Gosselet Julien
Celisse Kyllian

Université Polytechnique HAUTS-DE-FRANCE

MultiCord : Développement et déploiement d’un bot Discord multifonction

M Benameur

2025 - 2026

Résumé

Le projet **MultiCord** porte sur la conception, le développement et le déploiement complet d'un bot multifonction pour la plateforme Discord, spécialisé dans la diffusion musicale et la gestion de communauté. L'objectif principal de cette réalisation était de mettre en œuvre une architecture client-serveur robuste, capable de traiter des flux musicaux sans jamais impacter la réactivité des outils de modération (kick, ban, mute, etc.).

Pour répondre à ces besoins de performance et garantir une disponibilité 24h/24, l'intégralité de la solution a été déployée sur un **serveur privé virtuel (VPS)** fonctionnant sous Linux. Le VPS centralise tout l'écosystème du projet : l'application principale développée en **TypeScript**, le serveur de traitement du flux musical **Lavalink**, ainsi que la base de données **SQLite** gérée via l'ORM **Prisma**.

Le développement de l'interface s'est concentré sur l'expérience utilisateur en exploitant les fonctionnalités de Discord. L'utilisation de TypeScript a également permis de sécuriser le code et de faciliter la maintenance du projet.

Les résultats finaux valident une infrastructure stable et autonome, capable de gérer des commandes diverses avec une latence minimale.

Introduction

Le projet **MultiCord** s'inscrit dans un contexte où les plateformes de communication instantanée, et particulièrement Discord, occupent une place importante dans la gestion de communautés et le travail collaboratif. Le choix de ce sujet repose sur la volonté de répondre à un besoin croissant : l'accès à des outils de divertissement et de gestion à la fois performants, stables et simples d'utilisation.

1. Contexte du projet

La plupart des solutions audio ou de modération existantes sur Discord souffrent de limitations techniques. L'idée de MultiCord est née de la nécessité de proposer une alternative complète et autonome, capable de se libérer des contraintes des bots publics souvent saturés. Ce projet permet d'explorer des thématiques techniques et variées : de l'administration système sous Linux au développement d'applications en **TypeScript**, en passant par la modélisation de données via l'ORM **Prisma**.

2. Objectifs du projet

L'objectif principal de ce travail est de réaliser un bot multifonction robuste capable de :

- **Garantir une lecture audio fluide** via l'intégration d'un serveur **Lavalink v4**, déplaçant ainsi le traitement lourd du flux sonore pour ne pas ralentir le bot.
- **Gérer la modération et la persistance des données** (historique des avertissements, configurations serveurs) en utilisant une base de données **SQLite**, garantissant ainsi que les informations ne sont pas perdues lors d'un redémarrage.
- **Proposer une interface utilisateur intuitive** en exploitant les fonctionnalités de l'API Discord, notamment les composants interactifs tels que les boutons et les menus déroulants.
- **Garantir une disponibilité constante** du service grâce à un déploiement sur un **serveur privé virtuel (VPS)**. Ce serveur héberge l'intégralité de l'écosystème : le bot, le moteur audio ainsi que la base de données.

Cahier des charges

1. Problématique et besoins

La problématique centrale de MultiCord est de proposer une solution capable de gérer la musique et la modération d'un serveur Discord, tout en restant stable.

Besoins fonctionnels :

- **Module Audio :** Recherche et lecture de pistes depuis SoundCloud, gestion d'une file d'attente (playlist) et interface de contrôle interactive via des boutons (Play/Pause, Skip, Stop).
- **Module Modération :** Outils de gestion de communauté (Kick, Ban, Mute) et mise en place d'un système de **"Warnings"** (avertissements).
- **Persistance des données :** Sauvegarde durable des sanctions et des configurations serveurs pour qu'elles restent accessibles même après un redémarrage.
- **Disponibilité :** Le service doit être opérationnel 24h/24 sans dépendre d'une machine locale.

2. Contraintes du projet

- **Contraintes matérielles (Hébergement) :** L'intégralité de l'écosystème (Bot + Lavalink + Base de données) est hébergée sur un **VPS Linux** disposant de ressources limitées (**4 Go de RAM**).
- **Contraintes techniques :** Nécessité de séparer la logique applicative du moteur de flux audio (Lavalink) pour garantir une écoute fluide, sans micro-coupures lors de l'exécution de commandes.
- **Sécurité et réseau :** Configuration des ports du VPS (notamment le port 2333 pour Lavalink) et gestion de la base de données locale pour garantir la confidentialité des données de modération.

