt_xxxxx', error: 'Insufficient funds' }
);
```

### 7. Logs d’erreurs API

```
await logApiError(
  '/api/bookings',
  new Error('Database connection failed'),
  LogActor.USER,
  userId,
  { action: 'CREATE_BOOKING', bookingData: {...} }
);
```

### 8. Logs système

```
await logSystem(
  'DATABASE_BACKUP',
  'Daily database backup completed',
  LogLevel.INFO,
  { backupSize: '2.5GB', duration: 1200 }
);
```

---

# Page d’administration

## Accès

La page d’administration des logs système est accessible à l’adresse :

```
/dashboard/admin/system-logs
```

## Fonctionnalités

### Visualisation

- 📊 **Tableau des logs** : 100 logs par page par défaut (configurable)
- 🔍 **Filtres avancés** :
- Niveau (INFO, WARNING, ERROR, CRITICAL)
- Acteur (USER, CREATOR, ADMIN, SYSTEM, GUEST)
- Type (recherche partielle)
- Recherche globale (message, type, actorId)
- Plage de dates (startDate, endDate)
- 📈 **Statistiques** : Nombre total de logs, page actuelle, filtres actifs
- 🔄 **Auto-actualisation** : Optionnelle, toutes les 30 secondes
- 👁️ **Détails** : Vue détaillée de chaque log avec métadonnées complètes

### Suppression

- 🗑️ **Suppression par date** : Sélectionner une plage de dates pour supprimer les logs
- ⚠️ **Avertissement** : Action irréversible, confirmation requise
- 🎯 **Suppression ciblée** : Appliquer les filtres actifs à la suppression

## Logs financiers séparés

Les logs financiers (TransactionLog) restent sur la page :

```
/dashboard/admin/logs
```

---

# Système de rétention automatique

## Cron Job Vercel

Le nettoyage automatique est configuré dans `vercel.json` :

```
{
  "crons": [
    {
      "path": "/api/cron/cleanup-logs",
      "schedule": "0 3 * * *"
    }
  ]
}
```

**Fréquence** : Quotidienne à 3h du matin (UTC)

## Script manuel

Vous pouvez également exécuter le nettoyage manuellement :

```
npm run cleanup-logs
```

Ou directement :

```
npx tsx scripts/cleanup-logs.ts
```

## Sécurité du cron

Le cron job est protégé par un secret dans les variables d'environnement :

```
CRON_SECRET=your-secret-token-here
```

Le header `Authorization: Bearer <CRON_SECRET>` doit être présent pour autoriser l'exécution.

---

## Exemples d’intégration

### Exemple 1 : Login

```ts
// app/api/auth/login/route.ts
import { logAuth } from '@/lib/system-logger';

export async function POST(request: NextRequest) {
  try {
    const user = await authenticateUser(email, password);

    // Log connexion réussie
    await logAuth('LOGIN', user.id, true, {
      email: user.email,
      role: user.role,
      ipAddress: request.headers.get('x-forwarded-for'),
      userAgent: request.headers.get('user-agent'),
    });

    return NextResponse.json({ success: true, user });
  } catch (error) {
    // Log connexion échouée
    await logAuth('LOGIN', 'unknown', false, {
      email,
      reason: error.message,
      ipAddress: request.headers.get('x-forwarded-for'),
    });

    return NextResponse.json({ error: 'Authentication failed' }, { status: 401 });
  }
}
```

### Exemple 2 : Création de booking

```
// app/api/bookings/route.ts
import { logBooking, logApiError } from '@/lib/system-logger';
import { LogActor } from '@prisma/client';

export async function POST(request: NextRequest) {
  try {
    const booking = await createBooking(userId, callOfferId);

    // Log création de booking
    await logBooking(
      'CREATED',
      booking.id,
      userId,
      booking.callOffer.creatorId,
      {
        callOfferId,
        price: booking.totalPrice,
        currency: booking.callOffer.currency,
        dateTime: booking.callOffer.dateTime,
      }
    );

    return NextResponse.json({ booking }, { status: 201 });
  } catch (error) {
    // Log erreur
    await logApiError(
      '/api/bookings',
      error,
      LogActor.USER,
      userId,
      { action: 'CREATE_BOOKING', callOfferId }
    );

    return NextResponse.json({ error: 'Booking failed' }, { status: 500 });
  }
}
```

### Exemple 3 : Webhook Stripe

```
// app/api/payments/webhook/route.ts
import { logWebhookEvent } from '@/lib/system-logger';

export async function POST(request: NextRequest) {
  try {
    const event = await verifyStripeWebhook(request);
    await processWebhookEvent(event);

    // Log webhook réussi
    await logWebhookEvent('STRIPE', event.type, true, {
      eventId: event.id,
      objectType: event.data.object.object,
    });

    return NextResponse.json({ received: true });
  } catch (error) {
    // Log webhook échoué
    await logWebhookEvent('STRIPE', event.type, false, {
      eventId: event.id,
      errorMessage: error.message,
    });

    return NextResponse.json({ error: 'Webhook failed' }, { status: 500 });
  }
}
```

### Exemple 4 : Action admin

```
// app/api/admin/payouts/[id]/approve/route.ts
import { logAdminAction } from '@/lib/system-logger';
import { LogLevel } from '@prisma/client';

export async function POST(request: NextRequest, { params }: { params: { id:
string } }) {
  try {
    const payout = await approvePayout(params.id, adminId);

