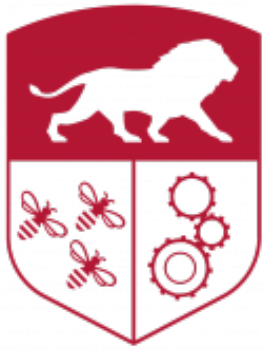

CENTRALE LYON

École Centrale Lyon

Les Applications Web
Rapport

Lecteur Multimédia : Groovy

Élèves :
Ilyas Ben alla
Ayoub Asseffar

Enseignants :
René Chalon
Romain Vuillemot

3 avril 2025

Table des matières

1 Introduction

Dans le cadre de notre projet, nous avons développé une application de lecteur multimédia afin de mettre en pratique nos acquis du cours. Ce projet s'inscrit dans une démarche d'apprentissage alliant théorie et pratique.

L'objectif principal était de concevoir une plateforme intuitive et fonctionnelle permettant de lire, organiser et gérer divers types de contenus multimédias. En intégrant des technologies modernes du web, nous avons cherché à optimiser l'expérience utilisateur tout en garantissant une bonne performance et une accessibilité sur différents supports.

Ce rapport détaille les différentes étapes de conception et de réalisation du projet, en mettant en avant les choix techniques effectués ainsi que les défis rencontrés lors du développement.

2 Choix techniques

Pour le développement de notre lecteur multimédia, nous avons opté pour un ensemble de technologies modernes du web, garantissant une application fluide, interactive et extensible. Les choix principaux incluent **HTML, CSS, JavaScript, Node.js**, ainsi que l'intégration d'**API externes** telles que YouTube et Spotify.

2.1 HTML, CSS et JavaScript : Structure et Dynamisme

HTML5 a été utilisé pour la structure du site, permettant d'intégrer le lecteur multimédia avec les balises `<audio>`, `<video>` et `<canvas>` pour l'affichage des images et l'édition basique. **CSS3** permet de styliser l'interface utilisateur, en offrant un design responsive et des modes clair/sombre. Des animations et transitions ont été également utilisées pour améliorer l'expérience utilisateur. Enfin, **JavaScript (ES6+)** gère l'interactivité du site, facilitant la lecture des fichiers multimédias, la gestion des playlists et de l'historique. Les fonctionnalités de manipulation du DOM et des événements assurent une navigation fluide.

2.2 Node.js et Backend

Pour le backend, nous avons choisi **Node.js**, permettant d'héberger l'application et de gérer les requêtes des utilisateurs. Le framework **Express.js** est utilisé pour gérer les routes et les API nécessaires à l'interaction avec les fichiers locaux et les services externes. En outre, le stockage des fichiers en **Base64** et dans le **LocalStorage** permet une gestion légère et rapide des médias sans nécessiter une base de données complexe.

2.3 Intégration d'API Externes

L'intégration des API externes est un atout majeur pour notre lecteur multimédia. L'**API YouTube IFrame** permet de rechercher et de lire des vidéos YouTube directement dans l'application. Elle facilite également le chargement de playlists et l'interaction avec les commandes de lecture. De plus, l'**API Spotify** permet d'accéder aux titres musicaux et aux playlists Spotify, avec une gestion de l'authentification via OAuth pour récupérer

les informations des utilisateurs, tout en tenant compte des restrictions entre comptes premium et gratuits.

3 Présentation de l'application

L'application offre une interface moderne et intuitive, permettant aux utilisateurs d'importer leurs médias, de créer des playlists personnalisées et d'accéder rapidement à leur historique de lecture. L'intégration des API externes telles que YouTube et Spotify permet de rechercher et de lire du contenu en ligne directement depuis la plateforme.

L'objectif principal était de concevoir un lecteur multimédia complet, combinant les fonctionnalités d'un lecteur local avec l'accès à des services de streaming, tout en assurant une navigation fluide et une compatibilité multiplateforme. Voici les principales caractéristiques de l'application :

3.1 Lecture de médias locaux

L'application prend en charge les fichiers audio, vidéo et images. Elle utilise le lecteur HTML5 intégré pour leur lecture et gère les erreurs ainsi que les formats non supportés.

