📊 Système de Logging - Call a Star

Ce document décrit l'utilisation du système de logging complet pour auditer **100% des emails et des crons** dans l'application Call a Star.

🎯 Objectifs

- **Audit complet** : Tous les emails envoyés et toutes les exécutions de cron jobs sont enregistrés
- **Traçabilité** : Chaque log contient le contexte complet (bookingId, userId, etc.)
- **Debugging facile** : Les erreurs incluent les stack traces complètes
- **Type-safe** : Utilisation de TypeScript pour éviter les erreurs

📁 Structure

Modèle Prisma (`prisma/schema.prisma`)

```
model Log {
  id        String    @id @default(cuid())
  type      LogType   // Email, cron, booking, payment, etc.
  status    LogStatus // SUCCESS ou ERROR
  message   String    // Message lisible pour l'admin
  context   Json?     // bookingId, userId, offerId, roomId, etc.
  error     String?   // Stack trace ou message d'erreur détaillé
  createdAt DateTime  @default(now())

  // Index pour les requêtes fréquentes
  @@index([type])
  @@index([status])
  @@index([createdAt])
  @@index([type, status])
  @@index([type, createdAt])
  @@index([status, createdAt])
}
```

Enums

```
enum LogType {
  EMAIL_SENT,
  EMAIL_ERROR,
  CRON_RUN,
  CRON_ERROR,
  DAILY_ROOM_DELETED,
  DAILY_ROOM_ERROR,
  DAILY_ROOM_CREATED,
  BOOKING_CREATED,
  BOOKING_ERROR,
  PAYMENT_SUCCESS,
  PAYMENT_ERROR,
  PAYMENT_REFUND,
  PAYOUT_SUCCESS,
  PAYOUT_ERROR,
  STRIPE_WEBHOOK,
  STRIPE_WEBHOOK_ERROR,
  NOTIFICATION_SENT,
  NOTIFICATION_ERROR,
  SYSTEM_ERROR,
  API_ERROR
}

enum LogStatus {
  SUCCESS,
  ERROR
}
```

🚀 Utilisation

Import

```
import {
  logEmailSent,
  logEmailError,
  logCronRun,
  logCronError,
  logDailyRoomDeleted,
  logDailyRoomError,
  logDailyRoomCreated,
  logBookingCreated,
  logBookingError,
  logPaymentSuccess,
  logPaymentError,
  logSuccess,
  logError,
  LogType,
  LogStatus
} from "@/lib/logger";
```

📧 Logging des Emails

Email envoyé avec succès

```
// Après l'envoi d'un email de confirmation de réservation
await logEmailSent(
  "clx123456", // bookingId
  "user789",   // userId
  "booking_confirmation", // emailType
  {
    recipientEmail: "user@example.com",
    subject: "Votre réservation est confirmée !",
    creatorId: "creator456"
  }
);
```

Email échoué

```
// Si l'envoi d'un email échoue
try {
  await resend.emails.send({
    // ... config email
  });
  await logEmailSent(bookingId, userId, "booking_confirmation");
} catch (error) {
  await logEmailError(
    bookingId,
    userId,
    "booking_confirmation",
    error,
    {
      recipientEmail: "user@example.com",
      subject: "Votre réservation est confirmée !",
      errorType: "RESEND_API_ERROR"
    }
  );
  throw error; // Re-throw pour gérer l'erreur
}
```

Exemples d’emails à logger

```
// Email de confirmation de réservation
await logEmailSent(bookingId, userId, "booking_confirmation");

// Email de rappel 24h avant l'appel
await logEmailSent(bookingId, userId, "booking_reminder_24h");

// Email de rappel 1h avant l'appel
await logEmailSent(bookingId, userId, "booking_reminder_1h");

// Email d'annulation
await logEmailSent(bookingId, userId, "booking_cancelled");

// Email au créateur pour nouvelle réservation
await logEmailSent(bookingId, creatorId, "creator_new_booking");

// Email de paiement reçu
await logEmailSent(bookingId, creatorId, "payment_received");
```