3. Outils et technologies envisagés

- **Langage : TypeScript**, pour sa capacité à sécuriser le code via le typage statique, ce qui est important pour un projet gérant de nombreuses interactions.
- **Environnement d'exécution : Node.js**, choisi pour sa gestion idéale pour traiter de multiples requêtes d'utilisateurs en temps réel sur un serveur Discord.
- **Environnement Audio : Lavalink v4** (tournant sous **Java 21**), pour son utilisation dans le décodage et le streaming audio.
- **Gestion des données :** L'ORM **Prisma** associé à une base de données **SQLite**. Ce choix permet une gestion des données directement sur le VPS.
- **Bibliothèques clés : Discord.js** pour la communication avec l'API Discord et **Shoukaku** pour faire le lien entre le code TypeScript et le serveur Lavalink.
- **Gestion de production : PM2** (Process Manager), utilisé pour surveiller le bot et le relancer automatiquement en cas de crash, garantissant une disponibilité.

Conception

1. Architecture générale du projet

Le projet **MultiCord** repose sur une architecture de type client-serveur.

- **Le Client (Bot Discord) :** Développé en TypeScript, il assure la réception des commandes utilisateurs et la gestion de l'interface visuelle.
- **Le Serveur Audio (Lavalink) :** Agit comme un moteur de rendu autonome qui traite les flux musicaux.
- **Le Lien (Shoukaku) :** Cette bibliothèque fait office de passerelle entre le bot et Lavalink.

2. Organisation du code et modélisation

L'arborescence du projet est structurée de façon à qu'il soit séparée par fonction :

- **`/src/commands`** : Regroupe les interactions Slash par catégories (Musique, Modération, Util).
- **`/src/events`** : Gère les réponses aux événements Discord (connexion, messages, interactions).
- **`/prisma`** : Contient le fichier schema.prisma définissant la structure de la base de données.
- **`database.ts`** : Point d'entrée de l'instance Prisma pour les requêtes vers la base SQLite.
- **`src/index.ts`** : Point d'entrée principal du bot et initialisation des clients.
- **`/lavalink`** : Contient le .jar et le fichier de configuration .yml.

3. Justification des choix techniques

Chaque technologie a été sélectionnée pour répondre à une contrainte précise du cahier des charges :

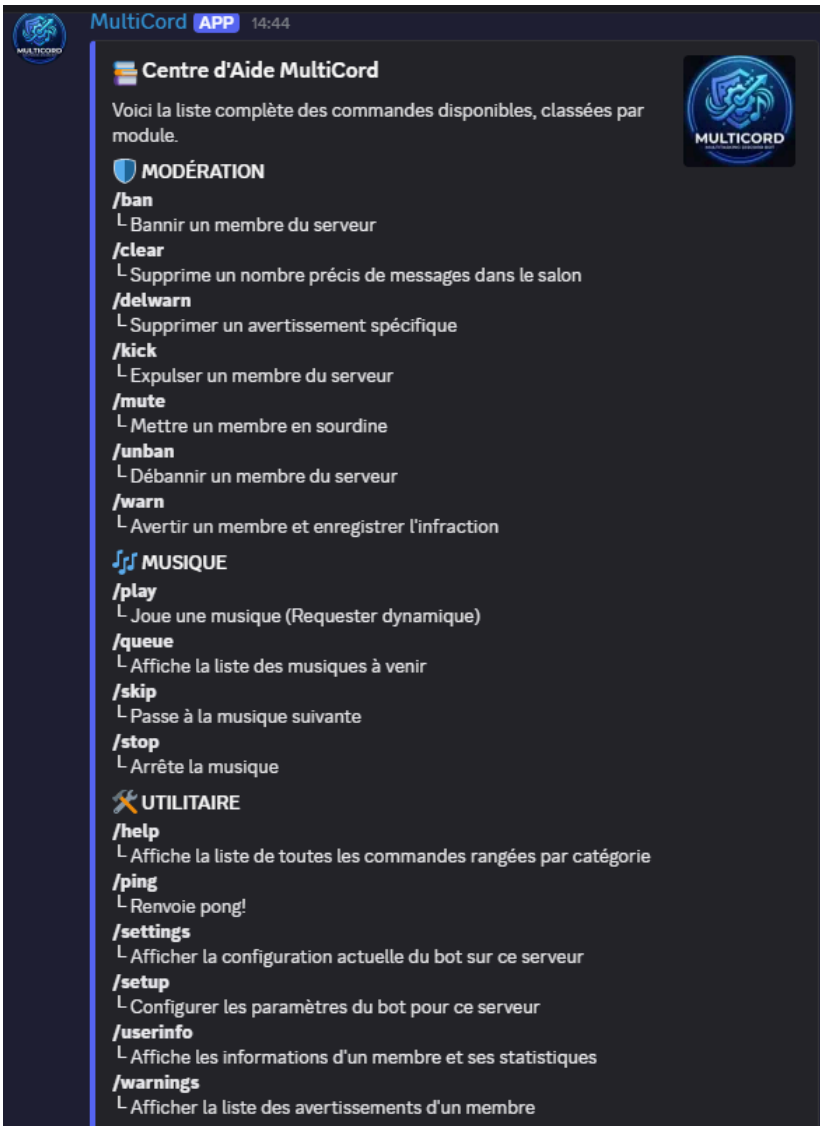
- **TypeScript :** Choisi pour son typage statique qui réduit drastiquement les erreurs de logique en phase de production.
- **Node.js :** Choisi comme environnement d'exécution pour sa gestion performante des entrées/sorties, permettant de traiter simultanément un grand nombre d'interactions sans bloquer le processus principal.
- **Lavalink v4 :** Utilisé pour sa capacité à gérer les flux audio et sa stabilité reconnue dans le milieu du développement de bots musicaux.
- **PM2 :** Utilisé pour la gestion des processus, garantissant que le service redémarre automatiquement en cas d'erreur critique sur le VPS.
- **Prisma :** Intégré pour simplifier et sécuriser les interactions avec la base de données SQLite. Son typage automatique s'accorde parfaitement avec TypeScript, garantissant une gestion fiable de la persistance des données.