3.2 Importation de fichiers

Les utilisateurs peuvent importer des fichiers via la méthode de glisser-déposer (drag drop) ou en sélectionnant des fichiers via l'explorateur de fichiers. Les fichiers sont ensuite convertis et stockés en base64 pour une gestion rapide.

3.3 Gestion de bibliothèque

La bibliothèque permet de stocker localement les médias importés, de les organiser par type (audio, vidéo, image) et d'afficher des statistiques d'utilisation (nombre de fichiers, taille totale). Il est aussi possible de supprimer les médias individuellement ou en totalité.

3.4 Playlists personnalisables

Les utilisateurs peuvent créer des playlists nommées, y ajouter ou supprimer des médias, et lire les playlists de manière séquentielle.

3.5 Historique de lecture

Un suivi des derniers médias joués est disponible, avec un accès rapide aux contenus récents et la possibilité de vider l'historique.

3.6 Intégration YouTube

L'application utilise l'**API YouTube Data v3** pour rechercher des vidéos et des musiques en ligne. Cette API permet de récupérer des informations détaillées sur les vidéos, les playlists, et d'effectuer des recherches personnalisées. Pour la lecture des vidéos, l'**API YouTube IFrame** est utilisée. Cette API permet d'intégrer un lecteur YouTube

directement dans l'application, offrant ainsi aux utilisateurs la possibilité de lire des vidéos directement depuis la plateforme. Un mode musique avec des contrôles simplifiés et un cache des résultats de recherche sont également inclus pour améliorer l'expérience utilisateur.

3.7 Intégration Spotify

L'**API Spotify** permet une connexion via OAuth, offrant deux modes de lecture : les utilisateurs premium peuvent profiter de la lecture complète des titres, tandis que les utilisateurs gratuits ont accès à des prévisualisations de 30 secondes. Les couvertures des albums sont également affichées.

3.8 Fonctionnalités avancées

L'application offre des fonctionnalités avancées telles que l'édition d'images (rotation, zoom, filtres), des raccourcis clavier (par exemple, Ctrl+Shift+R pour réinitialiser), ainsi que des modes clair/sombre en fonction des préférences utilisateur. La gestion des erreurs et des messages utilisateur est également optimisée pour une meilleure expérience.

3.9 Responsive design

L'application est entièrement responsive, adaptée aux différents types d'écrans. Des barres de progression interactives et des notifications contextuelles enrichissent l'expérience utilisateur.

3.10 Sécurité et performances

La gestion de la taille du cache et des quotas de stockage est optimisée pour garantir de bonnes performances. De plus, les requêtes API sont efficacement optimisées pour éviter toute surcharge du système.

Ainsi, cette application combine à la fois les fonctionnalités d'un lecteur multimédia local et l'accès à des plateformes externes telles que YouTube et Spotify, tout en offrant une interface unifiée et intuitive.

4 Analyse des points importants du code JS

4.1 script.js

4.1.1 Constructor()

Le constructeur de la classe `MediaPlayer` initialise tous les éléments et propriétés nécessaires pour l'application de lecture multimédia. Il configure les références aux éléments du DOM, initialise la bibliothèque multimédia à partir du `localStorage`, paramètre les clés API de YouTube et Spotify, et met en place diverses variables d'état. Le constructeur appelle également `initializeBasicFeatures()` pour configurer les fonctionnalités de base.

Principales fonctionnalités : Il comprend les références à tous les éléments du DOM importants (entrées de fichiers, lecteurs, listes de bibliothèque), l'initialisation de la bibliothèque multimédia, la configuration de l'intégration des API YouTube et Spotify, ainsi que la gestion de l'historique des médias et des mécanismes de mise en cache.