⏰ Logging des Cron Jobs

Cron exécuté avec succès

```
// api/cron/cleanup-daily-rooms.ts
export async function GET(request: Request) {
  const startTime = Date.now();
  let roomsDeleted = 0;

  try {
    // Récupérer les rooms à supprimer
    const expiredBookings = await prisma.booking.findMany({
      where: {
        status: "COMPLETED",
        callOffer: {
          dateTime: {
            lt: new Date(Date.now() - 24 * 60 * 60 * 1000) // 24h ago
          }
        }
      }
    });

    // Supprimer chaque room
    for (const booking of expiredBookings) {
      if (booking.dailyRoomName) {
        await daily.deleteRoom(booking.dailyRoomName);
        await logDailyRoomDeleted(
          booking.dailyRoomName,
          booking.id,
          {
            roomUrl: booking.dailyRoomUrl || undefined,
            deletedBy: "cron_job"
          }
        );
        roomsDeleted++;
      }
    }

    const duration = Date.now() - startTime;

    // Logger le succès du cron
    await logCronRun(
      "cleanup_daily_rooms",
      roomsDeleted,
      duration,
      {
        startTime: new Date(startTime).toISOString(),
        endTime: new Date().toISOString(),
        totalBookingsChecked: expiredBookings.length
      }
    );

    return NextResponse.json({
      success: true,
      roomsDeleted,
      duration: `${Math.round(duration / 1000)}s`
    });
  } catch (error) {
    const duration = Date.now() - startTime;

    // Logger l'erreur du cron
    await logCronError(
      "cleanup_daily_rooms",
      error,
      {
        roomsDeletedBeforeError: roomsDeleted,
```

```
        duration,
        startTime: new Date(startTime).toISOString()
      }
    );

    return NextResponse.json(
      { error: "Cron job failed" },
      { status: 500 }
    );
  }
}
```

Cron d’envoi de rappels d’emails

```
// api/cron/send-reminders.ts
export async function GET(request: Request) {
  const startTime = Date.now();
  let emailsSent = 0;

  try {
    // Trouver les réservations dans 24h
    const upcomingBookings = await prisma.booking.findMany({
      where: {
        status: "CONFIRMED",
        callOffer: {
          dateTime: {
            gte: new Date(Date.now() + 23 * 60 * 60 * 1000),
            lte: new Date(Date.now() + 25 * 60 * 60 * 1000)
          }
        }
      },
      include: {
        user: true,
        callOffer: {
          include: {
            creator: {
              include: {
                user: true
              }
            }
          }
        }
      }
    });

    // Envoyer un email de rappel pour chaque réservation
    for (const booking of upcomingBookings) {
      try {
        await sendBookingReminderEmail(booking);
        await logEmailSent(
          booking.id,
          booking.userId,
          "booking_reminder_24h",
          {
            recipientEmail: booking.user.email,
            creatorName: booking.callOffer.creator.user.name,
            callDateTime: booking.callOffer.dateTime.toISOString()
          }
        );
        emailsSent++;
      } catch (emailError) {
        await logEmailError(
          booking.id,
          booking.userId,
          "booking_reminder_24h",
          emailError,
          {
            recipientEmail: booking.user.email
          }
        );
        // Continue avec les autres emails même en cas d'erreur
      }
    }

    const duration = Date.now() - startTime;
```

```
    await logCronRun(
      "send_booking_reminders_24h",
      emailsSent,
      duration,
      {
        totalBookingsFound: upcomingBookings.length,
        emailsSent,
        emailsFailed: upcomingBookings.length - emailsSent
      }
    );

    return NextResponse.json({
      success: true,
      emailsSent,
      duration: `${Math.round(duration / 1000)}s`
    });
  } catch (error) {
    await logCronError(
      "send_booking_reminders_24h",
      error,
      {
        emailsSentBeforeError: emailsSent
      }
    );

    return NextResponse.json(
      { error: "Cron job failed" },
      { status: 500 }
    );
  }
}
```

🎥 Logging des Opérations Daily.io

Création d’une room

```
// Lors de la création d'une réservation
try {
  const room = await daily.createRoom({
    name: `call-${booking.id}`,
    privacy: "private",
    properties: {
      start_audio_off: true,
      start_video_off: true,
      enable_recording: "cloud"
    }
  });

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

  await logDailyRoomCreated(
    room.name,
    booking.id,
    {
      roomUrl: room.url,
      creatorId: booking.callOffer.creatorId,
      userId: booking.userId,
      privacy: "private"
    }
  );
} catch (error) {
  await logDailyRoomError(
    `call-${booking.id}`,
    booking.id,
    error,
    {
      operation: "create",
      userId: booking.userId
    }
  );
  throw error;
}
```

Suppression d’une room

```
// Après la fin d'un appel
try {
  if (booking.dailyRoomName) {
    await daily.deleteRoom(booking.dailyRoomName);

    await logDailyRoomDeleted(
      booking.dailyRoomName,
      booking.id,
      {
        roomUrl: booking.dailyRoomUrl || undefined,
        deletedBy: "manual", // ou "cron_job"
        creatorId: booking.callOffer.creatorId,
        userId: booking.userId
      }
    );
  }
} catch (error) {
  await logDailyRoomError(
    booking.dailyRoomName!,
    booking.id,
    error,
    {
      operation: "delete",
      roomUrl: booking.dailyRoomUrl || undefined
    }
  );
  // Ne pas throw ici pour éviter de bloquer la finalisation de la réservation
}
```

