Scan de Satellites et Console de Gestion Web

« Application web Django pour gérer les configurations de chaînes TV MuMuDVB, incluant scan satellite et synchronisation IPTV GitHub. »

Par Vasco Valadares Semana – En 2023-Actuel – Projet en Entreprise

Table des Matières

1. Introduction

Ce projet concerne le "Channels Manager", une application web développée avec Django, conçue spécifiquement pour la gestion des configurations de chaînes de télévision avec MuMuDVB. La mission principale de cette application est de permettre une administration centralisée de divers éléments tels que les catégories de chaînes, les clients, les pays, les fréquences et les chaînes elles-mêmes, le tout accessible via une interface web ou des commandes dans le terminal. L'objectif est de fournir une solution complète pour l'automatisation et la gestion des chaînes de télévision/radio par satellite ainsi que des flux IPTV, en s'appuyant sur une intégration étroite entre les données issues des scans satellites (via MuMuDVB) et les informations provenant d'un référentiel GitHub de chaînes IPTV.

2. Objectifs du Projet

- **Gestion centralisée des configurations de chaînes TV pour MuMuDVB via une interface web** : Le projet vise une gestion centralisée et simplifiée des configurations de chaînes TV. Une application web Django offre une interface pour administrer les catégories, clients, pays, fréquences, listes de chaînes, serveurs, adapteurs, etc. Cela assure un contrôle précis et une vue d'ensemble des paramètres de diffusion ;
- **Automatisation du scan de fréquences satellites** : L'application a pour objectif d'automatiser le balayage des fréquences pour de multiples satellites, comme 13.0°E ou 19.2°E, en utilisant des adaptateurs DVB (Digital Video Broadcast). Ce processus interagit avec des outils de scan externes et intègre les données recueillies dans la base de données ;
- **Synchronisation et matching avec des référentiels de chaînes IPTV via GitHub** : Un objectif essentiel est de maintenir la base de données locale à jour en synchronisant les informations des chaînes IPTV depuis des référentiels externes, notamment GitHub (iptv-org). Le système réalise ensuite un "matching" intelligent, via des algorithmes de "fuzzy matching", pour lier ces chaînes IPTV aux chaînes satellite MuMuDVB existantes ;
- **Fournir une API REST pour l'interaction avec le système** : Le projet expose une API REST complète pour permettre la gestion depuis l'interface utilisateur. Cette API supporte l'authentification, la pagination côté serveur, le tri dynamique, le filtrage des données et communique en JSON.

3. Architecture du Projet

3.1 Structure Globale du Projet

Voici un aperçu de l’arborescence du projet :

```
channels_manager/
├── channels_console/       # Projet Django principal
│   ├── console/            # Application Django principale
│   │   ├── api/            # API REST pour les différentes fonctionnalités
│   │   ├── utils/          # Utilitaires pour l'application
│   │   ├── migrations/     # Migrations de base de données Django
│   │   └── models.py       # Modèles de données
│   ├── static/             # Fichiers statiques (CSS, JS)
│   └── templates/          # Templates HTML
├── functions/              # Fonctions utilitaires spécifiques au projet
├── gen/                    # Fichiers et logs générés
│   ├── db_backups/         # Sauvegardes de base de données
│   ├── SATS/               # Fichiers de configuration satellite
│   └── scans/              # Résultats de scan de fréquences et chaînes
├── main.py                 # Script principal pour l'exécution des commandes
├── paths.py                # Définition des chemins du projet
└── requirements.txt        # Dépendances Python
```

3.2 Choix Technologiques

- **Backend :** Django 5.1+ ;
- **Base de données :** SQLite (développement), MySQL (production) ;
- **Frontend :** HTML/CSS/JavaScript vanilla ;
- **API :** API REST avec JSON ;
- **Autres outils clés :** MuMuDVB, w_scan_cpp, dvbv5-scan.