Implémentation - Partie 1 (Modération et Gestion)

1. Commandes utilitaires

Pour améliorer l'expérience utilisateur, un ensemble de commandes utilitaires a été implémenté (dossier /util) :

- **/settings** : Permet aux administrateurs d'afficher la configuration actuelle du bot.
- **/userinfo** : Affiche les informations détaillées d'un membre (date d'arrivée, rôles, identifiants).
- **/help** : Système d'aide listant les commandes disponibles.
- **/setup** : Permet de configurer le salon de logs du serveur.
- **/ping** : Commande permettant de vérifier si le bot est actif.
- **/warnings** : Permet d'afficher la liste des avertissements reçus par un membre.



2. Outils de modération

- **Commandes d'expulsion et de bannissement (/kick, /ban) :** Ces outils permettent de retirer définitivement ou temporairement un utilisateur du serveur. L'implémentation inclut des vérifications de hiérarchie pour s'assurer que le bot ne peut pas agir sur des membres ayant un rôle supérieur à lui-même.
- **Gestion des messages (/clear) :** Une commande de suppression de masse a été développée pour permettre de nettoyer rapidement un salon textuel (jusqu'à 100 messages simultanément). Cette fonction est essentielle pour effacer les contenus inappropriés ou le spam. Elle permet de cibler soit un utilisateur précis, soit l'ensemble des messages.
- **Système de Mute/Timeout (/mute) :** Le bot peut "réduire au silence" un utilisateur pour une durée déterminée, l'empêchant de parler ou d'écrire sans pour autant l'exclure du serveur.
- **Persistance de la modération (/warn) :** MultiCord utilise **Prisma** pour enregistrer chaque avertissement (warn) émis par les modérateurs. Le fichier warnings.ts interagit directement avec la base de données SQLite pour permettre un historique complet des sanctions par membre.
- **Annulation de sanctions (/unban, /delwarn) :** Permet de lever un bannissement ou un avertissement à un utilisateur choisi.


```
model Warn {
  id          Int      @id @default(autoincrement())
  guildId     String   // ID du serveur
  userId      String   // ID de l'utilisateur puni
  moderatorId String   // ID du modérateur
  reason      String   // Raison du warn
  createdAt   DateTime @default(now())
}
```

3. Gestion des permissions et sécurité

- **Vérification des droits :** Chaque commande de modération vérifie si l'utilisateur qui l'exécute possède les permissions requises (ex: Administrateur). Si les droits sont insuffisants, le bot refuse l'action avec un message d'erreur.
- **Journalisation des actions (Logging) :** Bien que non visuel pour l'utilisateur final, le bot génère des logs dans la console du serveur pour chaque action de modération effectuée, permettant une traçabilité pour le propriétaire du serveur.

4. Expérience utilisateur

- **Réponses éphémères :** Pour ne pas encombrer les salons publics, certaines confirmations de modération sont visibles uniquement par le modérateur.
- **Messages d'erreur clairs :** En cas d'échec (utilisateur introuvable, permissions manquantes), le bot fournit une explication précise plutôt qu'un code d'erreur générique, ce qui facilite le travail de l'équipe d'administration.

