Rapport Technique

Création d'une application bureau de reconnaissance musicale

SAE FISA1 - ISIMA

Groupe 8 :
Laurian JAMIN, Dorian FAURE, Corentin RICHARD, Yann MERLIN

Table des matières

Table des figures....3
I. Présentation....4
i. Création d'une application de reconnaissance musicale....4
a. Contexte et objectif....4
b. Choix des technologies et architecture générale....5
ii. Organisation et outils....7
a. Gestion de projet....7
b. Développement....8
iii. Modélisation....9
a. Diagramme de cas d'utilisation....9
b. Diagramme de séquence du cas « Analyser un audio »....10
c. Diagramme de classes serveur....11
II. Travail effectué....13
i. Front-end....13
a. Le framework Angular....13
b. Electron JS....16
ii. Back-end....17
a. Le principe d'API....17
b. L'algorithme de reconnaissance audio....18
c. L'authentification....22
iii. Base de données....23
a. Technologies utilisées....23
b. Création et modification automatique....23
iv. Serveur....24
III. Bilan....25
i. Bilan technique....25
ii. Bilan scolaire et humain....26
iii. Difficultés rencontrées....27
iv. Conclusion....28
Annexes....29
i. Démonstration de l'application....29
ii. Glossaire....33
iii. Ressources utilisées....35

Table des figures

1 - Schéma des objectifs....6
2 - Diagramme de Gantt rétrospectif du projet....7
3 - Tableau Kanban GitLab....8
4 - Diagramme de cas d'utilisation....9
5 - Diagramme de séquence....10
6 - Diagramme de classes serveur....11
7 - Exemple de composant Angular - TypeScript....13
8 - Historique des musiques....14
9 - Bibliothèque des favoris....14
10 - Exemple de service Angular....15
11 - Extrait du app.js....16
12 - Schéma d'architecture d'une API....17
13 - Extrait de code du backend Services/Utils/AudioConverter.cs/ExtractPeaksWithTime....18
14 - Extrait de code du backend Services/Utils/AudioConverter.cs/GenerateFingerprints....19
15 - Extrait de code du backend Services/AudioServices.cs/SearchAudioWithFingerprintsAsyn21
16 - Stockage d'un user en BD....22
17 - Les bibliothèques utilisées pour la BD....23

I. Présentation

i. Création d'une application de reconnaissance musicale.

a. Contexte et objectif

Dans le cadre de notre première année de diplôme d'ingénieur en apprentissage à l'ISIMA, nous avons eu l'opportunité de réaliser un projet informatique complet, de la conception et modélisation* du logiciel, jusqu'à la réalisation et livraison de notre solution. Le sujet est la réalisation d'un algorithme* de reconnaissance musicale, type « Shazam », en utilisant des techniques de transformation numérique du signal, comme la transformée de Fourier. Concernant la forme de l'application, le choix est libre. Notre groupe a choisi de réaliser une application de bureau* pour Windows*, intitulée Larrythmique.

C'est dans ce contexte que nous avons débuté la réalisation de ce projet. Une première phase de conception et de modélisation nous a permis de déterminer quelles seront les fonctionnalités principales de notre application, ainsi que son architecture générale.

Les fonctionnalités ainsi identifiées constituent nos objectifs pour ce projet :

1 – L'utilisateur doit pouvoir entrer un fichier audio et recevoir une réponse identifiant le titre de la musique présente dans son audio, s'il y en a une.

2 – L'utilisateur a à sa disposition 3 moyens différents d'entrer un audio dans l'application :

A – Via le micro de son ordinateur, qui capture le son de son environnement. Cela permet de reconnaître par exemple une musique qui est jouée dans un café ou autre.

B – Via un fichier au format .mp3 ou .wav dont il dispose sur le disque dur de son ordinateur. Utile pour reconnaître une musique précédemment téléchargée dont le titre a été perdu par exemple.

C – Via le son du système. L'application récupère la bande audio en cours sur l'ordinateur de l'utilisateur. Utile pour identifier une musique dans une vidéo YouTube ou un film en cours.

3 – L'utilisateur peut utiliser l'application de manière anonyme, mais il peut également créer un compte et s'authentifier pour disposer de fonctionnalités supplémentaires :

A – Garder un historique des titres reconnus.

B – Disposer d'une bibliothèque de titres favoris pour les retrouver plus facilement.

Les termes utilisés pour la première fois suivis d'un astérisque rouge (*) sont définis dans le glossaire.

b. Choix des technologies et architecture générale

Le choix des technologies étant libre, nous avons comparé différentes options, mais nous avons principalement opté pour des solutions avec lesquelles nous étions familiers. Nous avons choisi des technologies que nous avons déjà l'habitude d'utiliser pour être le plus efficace possible durant la réalisation de ce projet.

Côté front-end*, nous avons voulu une solution utilisant HTML*, CSS*, et TypeScript*. Les deux frameworks* web principaux qui s'offraient alors à nous étaient Angular* ou React*. Nous avons opté pour Angular car Yann et Corentin l'utilisent déjà en entreprise.