💰 Logging des Paiements et Réservations

Création de réservation

```
// POST /api/bookings
try {
  const booking = await prisma.booking.create({
    data: {
      userId: session.user.id,
      callOfferId: callOfferId,
      totalPrice: callOffer.price,
      status: "PENDING"
    }
  });

  await logBookingCreated(
    booking.id,
    session.user.id,
    {
      creatorId: callOffer.creatorId,
      offerId: callOfferId,
      amount: Number(callOffer.price),
      currency: callOffer.currency,
      callDateTime: callOffer.dateTime.toISOString()
    }
  );

  return booking;
} catch (error) {
  await logBookingError(
    session.user.id,
    error,
    {
      offerId: callOfferId,
      amount: Number(callOffer.price),
      currency: callOffer.currency
    }
  );
  throw error;
}
```

Paiement réussi

```
// Webhook Stripe - payment_intent.succeeded
try {
  const payment = await prisma.payment.update({
    where: { stripePaymentIntentId: paymentIntent.id },
    data: { status: "SUCCEEDED" }
  });

  await logPaymentSuccess(
    payment.id,
    payment.bookingId,
    Number(payment.amount),
    payment.currency,
    {
      stripePaymentIntentId: paymentIntent.id,
      userId: booking.userId,
      creatorId: booking.callOffer.creatorId
    }
  );
} catch (error) {
  await logPaymentError(
    payment.bookingId,
    error,
    {
      stripePaymentIntentId: paymentIntent.id,
      amount: Number(payment.amount),
      currency: payment.currency
    }
  );
}
```

🔧 Fonctions Génériques

Logger un succès personnalisé

```
import { logSuccess, LogType } from "@/lib/logger";

await logSuccess(
  LogType.NOTIFICATION_SENT,
  "Push notification envoyée avec succès",
  {
    userId: "user123",
    notificationType: "booking_reminder",
    deviceToken: "abc123..."
  }
);
```

Logger une erreur personnalisée

```
import { logError, LogType } from "@/lib/logger";

try {
  // ... code qui peut échouer
} catch (error) {
  await logError(
    LogType.API_ERROR,
    "Échec de l'appel à l'API Stripe",
    error,
    {
      endpoint: "/v1/payment_intents",
      method: "POST",
      statusCode: 500
    }
  );
}
```

📊 Consultation des Logs

Récupérer les logs récents

```
import { getRecentLogs, LogType, LogStatus } from "@/lib/logger";

// Tous les logs récents
const allLogs = await getRecentLogs({ limit: 100 });

// Uniquement les erreurs d'emails
const emailErrors = await getRecentLogs({
  type: LogType.EMAIL_ERROR,
  status: LogStatus.ERROR,
  limit: 50
});

// Logs d'une période spécifique
const logsToday = await getRecentLogs({
  startDate: new Date(new Date().setHours(0, 0, 0, 0)),
  endDate: new Date(),
  limit: 1000
});
```

Obtenir des statistiques

```
import { getLogStats } from "@/lib/logger";

// Stats de tous les temps
const allTimeStats = await getLogStats();

// Stats des 7 derniers jours
const weekStats = await getLogStats({
  startDate: new Date(Date.now() - 7 * 24 * 60 * 60 * 1000),
  endDate: new Date()
});

console.log(weekStats);
// {
//   totalLogs: 1523,
//   successLogs: 1450,
//   errorLogs: 73,
//   successRate: "95.21",
//   errorRate: "4.79",
//   emails: {
//     sent: 456,
//     errors: 12,
//     successRate: "97.44"
//   },
//   crons: {
//     runs: 168,
//     errors: 3,
//     successRate: "98.25"
//   }
// }
```

🧹 Nettoyage des Logs

Script de nettoyage automatique

```
// scripts/cleanup-logs.ts
import { cleanupOldLogs } from "@/lib/logger";

async function main() {
  console.log("🧹 Nettoyage des logs anciens...");

  // Supprimer les logs de plus de 90 jours
  const deletedCount = await cleanupOldLogs(90);

  console.log(`✅ ${deletedCount} logs supprimés`);
}

main();
```

Cron job de nettoyage

```
// api/cron/cleanup-logs.ts
import { cleanupOldLogs, logCronRun, logCronError } from "@/lib/logger";

export async function GET(request: Request) {
  const startTime = Date.now();

  try {
    // Supprimer les logs de plus de 90 jours
    const deletedCount = await cleanupOldLogs(90);
    const duration = Date.now() - startTime;

    await logCronRun(
      "cleanup_logs",
      deletedCount,
      duration,
      {
        retentionDays: 90,
        deletedCount
      }
    );

    return NextResponse.json({
      success: true,
      deletedCount,
      duration: `${Math.round(duration / 1000)}s`
    });
  } catch (error) {
    await logCronError("cleanup_logs", error);
    return NextResponse.json({ error: "Failed" }, { status: 500 });
  }
}
```