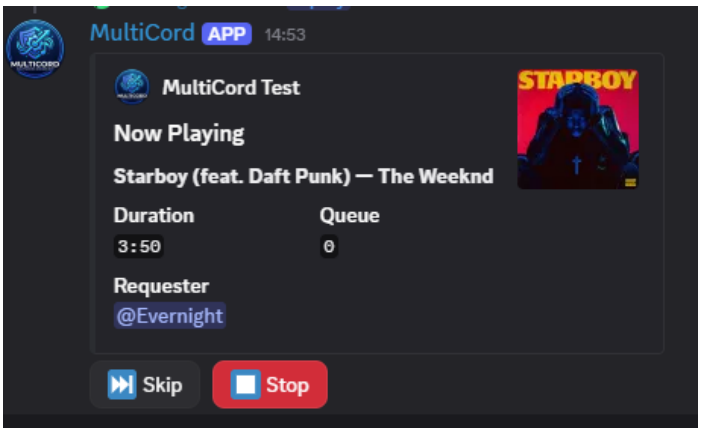
Implémentation - Partie 2 (Musique)

1. Fonctionnalités Musicales

- **Commande `/play` :** Permet la recherche dynamique de morceaux. Elle intègre un système d'autocomplétion qui suggère des titres en temps réel à l'utilisateur pendant qu'il tape sa recherche.
- **Gestion de la file d'attente (/queue) :** Les morceaux sont ajoutés à une liste de lecture. Chaque nouveau titre s'ajoute à la suite de la musique en cours.
- **Contrôle manuel et interactif (/skip, /stop) :** L'utilisateur garde le plein contrôle sur la session grâce aux commandes /skip et /stop permettant une gestion flexible.
- **Automatisation et optimisation :** Le bot détecte la fin d'une piste pour lancer la suivante sans interruption. Si la file d'attente est vide ou après une commande d'arrêt, le bot se déconnecte proprement du salon vocal pour économiser les ressources du serveur.

2. Interface utilisateur interactive

- **Cartes de lecture :** Chaque musique lancée génère une carte visuelle affichant le titre, l'auteur, la durée et la pochette du morceau.
- **Contrôles par boutons :** Au lieu de taper des commandes textuelles, les utilisateurs disposent de boutons interactifs "Skip" et "Stop" directement sous la carte de lecture. Ces boutons sont programmés pour se désactiver dès que la musique change ou s'arrête.



Implémentation - Difficultés et Solutions

1. Problématique de configuration réseau (Port 2333)

L'une des premières difficultés majeures a été la difficulté pour le bot de se connecter au serveur audio Lavalink, malgré une configuration qui semblait correcte en apparence.

- **Le problème :** Le bot renvoyait une erreur de connexion refusée alors que le serveur Lavalink était marqué comme actif.
- **La démarche suivie :** Une analyse des logs système via la commande netstat a permis de découvrir que le serveur écoutait sur le port par défaut (8080) au lieu du port personnalisé (2333) défini dans le code du bot.
- **La solution :** Une modification manuelle du fichier de configuration application.yml a été effectuée pour forcer l'ouverture correspondante des ports.

2. Erreurs de configuration YAML

- **Le problème :** Le moteur audio refusait de démarrer car il ne parvenait pas à interpréter les **variables d'environnement** renseignées directement dans le fichier .yml.
- **La démarche suivie :** Une analyse des logs a révélé que le système tentait de lire les variables comme du texte brut au lieu d'extraire leurs valeurs réelles.
- **La solution :** Le fichier a été corrigé en remplaçant les appels de variables incompatibles par des valeurs statiques.

3. Optimisation de l'interface utilisateur

Une difficulté ergonomique est apparue : les utilisateurs pouvaient cliquer plusieurs fois sur "Skip" même après la fin d'une musique, générant des messages d'erreur.

- **La solution :** L'implémentation d'une fonction de nettoyage automatique qui modifie le message original pour **retirer les boutons interactifs** dès que la musique se termine ou est passée. Cela garantit une interface toujours cohérente avec l'état réel du bot.