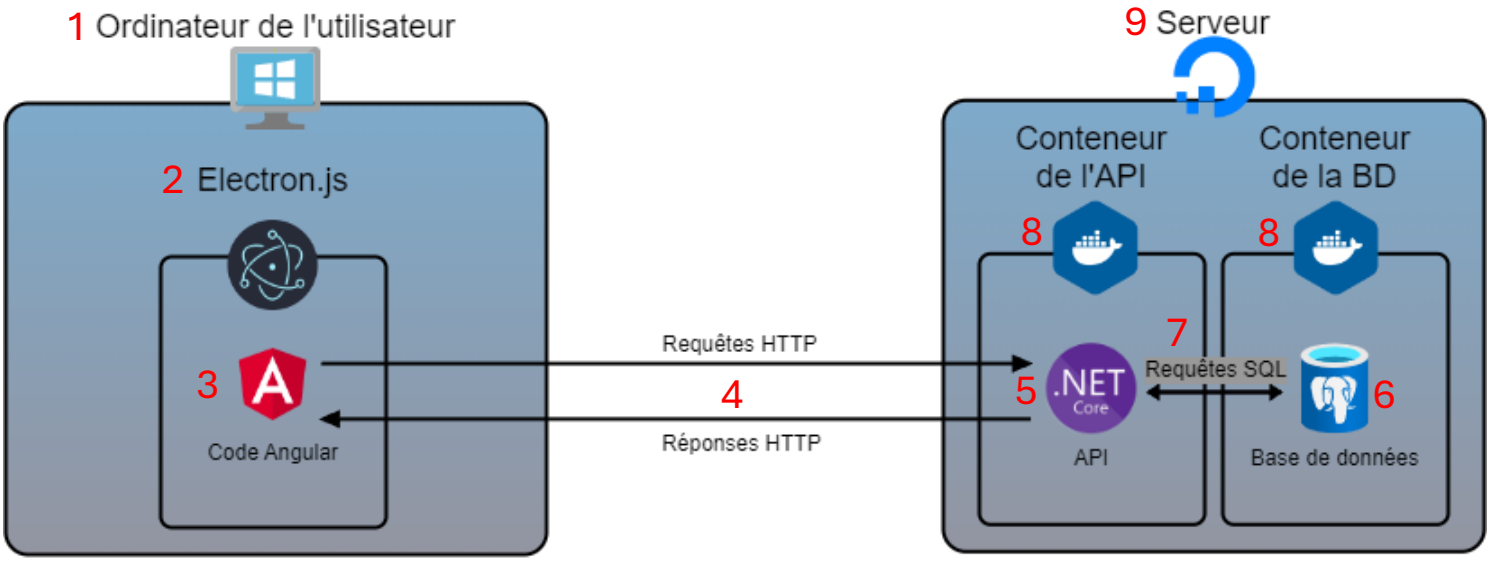
Comme nous voulons une application de bureau, nous avons besoin d'un framework pour « emballer » l'application web Angular comme une application native, exécutable sur Windows. Les deux solutions les plus populaires étant Tauri* ou Electron.js*, nous avons opté pour Electron.js. C'est un choix fait pour sa robustesse et sa maturité, en effet des applications comme Discord ou Microsoft Teams sont basées sur Electron.

Pour le back-end*, nous avons hésité entre le Java* avec le framework Spring Boot*, ou le C#* avec le framework ASP.NET Core*. Nous avons opté pour ASP.NET puisque Corentin et Dorian l'utilisent régulièrement.

Notre système de gestion de base de données sera PostgreSQL*. Ce choix était évident de par sa facilité d'utilisation, comparé à des solutions comme Oracle*, son statut open source*, sa grande compatibilité, et enfin le fait que Yann et Corentin aient déjà l'habitude de l'utiliser.

Enfin, notre back-end et notre base de données* seront conteneurisé* avec Docker*, puis déployés sur un serveur*. Ce serveur est hébergé chez DigitalOcean* car c'est un serveur que Dorian louait déjà, que nous réutilisons pour ce projet. L'hébergeur DigitalOcean avait été choisi pour ses prix abordables et ses bons avis.

Avec ces choix technologiques, et grâce aux objectifs énoncés dans la partie précédente, nous avons imaginé l'architecture générale suivante, qui constitue notre schéma des objectifs :



1 - Schéma des objectifs

Sur l'ordinateur de l'utilisateur [1] est installée l'application Larrythmique. Cette application utilise le framework Electron.js [2] pour « emballer » le code de l'application web réalisée en Angular [3] et ainsi la rendre native, exécutable sur Windows. Cette dernière communique via des requêtes HTTP* [4] avec l'API* .NET [5] qui contient l'algorithme principal de reconnaissance musicale. Les requêtes HTTP [4] contiennent le fichier audio que l'algorithme devra analyser. L'API [5] communique avec la base de données [6] via des requêtes SQL* [7] pour enregistrer les empreintes audio* et effectuer des recherches parmi elles. L'API [5] et la BD [6] sont conteneurisées avec Docker [8] pour être déployés sur le serveur [9] hébergé par DigitalOcean.

ii. Organisation et outils

a. Gestion de projet

Pour mener à bien ce projet, nous avons suivi une organisation simple.

Premièrement, notre groupe est composé de 4 étudiants, et chacun a un rôle défini :

Corentin Richard : Responsable de la gestion de projet et de la conception de la base de données