4.1.2 initializeBasicFeatures()

Cette méthode est le point d'initialisation principal des fonctionnalités du lecteur. Elle met en place les écouteurs d'événements, rend les premières vues de la bibliothèque et des playlists, gère le basculement entre les modes (local, YouTube, Spotify), charge le contenu de la bibliothèque côté serveur et configure les contrôles de lecture.

Principales fonctionnalités : L'initialisation des écouteurs d'événements, le rendu initial de la bibliothèque multimédia et des playlists, la gestion du changement de mode, le chargement des médias distants et l'initialisation des contrôles de lecture.

4.1.3 initializeEventListeners()

Cette méthode configure tous les écouteurs d'événements nécessaires pour permettre à l'application de répondre aux interactions des utilisateurs. Cela inclut la gestion des téléchargements de fichiers, les raccourcis clavier et les interactions avec l'interface utilisateur.

Principales fonctionnalités : La gestion des clics sur les entrées de fichiers, la prise en charge du glisser-déposer pour les fichiers, l'ajout de raccourcis clavier (par exemple, `Ctrl+Shift+R` pour réinitialiser la bibliothèque) et un retour réactif pendant les opérations de fichiers.

4.1.4 handleFiles()

Cette méthode traite une liste de fichiers sélectionnés ou déposés dans l'application. Elle valide les types de fichiers et transmet les fichiers multimédias valides au processus d'ajout à la bibliothèque.

Principales fonctionnalités : Validation des types de fichiers (audio, vidéo, images), traitement par lots des fichiers, gestion des erreurs pour les formats non pris en charge et affichage des messages de progression.

4.1.5 addToLibrary()

Ajoute un fichier multimédia valide à la bibliothèque de l'application. Il est converti en URL de données Base64 pour le stockage et un objet média avec des métadonnées est créé.

Principales fonctionnalités : Validation de la taille et du type du fichier, conversion en Base64, création des métadonnées (ID, nom, type, horodatage), mise à jour de l'interface utilisateur et gestion des erreurs avec retour utilisateur.

4.1.6 fileToBase64()

Convertit un objet fichier en une URL de données Base64 pour le stockage dans la bibliothèque. La méthode inclut un suivi de la progression pour les fichiers volumineux.

Principales fonctionnalités : Utilisation de l'API `FileReader`, suivi de progression pour les fichiers volumineux, validation des données, gestion asynchrone basée sur les promesses et gestion complète des erreurs.

4.1.7 playMedia()

Cette fonction principale gère la lecture de différents types de médias (audio, vidéo, images) en fonction du type du fichier. Elle met à jour l'historique de lecture et l'état de l'interface utilisateur.

Principales fonctionnalités : Détection et routage du type de média, suivi de l'historique de lecture, mise à jour de l'état de l'interface utilisateur et intégration avec le suivi de playlist.

4.1.8 saveLibrary()

Sauvegarde la bibliothèque multimédia actuelle dans `localStorage` avec une gestion complète des erreurs, notamment en cas de dépassement du quota de stockage.

Principales fonctionnalités : Persistance avec `localStorage`, estimation de la taille et avertissements utilisateur, stratégies de repli pour les fichiers volumineux et journalisation détaillée.

4.1.9 loadLibrary()

Charge la bibliothèque multimédia à partir du `localStorage` en gérant les erreurs liées aux données corrompues ou manquantes.

Principales fonctionnalités : Récupération des données, validation, récupération après erreur, support de migration pour les futures versions et reporting des statistiques.

4.1.10 renderLibrary()

Génère et affiche l'interface utilisateur de la bibliothèque multimédia, y compris les statistiques et les éléments individuels avec leurs métadonnées et actions associées.

Principales fonctionnalités : Génération dynamique de l'HTML, affichage des statistiques, organisation des éléments, interactivité et retour visuel sur les éléments en cours de lecture.

4.1.11 renderPlaylist()

Affiche les éléments multimédias récemment lus dans la section des playlists de l'interface utilisateur.