4. Modèles de Données

```python
class MumuChannels(models.Model):
    id = models.BigAutoField(primary_key=True, db_column="mumu_channel_id")
    name = models.CharField(max_length=255, db_column="mumu_channel_name")
    media = models.CharField(max_length=5, db_column="mumu_channel_media")
    ip = models.CharField(max_length=15, db_column="mumu_channel_ip", unique=True)
    position = models.IntegerField(db_column="mumu_channel_position", null=True)
    country_position = models.IntegerField(db_column="mumu_channel_country_position", null=True)
    to_stream = models.BooleanField(default=True, db_column="mumu_channel_to_stream")
    force_stream = models.BooleanField(default=False, db_column="mumu_channel_force_stream")
    block_stream = models.BooleanField(default=False, db_column="mumu_channel_block_stream")
    github_channel_id = models.ForeignKey(GithubChannels, null=True, on_delete=models.SET_NULL,
                                          db_column="github_channel_id")
    is_github_validated = models.BooleanField(default=False,
db_column="mumu_channel_is_github_validated")
    clients = models.ManyToManyField(Clients, through='ChannelClients')
    # Note: occurrences are accessed via channel_instance.occurrences.all()

    class Meta:
        db_table = 'mumu_channels'
```

```python
class Frequencies(models.Model):
    id = models.BigAutoField(primary_key=True, db_column="frequency_id")
    satellite = models.CharField(max_length=255, db_column="frequency_satellite")
    frequency = models.IntegerField(db_column="frequency")
    polarity = models.CharField(max_length=1, db_column="frequency_polarity")
    standard = models.CharField(max_length=5, db_column="frequency_standard")
    modulation = models.CharField(max_length=4, db_column="frequency_modulation")
    symbol_rate = models.IntegerField(db_column="frequency_symbol_rate")
    date = models.CharField(max_length=255, db_column="frequency_date", null=True)
    ber = models.FloatField(db_column="frequency_ber", null=True)
    signal = models.FloatField(db_column="frequency_signal", null=True)
    snr = models.FloatField(db_column="frequency_snr", null=True)
    ub = models.FloatField(db_column="frequency_ub", null=True)
    ts_discontinuities = models.IntegerField(db_column="frequency_ts_discontinuities", null=True)

    class Meta:
        db_table = 'frequencies'
```

5. Fonctionnalités Principales et Processus Clés

5.1. Synchronisation avec GitHub :

La fonction sync_github_channels() automatise la récupération et la mise à jour des chaînes TV depuis le dépôt GitHub. Elle interroge l'API publique pour obtenir les dernières informations sur les chaînes (identifiant, nom, catégories, langues, pays), puis traite ces

données par lots de 100 pour les synchroniser avec notre base de données. Pour chaque chaîne, elle vérifie si elle existe déjà dans notre système : si c'est une nouvelle chaîne, elle la crée intégralement avec toutes ses relations (catégories, langues, pays) ; si la chaîne existe déjà, elle met à jour uniquement les informations qui ont changé. Le processus gère automatiquement la création de nouvelles catégories, langues ou pays si nécessaire, et s'assure que toutes les associations sont correctement maintenues. Tout au long du processus, un suivi détaillé est effectué avec gestion des erreurs pour garantir l'intégrité des données, même en cas de problème sur une chaîne spécifique.

```python
# Récupération des chaînes depuis l'API GitHub
api = requests.get(api_github, timeout=10)
github_channels: list[dict] = json.loads(api.content)

# Traitement de chaque chaîne avec gestion des transactions atomiques
for github_channel in batch:
    try:
        with transaction.atomic():
            # Extraction et normalisation des données
            github_id = github_channel.get("id", None)
            github_categories = set(github_channel.get("categories", []))
            if len(github_categories) == 0:
                github_categories = {"general"}

            # Récupération ou création de la chaîne dans la base de données
            channel = GithubChannels.objects.filter(id=github_id).first()
            if channel is None:
                # Création d'une nouvelle chaîne
                channel = GithubChannels.objects.create(id=github_id, name=github_name,
is_nsfw=github_is_nsfw, country_id=country)
            else:
                # Mise à jour d'une chaîne existante (seulement si changements)
                changes = []
                if channel.name != github_name:
                    changes.append(f"name: {channel.name} -> {github_name}")
                    channel.name = github_name
                # Autres mises à jour...
```

(Code à titre représentatif uniquement)