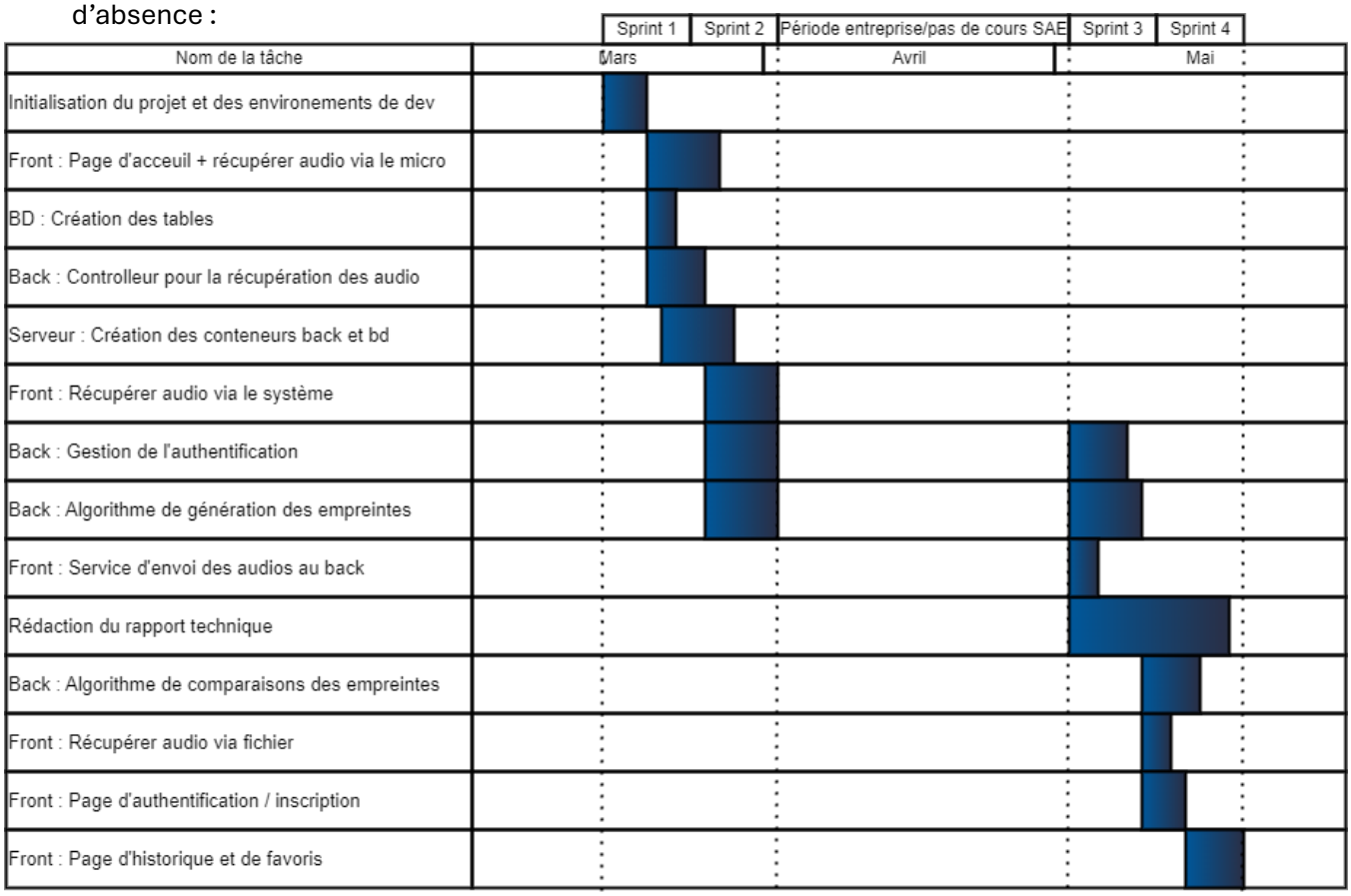
Laurian Jamin : Développeur front-end et réalise les logos, les maquettes, et le thème visuel.

Dorian Faure : Développeur back-end et responsable de l'hébergement du serveur.

Merlin Yann : Développeur front-end et back-end et responsable du rapport technique.

Ensuite, notre logiciel est développé en mode agile*, une approche de gestion de projet fondée sur des itérations de travail courtes appelées sprints. Chaque sprint dure environ 10 jours dans notre cas, et permet de réaliser un ensemble de tâches de développement définies. L'objectif de cette méthode est de proposer une grande adaptabilité face aux imprévus : si par exemple, une tâche a été oubliée ou si les objectifs évoluent, la liste de tâches du sprint suivant peut être ajustée en conséquence. C'est l'opposé de la méthode en cascade, où toutes les tâches sont définies en amont du projet, et ne changent plus.

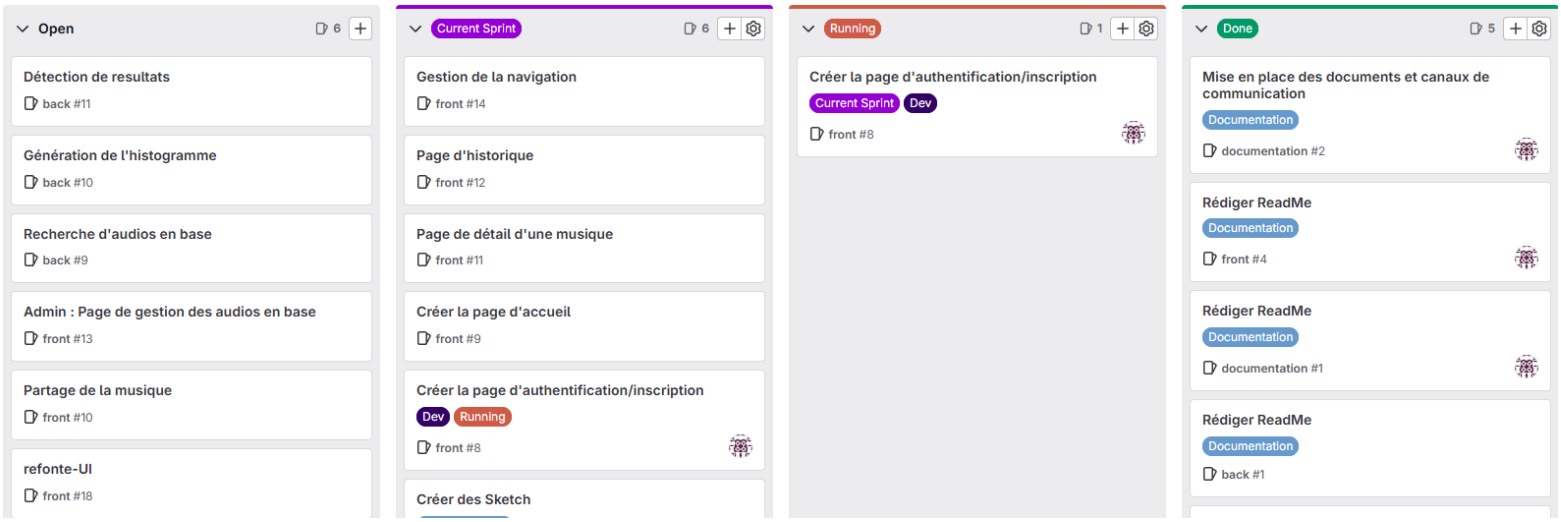
Après avoir découpé le développement de Larrythmique en une longue liste de tâches, nous les avons assignées sur quatre sprints, soit quatre itérations de travail, coupées par une période d'absence :



2 - Diagramme de Gantt rétrospectif du projet

Le diagramme de Gantt ci-dessus décrit la répartition des tâches identifiées sur nos 4 sprints. On peut constater que notre organisation n'était pas la meilleure puisque bien souvent des tâches n'étaient pas finalisées et ont dépassé d'un sprint à l'autre.