📈 Dashboard Admin (Exemple)

```tsx
// app/admin/logs/page.tsx
import { getRecentLogs, getLogStats, LogStatus } from "@/lib/logger";

export default async function AdminLogsPage() {
  const stats = await getLogStats({
    startDate: new Date(Date.now() - 7 * 24 * 60 * 60 * 1000)
  });

  const recentErrors = await getRecentLogs({
    status: LogStatus.ERROR,
    limit: 20
  });

  return (
    <div>
      <h1>📊 Logs - 7 derniers jours</h1>

      <div className="stats-grid">
        <StatCard title="Total Logs" value={stats.totalLogs} />
        <StatCard title="Taux de succès" value={`${stats.successRate}%`} />
        <StatCard title="Emails envoyés" value={stats.emails.sent} />
        <StatCard title="Emails échoués" value={stats.emails.errors} />
        <StatCard title="Crons exécutés" value={stats.crons.runs} />
        <StatCard title="Crons échoués" value={stats.crons.errors} />
      </div>

      <h2>❌ Erreurs récentes</h2>
      <LogsTable logs={recentErrors} />
    </div>
  );
}
```

✅ Checklist d'Intégration

Pour chaque email envoyé :

- [ ] Ajouter `logEmailSent()` après l'envoi réussi
- [ ] Ajouter `logEmailError()` dans le catch en cas d'échec
- [ ] Inclure le contexte : `bookingId`, `userId`, `emailType`

Pour chaque cron job :

- [ ] Ajouter `const startTime = Date.now()` au début
- [ ] Compter les items traités
- [ ] Ajouter `logCronRun()` en cas de succès
- [ ] Ajouter `logCronError()` dans le catch en cas d'échec
- [ ] Calculer la durée : `Date.now() - startTime`

Pour chaque opération Daily.io :

- [ ] Logger la création avec `logDailyRoomCreated()`
- [ ] Logger la suppression avec `logDailyRoomDeleted()`
- [ ] Logger les erreurs avec `logDailyRoomError()`

Pour chaque paiement :

- [ ] Logger le succès avec `logPaymentSuccess()`
- [ ] Logger les erreurs avec `logPaymentError()`

🔍 Exemple de Requête SQL pour Analyse

```
-- Emails échoués dans les 24 dernières heures
SELECT
  "createdAt",
  "message",
  "context"->>'emailType' as email_type,
  "context"->>'recipientEmail' as recipient,
  "error"
FROM "Log"
WHERE "type" = 'EMAIL_ERROR'
  AND "createdAt" >= NOW() - INTERVAL '24 hours'
ORDER BY "createdAt" DESC;

-- Taux de succès des crons par type
SELECT
  "context"->>'cronType' as cron_type,
  COUNT(*) as total,
  COUNT(CASE WHEN "status" = 'SUCCESS' THEN 1 END) as success_count,
  ROUND(
    COUNT(CASE WHEN "status" = 'SUCCESS' THEN 1 END)::numeric / COUNT(*)::numeric * 10
0,
    2
  ) as success_rate
FROM "Log"
WHERE "type" IN ('CRON_RUN', 'CRON_ERROR')
GROUP BY cron_type;
```

🎯 Bonnes Pratiques

1. **Toujours logger** : Ne jamais oublier de logger un email ou un cron
2. **Contexte complet** : Inclure tous les IDs pertinents (bookingId, userId, etc.)
3. **Messages clairs** : Rédiger des messages lisibles pour les admins
4. **Erreurs détaillées** : Toujours passer l'objet Error complet, pas juste le message
5. **Try/Catch** : Logger les erreurs dans les blocs catch
6. **Performance** : Les logs sont asynchrones et n'impactent pas les performances
7. **Nettoyage** : Configurer un cron pour supprimer les vieux logs (90+ jours)

🚨 Troubleshooting

Les logs ne s'enregistrent pas

Vérifiez que :
- La base de données est accessible
- Les migrations Prisma sont appliquées : `npx prisma migrate deploy`
- Le Prisma Client est généré : `npx prisma generate`

Erreur “LogType is not defined”

```
// ✅ Import correct
import { LogType, logEmailSent } from "@/lib/logger";

// ❌ Import incorrect
import { logEmailSent } from "@/lib/logger";
await logEmailSent(bookingId, userId, LogType.EMAIL_SENT); // LogType n'est pas im-
porté
```

Logs trop volumineux

Configurez un cron job de nettoyage :

```
// Supprimer les logs de plus de 90 jours chaque semaine
await cleanupOldLogs(90);
```

📞 Support

Pour toute question sur le système de logging, contactez l’équipe technique ou consultez la documentation Prisma : https://www.prisma.io/docs