5.2. Scan des Fréquences Satellites :

La fonction scan() permet de scanner les fréquences et les chaînes d'un satellite spécifique. Ce processus se décompose en trois étapes principales : d'abord le scan des fréquences avec l'outil w_scan_cpp, puis le scan des chaînes avec dvbv5-scan, et enfin l'importation des données scannées dans la base de données. Le système utilise un adaptateur DVB-S disponible, crée des dossiers datés pour organiser les résultats, et gère les autorisations nécessaires lorsqu'il est exécuté en tant que serveur web.

```python
# Première étape: Scan des fréquences avec w_scan_cpp
def scan_frequencies(satellite: str, adapter: str, datetime_folder: str):
    # Configuration et vérification de l'adaptateur satellite
    device_path = f"/dev/dvb/adapter{adapter}"
    freq_dir = PathsGen.SCANS_DIR / datetime_folder / "frequencies"
    freq_dir.mkdir(parents=True, exist_ok=True)

    # Exécution du scan des fréquences
    scan_cmd = f"w_scan_cpp -f s -s {satellite} -a {device_path}/frontend0 -I -x"
    process = subprocess.run(scan_cmd, shell=True, capture_output=True, text=True)

    # Sauvegarde des résultats dans un fichier de configuration
    with open(freq_dir / f"{satellite}.conf", 'w', encoding='utf-8') as f:
        f.write(process.stdout)

# Deuxième étape: Scan des chaînes sur les fréquences détectées
def scan_channels(satellite: str, adapter: str, datetime_folder: str):
    freq_file = PathsGen.SCANS_DIR / datetime_folder / "frequencies" / f"{satellite}.conf"
    channel_file = PathsGen.SCANS_DIR / datetime_folder / "channels" / f"{satellite}.conf"

    # Utilisation des fréquences détectées pour scanner les chaînes
    scan_cmd = f"dvbv5-scan -a {adapter} -I DVBV5 -l EXTENDED -o {channel_file} {freq_file}"
    subprocess.run(scan_cmd, shell=True, capture_output=True, text=True)
```

(Code à titre représentatif seulement)

5.3. Matching entre Chaînes MuMu et GitHub :

Le processus de matching associe les chaînes captées par satellite (MumuChannels) aux références standardisées du dépôt GitHub (GithubChannels). Ce système utilise un algorithme de similarité textuelle pour trouver la correspondance la plus proche entre les noms des chaînes (fuzzy finding), puis détermine la meilleure source d'émission pour chaque chaîne en fonction de la qualité du signal et de l'absence de brouillage. Cela permet d'enrichir les métadonnées des chaînes et d'optimiser la qualité du service en diffusant uniquement la meilleure version de chaque chaîne.

```python
# Association intelligente des chaînes satellite (MuMu) avec les références GitHub
for mumu_channel in mumu_channels.iterator():
    # Chercher la meilleure correspondance dans le référentiel GitHub
    github_channel = get_best_match(mumu_channel, github_channels.iterator())

    if github_channel:
        # Associer la référence et vérifier l'éligibilité à la diffusion
        mumu_channel.github_channel_id = github_channel
        mumu_channel.to_stream = (
            github_channel.country_id.to_stream and
            not github_channel.categories.filter(category_id__to_stream=False).exists()
        )

        # Sélectionner la meilleure occurrence (fréquence) pour cette chaîne
        best_occurrence = get_best_occurrence(mumu_channel)

        # Ne pas diffuser s'il existe déjà une meilleure source pour cette chaîne
        if better_signal_exists_elsewhere(mumu_channel, github_channel):
            mumu_channel.to_stream = False
    else:
        # Pas de correspondance trouvée, ne pas diffuser
        mumu_channel.to_stream = False
```

(Code à titre représentatif seulement)

5.4. Gestion des Tâches et Monitoring :

Le TaskManager assure l'exécution fluide des opérations longues durée en arrière-plan. Il permet de suivre et gérer le cycle de vie complet des différentes tâches d'automatisation comme la synchronisation des chaînes, le scan satellite et le matching.

Il permet les choses suivantes :

- **Exécution asynchrone** : Lance les opérations dans des threads séparés pour ne pas bloquer l'interface utilisateur pendant l'exécution des tâches longues ;
- **Suivi d'état** : Maintient un registre de l'état de chaque tâche (en attente, en cours, terminée, échouée) avec horodatage et description ;
- **Monitoring de progression** : Permet aux fonctions de mettre à jour leur pourcentage d'avancement, visible en temps réel par l'utilisateur ;
- **Gestion robuste des erreurs** : Capture et journalise les exceptions, enregistre les traces complètes pour faciliter le débogage ;
- **Isolation des connexions** : Ferme et rouvre les connexions à la base de données entre les threads pour prévenir les fuites ;
- **Récupération flexible** : Offre diverses méthodes pour récupérer les tâches (par ID, nom, statut, récentes, actives, échouées).

Le système est conçu pour être utilisé avec le décorateur « with_task_monitoring », qui permet d'envelopper automatiquement une fonction dans le système de gestion des tâches. Ceci garantit que toutes les opérations majeures bénéficient d'une surveillance constante et

d'un feedback utilisateur, permettant ainsi à l'application de rester réactive même pendant les processus intensifs comme le scan des satellites ou le traitement par lots des chaînes.