Pour gérer les tâches nous avons utilisé les outils de gestion de projets fournis par GitLab*, en particulier le tableau Kanban* :



3 - Tableau Kanban GitLab

Grâce à des tags et aux différentes colonnes du tableau, nous avons efficacement pu gérer les différentes tâches et leur statut.

b. Développement

Les outils de développement que nous avons utilisés sont les suivants :

- L'IDE* Visual Studio Code pour la partie front-end
- L'IDE Visual Studio pour la partie back-end
- Le logiciel DBeaver pour la gestion de la base de données
- L'interface WEB de DigitalOcean pour la gestion du serveur

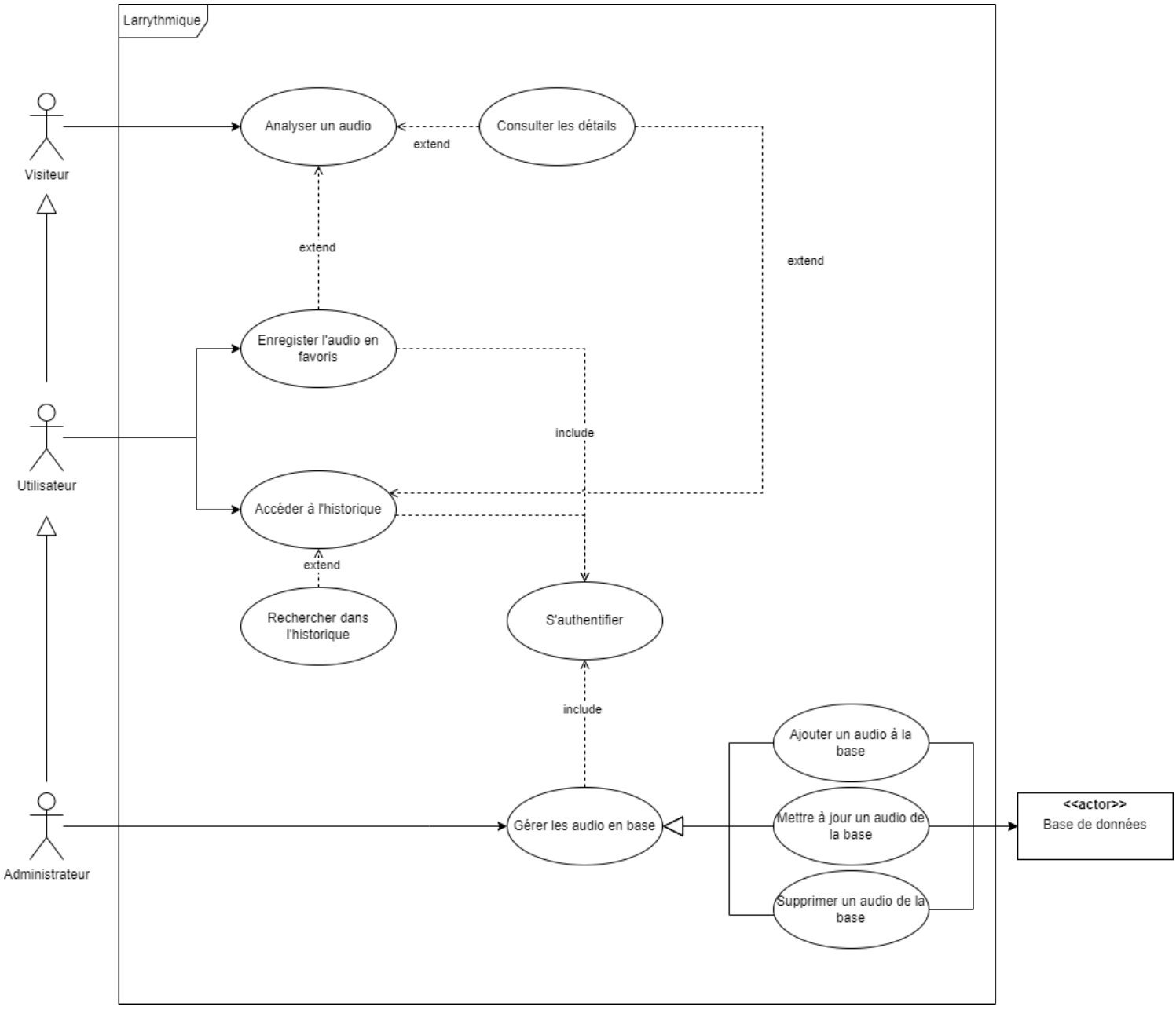
Dans notre code, nous avons également utilisé les bibliothèques externes suivantes :

- Angular Material pour les éléments graphiques
- Accord.Math pour les fonctions mathématiques côté backend, notamment la transformée de Fourier
- NAudio pour la gestion des fichiers audio côté back-end

iii. Modélisation

a. Diagramme de cas d'utilisation

Le diagramme de cas d'utilisation n'a pas changé par rapport à celui présenté précédemment dans le dossier de modélisation :