Principales fonctionnalités : Suivi des lectures récentes, tri temporel, icônes spécifiques aux types de fichiers et gestion interactive de la lecture.

4.1.12 initializePlaylists()

Configure la gestion des playlists, y compris la création, la gestion et la lecture de playlists.

Principales fonctionnalités : Stockage et récupération des playlists, gestion de l'interface utilisateur, ajout de médias aux playlists et intégration avec la bibliothèque principale.

4.1.13 playPlaylist()

Gère la lecture d'une playlist avec mise à jour de l'interface utilisateur et lecture séquentielle des médias.

Principales fonctionnalités : Séquencement des éléments, gestion de l'état de lecture, synchronisation de l'interface utilisateur et récupération en cas d'erreur.

4.1.14 initializeModeSwitch()

Gère le basculement entre les différents modes d'application (fichiers locaux, YouTube, Spotify).

Principales fonctionnalités : Gestion de l'état de l'interface, activation des fonctionnalités spécifiques au mode, configuration des types de recherche et persistance des états.

4.1.15 performYouTubeSearch()

Effectue une recherche de contenu sur YouTube en utilisant l'API YouTube avec des mécanismes de secours en cas de dépassement des limites API.

Principales fonctionnalités : Intégration de l'API YouTube, mise en cache des résultats, récupération après échec et affichage réactif des résultats.

4.1.16 initializeSpotify()

Configure l'intégration avec Spotify, y compris l'authentification et la détection du mode de lecture (Premium vs gratuit).

Principales fonctionnalités : Flux d'authentification, détection du type de compte, sélection du mode de lecture et gestion des erreurs.

4.2 server.js

Le fichier `server.js` est le cœur de l'application Node.js, servant de serveur pour gérer l'authentification avec Spotify et la gestion des fichiers multimédia. Il utilise des fonctionnalités asynchrones et des middlewares pour assurer un fonctionnement fluide. Ci-dessous, nous expliquons les principales fonctions et leur rôle dans ce fichier.

4.2.1 Gestion des Requêtes Asynchrones

Le code utilise la fonctionnalité asynchrone pour effectuer des appels HTTP et gérer les réponses de manière non bloquante.

Fonction `fetch` La fonction `fetch` est utilisée dans plusieurs endroits du code pour faire des requêtes HTTP vers l'API Spotify et récupérer des informations, telles que le token d'accès ou la liste des fichiers multimédia. Cette fonction retourne une promesse et nécessite un traitement avec `.then()` ou `await` dans une fonction asynchrone.

- `fetch('https://accounts.spotify.com/api/token', ...)` : Cette fonction est utilisée pour envoyer une requête POST à l'API de Spotify pour obtenir un token d'accès. Elle envoie les paramètres nécessaires, tels que le `code` d'autorisation ou le `refresh_token`, et récupère le token d'accès en réponse.
- `await fetch(...)` : Le mot-clé `await` est utilisé pour attendre que la promesse retournée par `fetch` soit résolue avant de poursuivre l'exécution du code. Cela permet d'éviter les blocages et garantit que la réponse est récupérée avant d'agir sur elle.
- `.then(response => response.json())` : Après avoir reçu une réponse de `fetch`, cette méthode est utilisée pour convertir la réponse en format JSON, ce qui permet d'accéder aux données renvoyées.

Utilisation d'Async/Await L'utilisation de la syntaxe `async/await` permet de simplifier la gestion des appels asynchrones dans le code. Les fonctions marquées avec `async` retournent une promesse, et l'utilisation de `await` permet de résoudre ces promesses de manière plus lisible, en évitant les chaînes de `.then()`.

- La fonction `callback` utilise `async/await` pour attendre la réponse de la requête `fetch` avant de poursuivre l'exécution du code.
- Par exemple, dans `await fetch('https://accounts.spotify.com/api/token', ...)`, l'exécution est suspendue jusqu'à ce que la réponse de l'API Spotify soit reçue et que le token d'accès soit extrait.