Tests et Validations

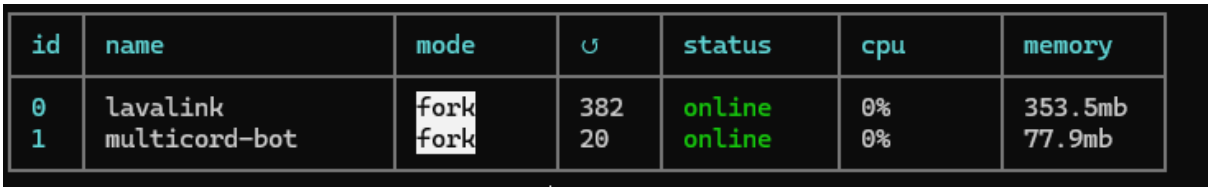
1. Méthodologie de test

Pour garantir un service sans interruption, plusieurs types de tests ont été réalisés :

- **Tests fonctionnels unitaires :** Chaque commande (/play, /skip, /stop, /kick, /clear, …) a été testée manuellement pour vérifier la validité des permissions et la précision des réponses.
- **Tests de scénarios limites :** Simulation de comportements imprévus, comme le skip de la dernière musique de la file ou l'ajout simultané de titres par une dizaine d'utilisateurs.
- **Tests de résilience réseau :** Coupure volontaire de la connexion entre le bot et Lavalink pour valider la capacité de reconnexion automatique du wrapper Shoukaku sans crash de l'application.

2. Résultats obtenus

- **Qualité audio :** La lecture est continue, confirmant que le paramétrage du tampon dans le fichier de configuration de Lavalink est optimal pour les ressources du VPS.
- **Résilience réseau :** Les tests de coupure ont validé l'efficacité de **Shoukaku** : le bot rétablit automatiquement sa liaison avec Lavalink dès que le service audio est de nouveau disponible, sans nécessiter de redémarrage complet de l'application.
- **Gestion de la charge :** Le bot a maintenu ses performances lors de l'ajout simultané de titres par plusieurs utilisateurs, confirmant que l'architecture gère efficacement les files d'attente volumineuses.



id	name	mode	↺	status	cpu	memory
0	lavalink	fork	382	online	0%	353.5mb
1	multicord-bot	fork	20	online	0%	77.9mb

3. Limites identifiées et pistes d'amélioration

- **Limites techniques :** SoundCloud peut occasionnellement bloquer les recherches si elles sont trop fréquentes depuis l'adresse IP unique du VPS.
- **Améliorations envisagées :** L'ajout du flux Youtube permettant un plus large catalogue de musique.
- **Pistes futures :** L'ajout de la possibilité de créer des playlists.

Conclusion et Perspectives

1. Bilan du projet

La réalisation de **MultiCord** a été une expérience enrichissante, couvrant l'ensemble du cycle de vie d'une application, de la conception au déploiement en production. Ce travail a permis de mettre en pratique des concepts avancés de programmation et d'administration système Linux. Malgré les difficultés liées à la configuration réseau et aux contraintes de sécurité des plateformes de streaming, chaque difficulté a été une occasion d'apprendre et de progresser techniquement.

2. Objectifs atteints

- **Infrastructure stable :** Le serveur Lavalink est opérationnel sur VPS et traite les flux audio.
- **Interface intuitive :** L'utilisation des composants interactifs a permis de simplifier l'interaction des utilisateurs.
- **Polyvalence :** Le bot remplit avec succès sa double mission de divertissement (musique) et de gestion de communauté (modération).
- **Disponibilité :** L'intégration de PM2 garantit un service opérationnel 24/7 avec une reconnexion automatique en cas d'incident.

3. Perspectives d’évolution :

- **Dashboard Web** : Création d’une interface graphique permettant aux administrateurs de configurer le bot sans utiliser les commandes Discord.
- **Système d'expérience (XP)** : Utilisation de la base de données existante pour instaurer un système de niveaux récompensant l'activité des membres.

Annexes et Bibliographie

1. Annexes

- **Dépôt GitHub** : L'intégralité du code source du projet MultiCord, est accessible en ligne à l'adresse suivante : **https://github.com/KyllianCel/MultiCord**

2.Bibliographie / Webographie

- **Documentation officielle de Discord.js** : Référence principale pour l'utilisation de l'API Discord, la gestion des interactions "Slash" et l'implémentation des composants visuels.
- **Shoukaku Documentation** : Guide technique essentiel pour l'intégration de Lavalink dans un environnement Node.jst.
- **Lavalink v4 Specifications** : Documentation technique concernant le moteur audio, le protocole de communication et la gestion des sources de streaming.
- **TypeScript Documentation** : Ressources consultées pour l'optimisation du typage statique et la gestion des processus.
- **Articles, tutoriels et blogs communautaires**: Diverses sources consultées pour la résolution des problématiques