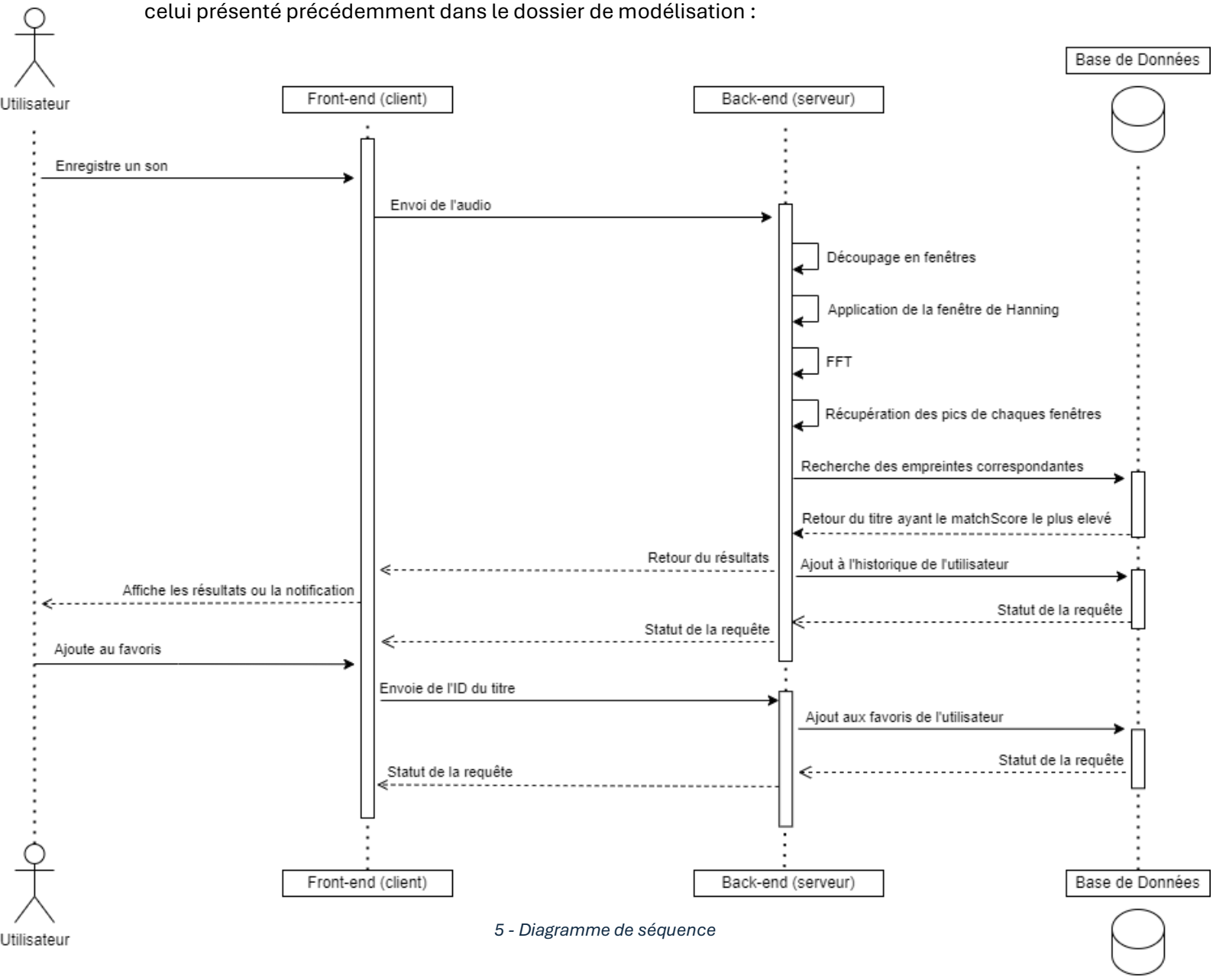


4 - Diagramme de cas d'utilisation

Le diagramme met en évidence trois acteurs principaux, à savoir l'utilisateur et l'administrateur, ce dernier étant un utilisateur avec des privilèges supplémentaires. L'utilisateur doit obligatoirement s'authentifier pour accéder à des fonctionnalités avancées, comme l'historique des analyses, tandis que l'administrateur peut gérer la base d'audio directement depuis l'application en ajoutant, modifiant ou supprimant des données.

b. Diagramme de séquence du cas « Analyser un audio »

Le diagramme de séquence du cas « Analyser un audio » a subi des modifications par rapport à celui présenté précédemment dans le dossier de modélisation :