4.2.2 Gestion des Fichiers Statics

Les fichiers multimédia (audio, vidéo, images) sont gérés par le serveur Node.js pour être accessibles à l'utilisateur via les routes définies.

Middleware pour les Fichiers Statics La fonction suivante est utilisée pour configurer le serveur afin qu'il serve des fichiers statiques (audio, vidéo, images) à l'utilisateur.

- `app.use(express.static(path.join(__dirname, 'public')))` : Cette fonction permet de servir les fichiers statiques à partir du répertoire `public`. Tous les fichiers

qui y sont présents, tels que les fichiers audio ou vidéo, peuvent être accessibles via les routes définies dans le serveur.
— La fonction `path.join(__dirname, 'public')` construit le chemin absolu vers le répertoire `public`, ce qui permet de servir correctement les fichiers, quel que soit l'environnement d'exécution du code.

Route pour la Gestion des Fichiers Les routes suivantes sont utilisées pour gérer l'accès aux fichiers multimédia.
— `app.get('/api/media/:type', (req, res) => ...)` : Cette route permet de récupérer une liste de fichiers d'un type spécifique (audio, vidéo, image). Le type est passé en tant que paramètre dans l'URL, et la fonction renvoie une liste de fichiers correspondants dans le répertoire approprié.
— `fs.readdir(mediaDir, (err, files) => ...)` : Cette fonction de `fs` lit le contenu du répertoire `mediaDir` (qui correspond à l'un des répertoires pour les fichiers audio, vidéo ou image). Elle renvoie une liste de fichiers que l'on peut traiter pour les afficher ou les manipuler.
— `path.extname(file)` : Cette méthode permet de vérifier l'extension du fichier et de s'assurer qu'il correspond au type attendu (audio, vidéo ou image). Elle permet de filtrer les fichiers qui ne correspondent pas au type.

4.2.3 Gestion de l'Authentification et Sécurisation des Routes

L'une des principales fonctionnalités du code est l'authentification des utilisateurs avec Spotify. Cela garantit que seuls les utilisateurs authentifiés peuvent accéder aux ressources protégées.

Sécurisation des Routes Certaines routes du serveur nécessitent que l'utilisateur soit authentifié avant de pouvoir y accéder. Ces routes sont sécurisées en vérifiant que le token d'accès est valide.
— `app.get('/callback', async (req, res) => ...)` : Lorsque l'utilisateur revient de l'authentification Spotify, cette route récupère le code d'autorisation, échange ce code contre un token d'accès, et vérifie sa validité avant de permettre à l'utilisateur d'accéder aux fonctionnalités protégées.
— `app.get('/refresh`$_{token'}$`, async(req, res) => ...)` :
Cette route permet de rafraîchir un token expiré. L'utilisateur doit fournir un `refresh_token` valide, et la fonction s'assure que celui-ci est bien traité pour récupérer un nouveau token d'accès.

Stockage Sécurisé des Tokens Les tokens d'accès et de rafraîchissement doivent être stockés de manière sécurisée. Ils sont généralement enregistrés dans une base de données ou un stockage sécurisé pour être utilisés ultérieurement, bien que dans ce code, ils sont envoyés directement au client via des fenêtres JavaScript.
— La réponse de l'API Spotify contient des informations sensibles, telles que le token d'accès et le token de rafraîchissement. Ces informations doivent être traitées avec soin pour éviter toute fuite de données.

4.2.4 Middleware de Gestion des Erreurs

Le serveur utilise également un middleware pour gérer les erreurs éventuelles qui pourraient survenir durant l'exécution de l'application.

Middleware de Gestion des Erreurs Le middleware d'erreur permet de capturer toutes les erreurs non gérées par le reste du code et d'envoyer une réponse d'erreur appropriée à l'utilisateur.

- `app.use((err, req, res, next) => ...)` : Ce middleware capture toutes les erreurs générées par le serveur et renvoie une réponse d'erreur HTTP 500 (erreur interne du serveur) à l'utilisateur.
- La fonction `next` permet de transmettre l'erreur à un autre middleware si nécessaire, bien que dans ce cas, elle soit utilisée pour traiter uniquement les erreurs non gérées.