Voici le modèle des tâches :

```python
class Task(models.Model):
    id = models.UUIDField(primary_key=True, default=uuid.uuid4, editable=False)
    name = models.CharField(max_length=255)
    description = models.TextField(null=True, blank=True)
    status = models.CharField(max_length=20, choices=[
        ('pending', 'Pending'),
        ('running', 'Running'),
        ('completed', 'Completed'),
        ('failed', 'Failed'),
    ], default='pending')
    progress = models.IntegerField(default=0)
    result = models.TextField(null=True, blank=True)
    error = models.TextField(null=True, blank=True)
    created_at = models.DateTimeField(auto_now_add=True)
    updated_at = models.DateTimeField(auto_now=True)

    def set_result(self, result):
        # Set task result, serializing to JSON if needed

    def get_result(self):
        # Get task result, deserializing from JSON if possible

    def set_status_running(self):
        # Set the status of the task to running

    def set_status_completed(self, result=None):
        # Set the status of the task to completed

    def set_status_failed(self, error):
        # Set the status of the task to failed

    def update_progress(self, progress):
        # Update the progress of the task (0-100%)

    class Meta:
        db_table = 'tasks'
        ordering = ['-created_at']
```

6. Difficultés et Solutions

- **Gestion Multi-threads avec Django** : Pour les opérations en arrière-plan, la gestion des threads inclut l'utilisation de close_old_connections() pour l'intégrité de la base de données Django, et un TaskManager exécute les fonctions en arrière-plan avec suivi ;

- **Fuzzy Matching pour les Chaînes TV** : Un algorithme de "fuzzy matching" (utilisant fuzz.ratio) est employé pour associer les chaînes MuMuDVB aux chaînes GitHub en comparant les noms avec un seuil de similarité ;

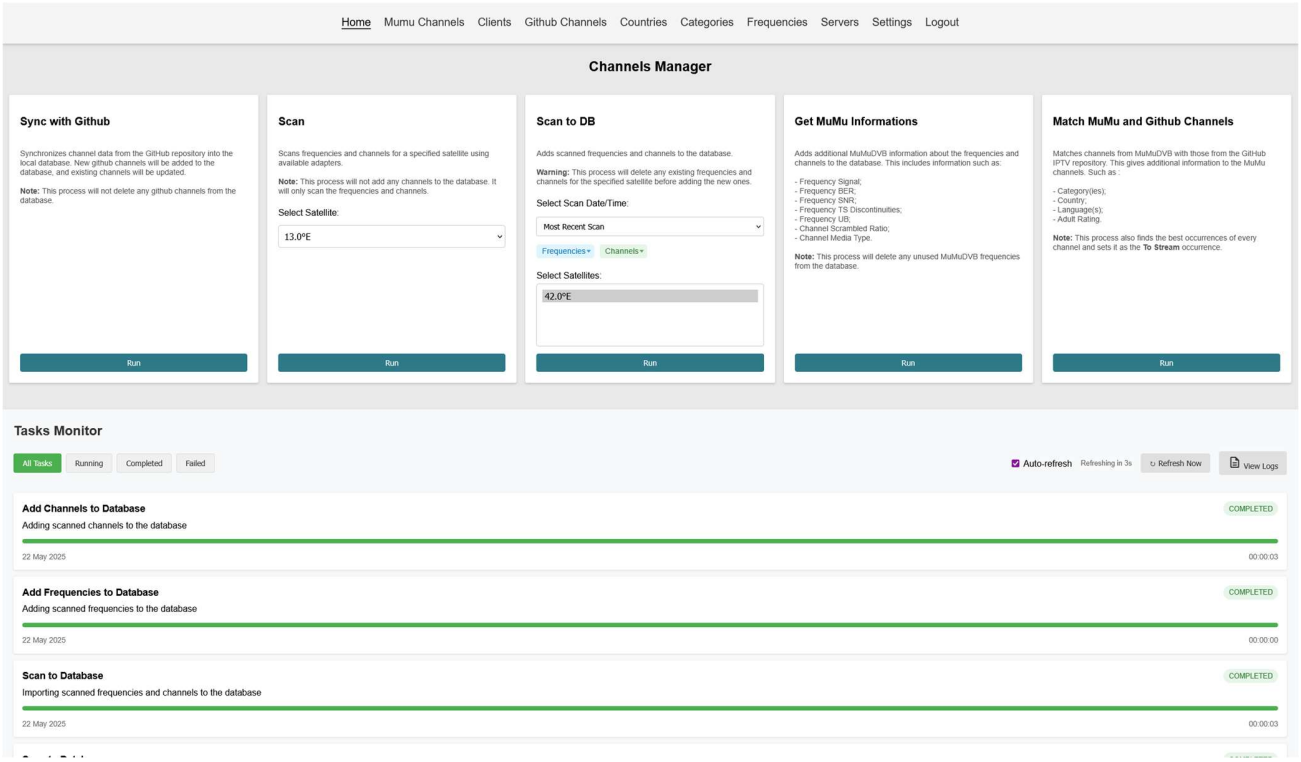
- **Interactions avec des Systèmes Externes** : Le projet gère les interactions avec l'API GitHub, les outils de scan satellite et MuMuDVB par une gestion robuste des erreurs, la validation des données et une journalisation détaillée ;