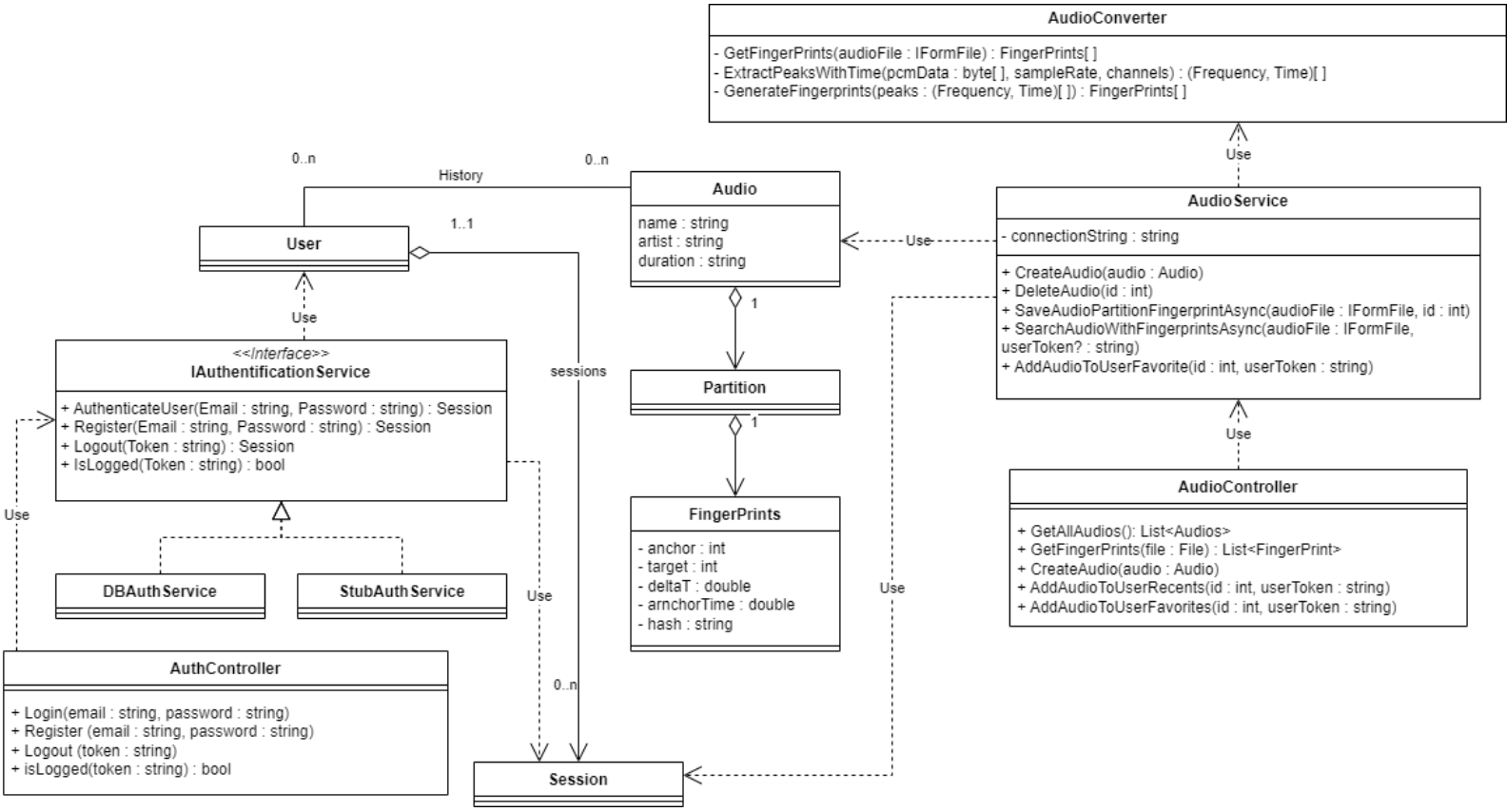


5 - Diagramme de séquence

Dans ce scénario d'analyse d'un audio pour un utilisateur authentifié, l'intégralité de l'algorithme de reconnaissance d'audio se déroule dans le back-end. Dans notre ancien diagramme de séquence pour le premier dossier de modélisation, nous avions prévu d'effectuer le calcul des spectrogrammes* sur le front-end, afin de ne pas avoir à envoyer l'audio complet au serveur, pour que les requêtes soient plus légères. Finalement, nous avons pris la décision pour la version finale de l'application de déporter toute la partie calcul sur le back-end. En effet, l'envoi de court fichier audio en binaire n'est pas si lourd et il est préférable pour une question d'organisation et d'efficacité de séparer les responsabilités. C'est-à-dire que le front n'est responsable que de l'affichage, et le back n'est responsable que des calculs et des appels à la base de données. Une fois la musique identifiée, l'utilisateur peut l'ajouter à sa liste de favoris.

c. Diagramme de classes serveur

Le diagramme de classe du serveur a été modifié par rapport à sa première itération. Principalement parce que le premier était trop ambitieux. Nous l'avons simplifié et clarifié :



6 - Diagramme de classes serveur

Le serveur s'articule en deux grands domaines fonctionnels : d'un côté, la reconnaissance et la gestion des contenus audio ; de l'autre, l'authentification et l'administration des utilisateurs.

Pour tout ce qui concerne les audios, chaque piste est d'abord décrite par ses attributs de base (titre, artiste, durée) puis découpée en « partitions », elles-mêmes constituées d'une série d'empreintes numériques. Ce choix technique répond à une contrainte importante : l'API n'accepte pas des fichiers .wav supérieurs à 30 Mo, or un morceau de trois minutes peut déjà dépasser cette limite. En morcelant artificiellement la piste, nous stockons chaque segment séparément sans perdre la cohésion de l'ensemble, ce qui permet de recomposer et d'identifier l'origine lors d'une requête de reconnaissance. Le processus de génération d'empreintes s'effectue en trois étapes : conversion du fichier audio en tableau d'octets, extraction des pics temps fréquence, puis hachage de ces pics pour produire la signature de l'audio. Le service audio prend en charge la création, la suppression et l'extension des partitions, la recherche d'un extrait par empreintes, ainsi que l'enregistrement des découvertes dans l'historique ou la bibliothèque de favoris de l'utilisateur.

Sur le plan de l'authentification, l'interface IAuthentificationService définit les opérations de login, d'inscription, de déconnexion et de vérification de session. À chaque authentification réussie, le système génère une Session dotée d'un token* à durée de validité limitée, incluant un flag IsAdmin pour distinguer les droits des utilisateurs. L'AuthController expose ces routes REST

et délègue leur exécution au service d'authentification, qui persiste à la fois les informations de l'utilisateur et ses sessions. Enfin, toutes les communications entre le front-end et le back-end reposent sur l'échange de tokens. Les mots de passe ne voyagent jamais en clair : ils sont d'abord hachés* côté client, puis concaténés avec un sel* unique et re-hachés côté serveur, ce qui élimine tout risque d'attaque par rainbow tables*. Seules les routes d'administration vérifient la présence du flag IsAdmin dans le token, assurant une stricte séparation entre les opérations accessibles aux utilisateurs ordinaires et celles réservées aux administrateurs.

II. Travail effectué

i. Front-end

a. Le framework Angular

Un framework est un ensemble d'outils, de composants et de règles qui facilitent et standardisent le développement d'applications. C'est comme une boîte à outils pour les développeurs, qui fournit une structure de base (architecture) et des fonctions toutes prêtes, pour qu'ils n'aient pas à tout coder depuis zéro.

Angular est un framework open-source développé par Google, utilisé pour créer des applications web dynamiques, principalement côté client (frontend). Écrit en TypeScript, et utilisant HTML et CSS, il fournit une structure complète pour développer des interfaces utilisateurs, avec des outils intégrés pour le routage, la gestion des formulaires, les appels HTTP, et la création de composants réutilisables.

Le mot d'ordre d'Angular est cette réutilisabilité des composants. L'objectif est d'avoir une base de code claire et simple, où le code ne se répète pas. On découpe les interfaces graphiques en composants qu'on assemble entre eux et que l'on peut réutiliser à souhait. Les composants sont indépendants et contrôlent une partie de l'écran appelée vue.