5 L'application en images

5.1 Page d'accueil

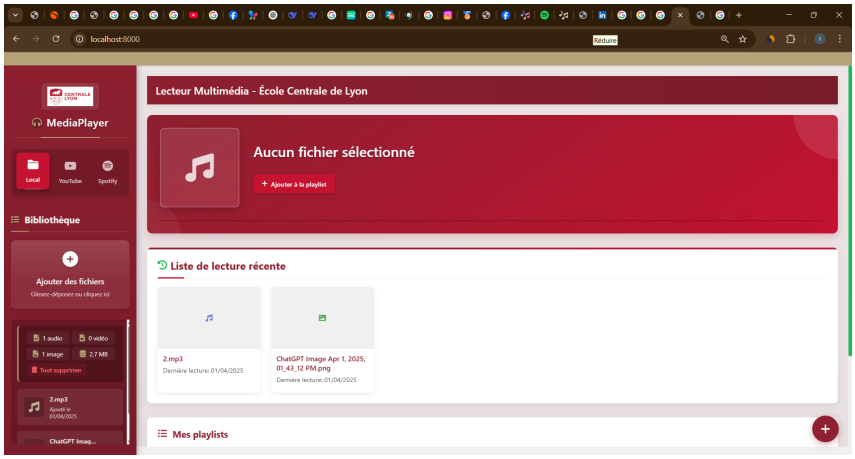

FIGURE 1 – Première page

5.2 Barre du menu

FIGURE 2 – Barre du menu

5.3 Lecteur audio

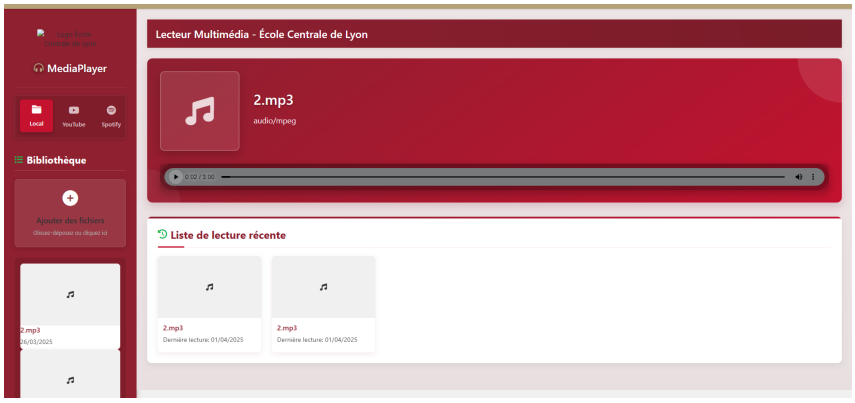

FIGURE 3 – Lecteur Audio

5.4 Création de playlistes

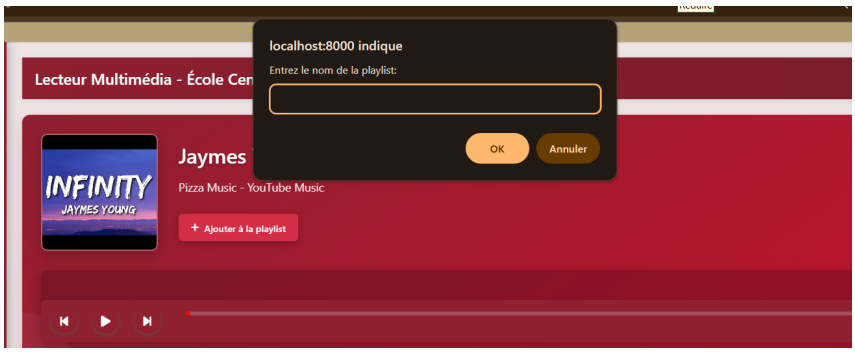

FIGURE 4 – Création de playlistes

5.5 Lecteur video

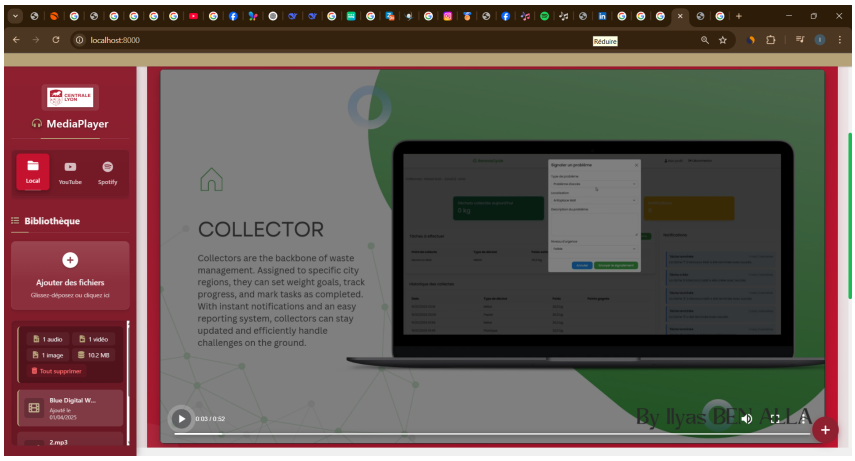

FIGURE 5 – Lecteur Video

5.6 Lecteur images

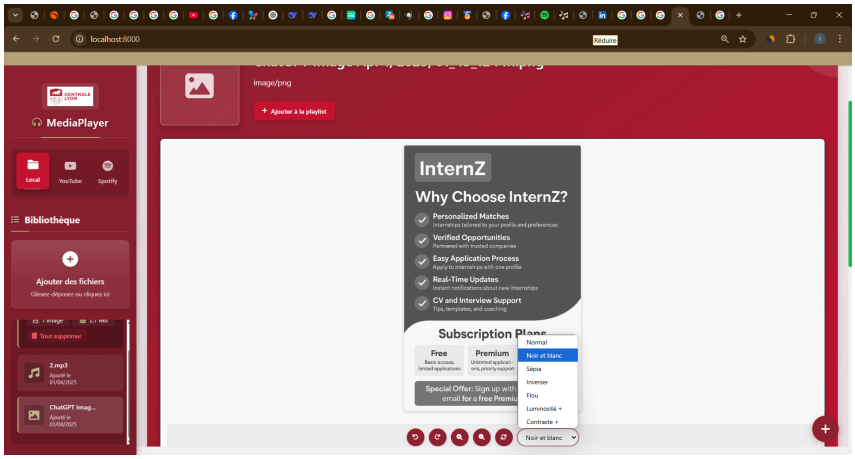

FIGURE 6 – Lecteur Images

5.7 Page Youtube

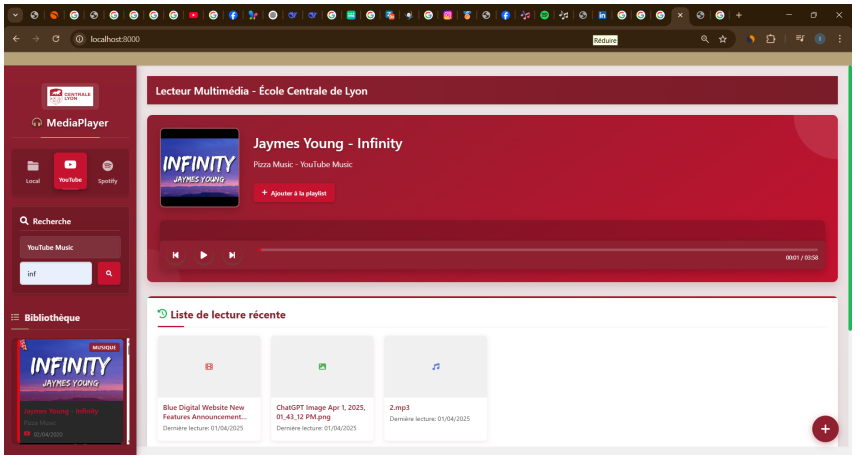

FIGURE 7 – Page YouTube

5.8 Page Spotify

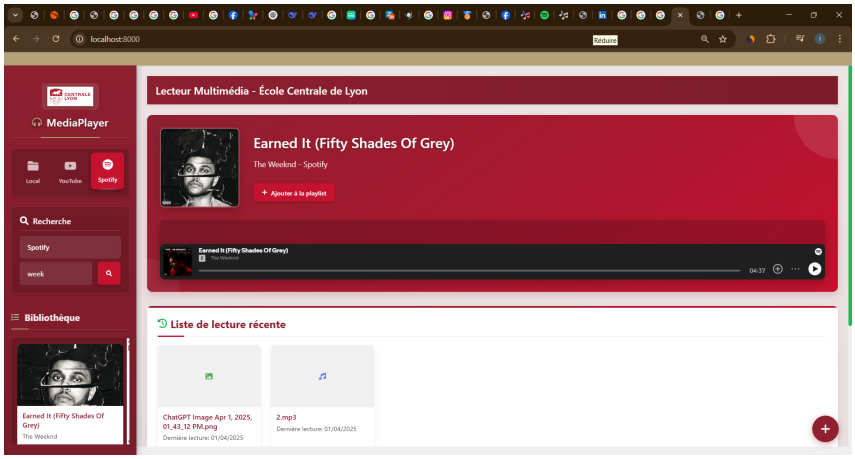

FIGURE 8 – Page Spotify

6 Conclusion & Perspectives

6.1 Conclusion

Ce projet a permis de développer une application multimédia complète et interactive, intégrant à la fois un lecteur de fichiers locaux et des fonctionnalités avancées telles que l'accès à YouTube et Spotify. Grâce à l'utilisation des technologies modernes comme HTML5, CSS3, JavaScript et Node.js, l'application