- **Performance avec de Grands Volumes de Données** : Les performances sont optimisées grâce au traitement par lots (batch processing) pour les opérations volumineuses (synchronisation, matching), à la pagination côté serveur dans l'API, et à l'optimisation des requêtes.

7. Interface Utilisateur (Console Web)

7.1 Page d’Accueil

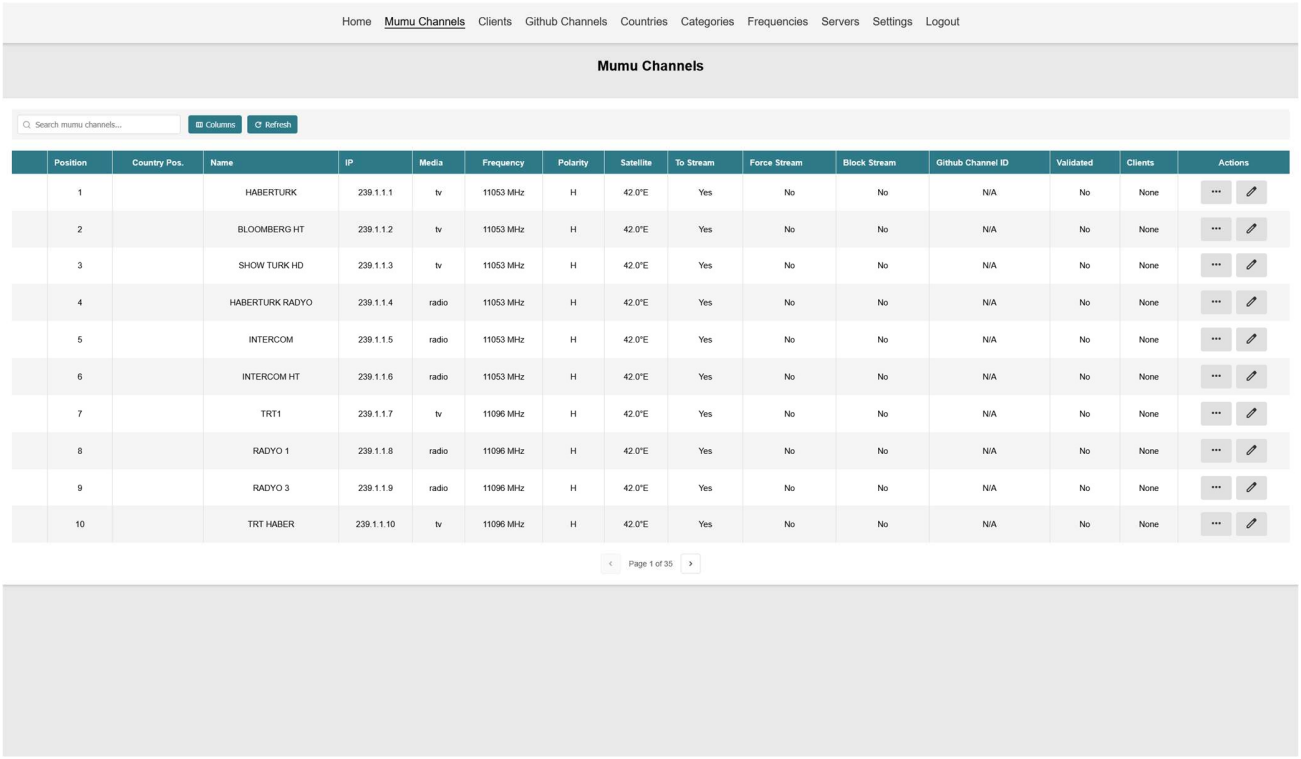
La page d'accueil présente cinq options principales pour gérer les chaînes TV : synchronisation GitHub, scan satellite, import en base de données, récupération d'informations MuMuDVB et matching des chaînes. Chaque option est clairement décrite et accompagnée de boutons d'action. La partie inférieure affiche un moniteur de tâches en temps réel permettant de suivre l'avancement des processus, filtrer par statut et consulter les détails via des modales interactives. Cette interface intuitive centralise toutes les opérations essentielles et fournit une visibilité complète sur les tâches en cours d'exécution.