Voici par exemple un composant que l'on utilise dans notre application :

```typescript
// music-item.component.ts
import { Component, Input, Output, EventEmitter } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Audio } from '../../models/audio/audio';

@Component({
  selector: 'app-music-item',
  standalone: true,
  imports: [CommonModule],
  templateUrl: './music-item.component.html',
  styleUrls: ['./music-item.component.scss'],
})
export class MusicItemComponent {
  @Input() music!: Audio;
  @Input() mode : 'history' | 'library' | 'admin' = 'library';

  @Output() removeFromHistory = new EventEmitter<Audio>();
  @Output() toggleLike = new EventEmitter<Audio>();
  @Output() removeFromLibrary = new EventEmitter<Audio>();
  @Output() remove = new EventEmitter<Audio>();
}
```

1 2 3 4

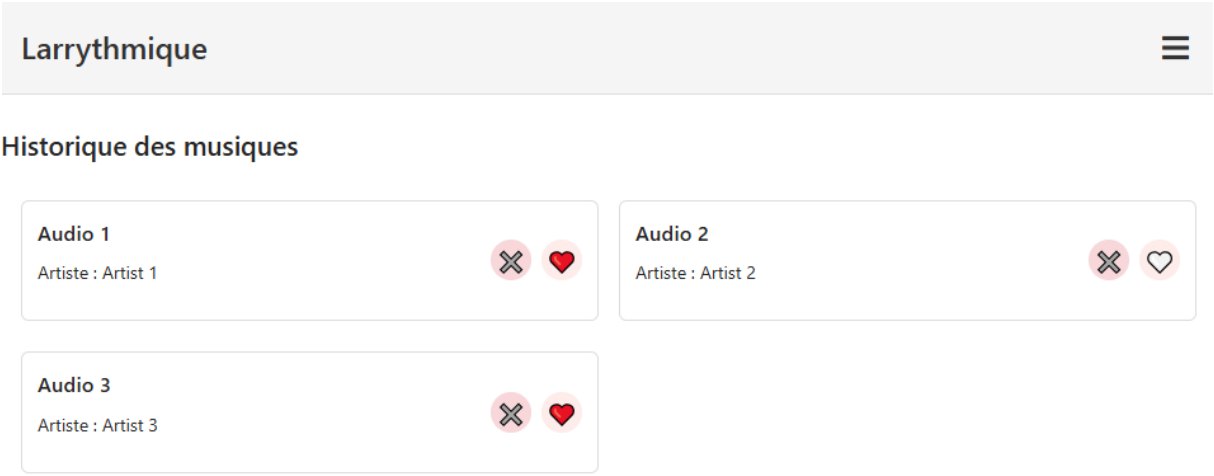
7 - Exemple de composant Angular - TypeScript

Un composant Angular est déclaré en une classe TypeScript, et on lui associe un fichier HTML [1] et un fichier CSS [2] pour définir son aspect. Un composant dispose ensuite d'Inputs [3] pour lui passer des données depuis son composant parent, et des Outputs [4] pour émettre des données à ses composants enfants s'il en a.

Enfin, celui-ci n'en a pas car il n'en nécessite pas, mais on peut également ajouter n'importe quelle méthode TypeScript au composant pour lui donner le comportement que l'on veut, ou lui faire exécuter des calculs ou autres opérations logiques, lors d'un clique bouton par exemple.

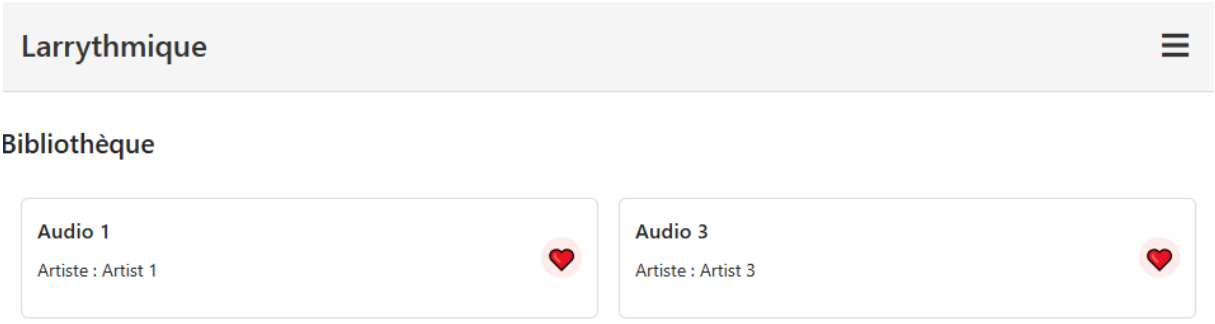
Ce composant représente une musique dans l'application, et grâce au framework Angular, a pu être réutilisé deux fois.

Une première fois pour l'historique des musiques :



8 - Historique des musiques

Et une deuxième pour la liste de favoris :



9 - Bibliothèque des favoris

Pour envoyer des données à un back-end, Angular utilise un système de service. Voici par exemple le service qui a pour rôle d'envoyer un audio à notre API :

```
import { HttpClient, HttpHeaders } from '@angular/common/http';
import { Injectable } from '@angular/core';

@Injectable({
  providedIn: 'root'
})
export class AudioService {

  constructor(private httpClient: HttpClient) { }

  sendAudio(audioBlob: Blob) {
    const formData = new FormData();
    formData.append('file', audioBlob, 'audio.wav');

    return this.httpClient.post("https://localhost:44334/audios/upload", formData, {
      responseType: 'text'
    });
  }
}
```

1 2 3 4

10 - Exemple de service Angular

Il s'agit essentiellement d'une méthode qui utilise un objet « client HTTP » [1] pour effectuer une requête du type que l'on veut. Ici il s'agit d'une requête de type POST [2], puisque l'on souhaite poster notre fichier audio au serveur. Il suffit ensuite d'indiquer l'adresse URL [3] qui mène au contrôleur du back-end que l'on souhaite contacter, et joindre la donnée à poster [4].

b. Electron JS

Electron.js est un framework open-source qui permet de créer des applications de bureau multiplateformes (Windows, macOS, Linux) en utilisant des technologies web comme HTML, CSS et JavaScript. Il combine Chromium* (pour l'affichage) et Node.js* (pour accéder au système de fichiers, réseau, etc.), ce qui permet de créer des applications desktop avec une seule base de code. Des applications comme Discord et Teams sont construites avec Electron.