    // Log action admin
    await logAdminAction(
      'PAYOUT_APPROVED',
      adminId,
      `Admin approved payout ${params.id}`,
      LogLevel.INFO,
      {
        payoutId: params.id,
        creatorId: payout.creatorId,
        amount: payout.amount,
        currency: payout.currency,
      }
    );

    return NextResponse.json({ success: true, payout });
  } catch (error) {
    // Log erreur critique
    await logAdminAction(
      'PAYOUT_APPROVAL_FAILED',
      adminId,
      `Failed to approve payout ${params.id}`,
      LogLevel.CRITICAL,
      {
        payoutId: params.id,
        errorMessage: error.message,
      }
    );

    return NextResponse.json({ error: 'Approval failed' }, { status: 500 });
  }
}
```

---

## API Routes

### GET /api/admin/system-logs

Récupérer les logs système avec filtrage et pagination.

**Query Parameters :**

- `level` : LogLevel (INFO, WARNING, ERROR, CRITICAL)
- `actor` : LogActor (USER, CREATOR, ADMIN, SYSTEM, GUEST)
- `type` : String (recherche partielle, case-insensitive)
- `actorId` : String
- `startDate` : ISO date string

- `endDate` : ISO date string
- `search` : String (recherche dans message, type, actorId)
- `limit` : Number (défaut: 100, max: 500)
- `page` : Number (défaut: 1)
- `orderBy` : ‘asc’ | ‘desc’ (défaut: ‘desc’)

**Exemple :**

```
GET /api/admin/system-logs?level=ERROR&actor=USER&page=1&limit=100
```

**Response :**

```
{
  "success": true,
  "logs": [
    {
      "id": "log_xxxxx",
      "level": "ERROR",
      "type": "API_ERROR",
      "actor": "USER",
      "actorId": "user_xxxxx",
      "message": "Failed to create booking",
      "metadata": { ... },
      "createdAt": "2024-01-15T10:30:00Z"
    }
  ],
  "pagination": {
    "totalCount": 1500,
    "totalPages": 15,
    "currentPage": 1,
    "limit": 100,
    "offset": 0,
    "hasMore": true
  }
}
```

## DELETE /api/admin/system-logs

Supprimer des logs par plage de dates ou par filtres.

**Request Body :**

```
{
  "deleteType": "dateRange",
  "startDate": "2024-01-01T00:00:00Z",
  "endDate": "2024-01-31T23:59:59Z",
  "level": "INFO"
}
```

**Response :**

```
{
  "success": true,
  "deletedCount": 1500,
  "message": "1500 log(s) supprimé(s) avec succès"
}
```

### POST /api/cron/cleanup-logs

Déclencher le nettoyage automatique des logs.

**Headers :**

```
Authorization: Bearer <CRON_SECRET>
```

**Response :**

```
{
  "success": true,
  "message": "Log cleanup completed successfully",
  "stats": {
    "infoDeleted": 1200,
    "warningDeleted": 300,
    "errorDeleted": 50,
    "totalDeleted": 1550,
    "durationMs": 1234
  }
}
```

---

## Optimisations et performances

### Index de base de données

Le modèle Log est optimisé avec plusieurs index pour des requêtes rapides :

```
@@index([createdAt])        // Tri chronologique
@@index([level])            // Filtrage par niveau
@@index([type])             // Filtrage par type
@@index([actor])            // Filtrage par acteur
@@index([actorId])          // Recherche par acteur spécifique
@@index([level, createdAt]) // Requêtes de rétention optimisées
```

### Pagination obligatoire

- Toutes les requêtes sont paginées côté backend
- Limite maximale : 500 logs par requête
- Pas de chargement massif côté frontend

### Logs asynchrones

Toutes les fonctions de logging sont asynchrones et n’attendent pas la confirmation d’écriture pour ne pas bloquer le flux applicatif.

### Gestion d’erreurs

En cas d’erreur lors de l’écriture d’un log, l’erreur est capturée et loggée dans la console mais ne bloque jamais l’application :

```
try {
  await prisma.log.create({ data: logEntry });
} catch (error) {
  // Logging should never crash the application
  console.error('[SystemLog Error]', error);
}
```

---

## Évolution future

### Fonctionnalités prévues

1. **Alertes automatiques** : Envoyer des notifications aux admins pour les logs CRITICAL
2. **Export de logs** : Télécharger les logs en CSV/JSON pour analyse externe
3. **Graphiques et statistiques** : Visualisation des tendances d’erreurs
4. **Intégration avec services externes** : Sentry, DataDog, etc.
5. **Recherche avancée** : Full-text search avec Elasticsearch

### Extensions possibles

- Ajouter des logs pour les actions de modération
- Logger les modifications de configuration
- Tracer les migrations de données
- Logger les exports de données RGPD

---

## Support et maintenance

Pour toute question ou problème concernant le système de logs :

1. Vérifier la page admin : `/dashboard/admin/system-logs`
2. Exécuter manuellement le cleanup : `npm run cleanup-logs`
3. Consulter les logs de la console serveur
4. Vérifier les variables d’environnement ( `CRON_SECRET` , `DATABASE_URL` )

---

## Résumé des fichiers

### Modèles et migrations

- `prisma/schema.prisma` : Modèle Log et enums
- `prisma/migrations/*/migration.sql` : Migration SQL

### Services et utilitaires

- `lib/system-logger.ts` : Service de logging centralisé

## API Routes

- `app/api/admin/system-logs/route.ts` : GET et DELETE pour les logs
- `app/api/cron/cleanup-logs/route.ts` : Cron job de nettoyage

## Pages admin

- `app/dashboard/admin/system-logs/page.tsx` : Interface admin

## Scripts

- `scripts/cleanup-logs.ts` : Script manuel de nettoyage

## Configuration

- `vercel.json` : Configuration des cron jobs
- `package.json` : Scripts npm

---

**Version** : 1.0.0
**Date** : Décembre 2024
**Auteur** : Callastar Team