7.2 Page de Gestion des Chaînes DVB

La page des chaînes MuMu offre une interface dédiée à la visualisation et la modification des chaînes satellite. Elle s'appuie sur la classe DataTable qui transforme les données brutes en un tableau interactif avec fonctionnalités de tri et filtrage. L'utilisateur peut éditer les chaînes via des formulaires modaux permettant de modifier l'adresse IP, le type de média, les options de diffusion et l'association avec les chaînes GitHub. La page permet également de gérer les occurrences (fréquences) des chaînes pour sélectionner la meilleure source de diffusion, facilitant ainsi l'optimisation de la qualité du service de streaming TV.

```
// Initialize the mumu channels table
mumuChannelsTable = new DataTable({
  containerId: "mumu-channels-container",
  tableId: "mumu-channels-table",
  columns: [
    { field: "position", label: "Position", sortable: true },
    { field: "country_position", label: "Country Pos.", sortable: false },
    { field: "name", label: "Name", sortable: true },
    { field: "ip", label: "IP", sortable: true },
    { field: "media", label: "Media", sortable: true },
    { field: "frequency", label: "Frequency", sortable: false, formatter: (f) => `${f / 1000} MHz` },
    { field: "polarity", label: "Polarity", sortable: false },
    { field: "satellite", label: "Satellite", sortable: false },
    { field: "to_stream", label: "To Stream", sortable: true },
    { field: "force_stream", label: "Force Stream", sortable: true },
  ],
  actions: {
    view: (channel) => viewMumuChannel(channel),
    edit: (channel) => editMumuChannel(channel),
  },
  apiEndpoint: "/api/mumu-channels",
  pageSize: 10,
  toolbox: {
    show: true,
    searchPlaceholder: "Search mumu channels...",
  },
  reordering: {
    enabled: true,
    orderFields: [{ field: "position", updateEndpoint: "/api/mumu-channels/reorder" }],
  },
  columnVisibility: {
    enabled: true,
    persistState: true,
  },
});
```



(Image utilisant des données factices)

8. Perspectives d'Évolution et Améliorations Futures

- **Modularité** : Faire évoluer l'architecture vers une plus grande modularité et une meilleure séparation des composants backend et frontend ;
- **Modernisation du Frontend** : Adopter un framework JavaScript moderne (comme React, Vue ou Svelte) pour dynamiser l'interface utilisateur et faciliter sa maintenance ;
- **Extension de l'API REST** : Améliorer l'API REST existante pour une gestion plus simple des ressources et l'exposition de nouvelles fonctionnalités ;
- **Mise en Cache des Données Fréquentes** : Implémenter des stratégies de mise en cache pour les données souvent consultées afin d'accélérer les temps de réponse ;
- **Système de File d'Attente pour Tâches Lourdes** : Intégrer un système de file d'attente dédié pour une gestion plus robuste et scalable des tâches asynchrones (scan, synchronisation) ;
- **Monitoring Opérationnel en Temps Réel** : Surveiller les chaînes diffusées et intégrer un système de « fallback » permettant de basculer sur une chaîne secondaire si le flux principal est perdu.

9. Conclusion

Le projet "Channels Manager" est une solution logicielle robuste et complète pour la gestion des chaînes de télévision. Il répond efficacement aux objectifs initiaux de gestion centralisée des configurations, d'automatisation des scans satellites, de synchronisation avec des référentiels IPTV externes comme celui présent dans GitHub, et de matching intelligent des chaînes. L'architecture basée sur Django, avec interface utilisateur web et des scripts en ligne de commande, offre de la flexibilité. Les défis techniques rencontrés ont été abordés par des

solutions intelligentes. Malgré sa complexité, je pense avoir réussi à créer une solution fiable et performante pour gérer les chaînes de SabSystem.