offre une interface fluide et adaptable à différents types de médias et appareils. Les fonctionnalités telles que l'importation de fichiers, la gestion des playlists, et l'édition d'images viennent enrichir l'expérience utilisateur et permettent de gérer efficacement les contenus multimédias.

Le design responsive et l'intégration des API externes contribuent à rendre l'application pertinente et dynamique, répondant ainsi aux besoins des utilisateurs recherchant une solution centralisée pour leurs médias.

6.2 Perspectives

Malgré les fonctionnalités riches déjà implémentées, plusieurs pistes d'amélioration et d'extension sont envisageables pour rendre l'application encore plus complète et performante :

- **Amélioration de la gestion des formats multimédias** : Actuellement, le support des formats multimédias est limité. L'ajout de la prise en charge d'autres formats (par exemple, des fichiers audio FLAC ou des vidéos en 4K) permettrait d'étendre la compatibilité de l'application.
- **Partage de médias** : Ajouter la possibilité de partager des médias entre utilisateurs, via des liens ou des réseaux sociaux, renforcerait l'aspect collaboratif de l'application.
- **Amélioration de la sécurité** : Intégrer une gestion des utilisateurs avec authentification et autorisation (par exemple, via un système de comptes et de sessions sécurisées) garantirait la protection des données personnelles.
- **Fonctionnalités de recommandation** : En intégrant des algorithmes de recommandation basés sur l'historique de lecture et les préférences des utilisateurs, l'application pourrait suggérer des contenus audio, vidéo ou image adaptés à chaque utilisateur.
- **Optimisation des performances** : Une meilleure gestion du cache et des requêtes API permettrait d'améliorer encore la réactivité de l'application, notamment pour les utilisateurs avec des connexions internet lentes.

Ces améliorations permettraient de répondre aux besoins des utilisateurs de manière encore plus fluide et d'étendre les fonctionnalités offertes par l'application pour en faire un outil encore plus puissant et polyvalent.