Dans notre projet, il sert à « empaqueter » notre application Angular comme un logiciel exécutable natif, installable sur l'ordinateur de l'utilisateur. Ca nous permet de bénéficier de la souplesse du développement web tout en conservant les avantages d'une application desktop. Le principal intérêt d'Electron dans notre contexte est de rendre notre interface utilisateur développée avec Angular accessible sans navigateur, et capable d'interagir avec le système de fichiers (notamment pour lire les fichiers audio).

Electron est configuré via un fichier appelé « app.js ». On y gère la création de la fenêtre de l'application [1], et ses paramètres, tels que sa taille [2] et les permissions dont elle dispose [3].

```
const { app, BrowserWindow, ipcMain, desktopCapturer } = require('electron');
const path = require('path');
const url = require('url');

let mainWindow;
let audioStream = null;

function createWindow() {
  mainWindow = new BrowserWindow({
    width: 1200,
    height: 800,
    webPreferences: {
      contextIsolation: true,
      nodeIntegration: false,
      preload: path.join(__dirname, 'preload.js'),
      sandbox: false,
      enableRemoteModule: false,
      // Configuration spécifique pour la capture audio
      webSecurity: false,
      allowRunningInsecureContent: true,
      // Permissions nécessaires
      permissions: {
        audioCapture: true,
        desktopCapture: true
      }
    },
    autoHideMenuBar: true,
  });
```

1 2 3

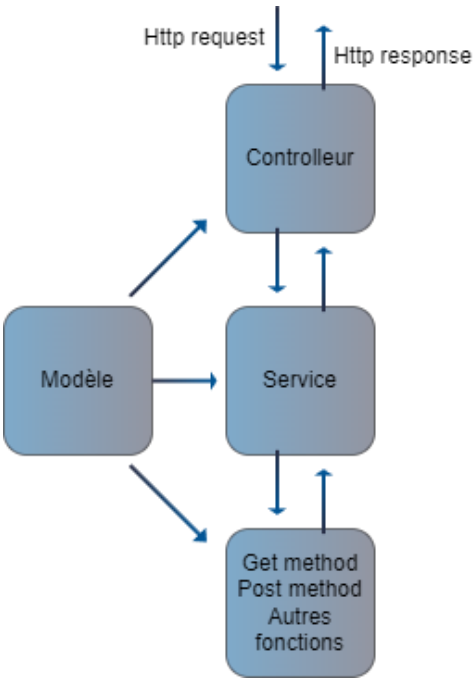
11 - Extrait du app.js

ii. Back-end

a. Le principe d'API

Une API, c'est un service qui permet de répondre à des questions et de faire le lien entre la base de données et le front-End. Elle garantit la sécurité, l'authenticité des demandes, et que les informations soient bien formatées. C'est un peu comme un traducteur entre deux personnes qui parlent des langues différentes : on lui demande de transmettre notre message et de nous traduire la réponse. Mais ici, ce traducteur veille aussi à ce que les échanges se fassent correctement, et que chaque personne aient bien le droit de communiquer.

La structure de base d'une API est la suivante :



12 - Schéma d'architecture d'une API

Un contrôleur s'occupe de recevoir les requêtes HTTP, et d'appeler le service correspondant à la demande contenue dans la requête. Un service est un groupement de fonctions travaillant ensemble pour mettre en place une fonctionnalité de l'application. Dans le service se trouve donc une multitude de fonctions et le contrôleur va appeler la fonction demandée par la requête http. Chacun de ces éléments s'appuie sur le ou les modèles qui servent à définir la structure des données qui transitent dans cette API. Le modèle définit par exemple les noms des données, leurs types, si elles sont nullable ou non.

b. L'algorithme de reconnaissance audio

En partant du principe que notre audio est déjà dans le bon format (PCM*, mono ou stéréo, fréquence d'échantillonnage constante, etc.), l'algorithme peut se découper en quatre grandes étapes :

- **Découpage en fenêtres :**
  Le signal audio est découpé en tranches de taille fixe (ex. 4096 échantillons) [1]. Cela permet d'analyser l'évolution du son dans le temps.
- **Application de la fenêtre de Hanning* :**
  Pour chaque tranche, on applique une fenêtre de Hanning [2] pour atténuer les effets de bord et améliorer la précision de la transformée de Fourier*.
- **Transformée de Fourier rapide (FFT) :**
  On convertit la tranche du domaine temporel au domaine fréquentiel. [3]
- **Récupération des plus hauts pics :**
  On extrait les fréquences les plus intenses (les « pics ») [4] de chaque tranche, en gardant à chaque fois le temps t auquel elles apparaissent. On obtient alors une liste [5] de couples (fréquence, temps).

```
int sampleSize = 4096;
int stepSize = 4096;
int sampleBytes = 2 * channels; // bytes per frame (sample * channels)

int windowByteSize = sampleSize * sampleBytes;
int stepByteSize = stepSize * sampleBytes;
```

1

```
// Hanning window
for (int j = 0; j < window.Length; j++)
    window[j] *= (0.5 * (1 - Math.Cos(2 * Math.PI * j / (window.Length - 1))));

Complex[] fftResult = window.Select(x => new Complex(x, 0)).ToArray();
FourierTransform.FFT(fftResult, FourierTransform.Direction.Forward);

double[] magnitudes = fftResult.Select(c => c.Magnitude).ToArray();

int peakCount = 3;
var topPeaks = magnitudes
    .Skip(10)
    .Select((value, index) => new { value, index = index + 10 })
    .OrderByDescending(p => p.value)
    .Take(peakCount)
    .Select(p => p.index);

foreach (var peak in topPeaks)
{
    peaksWithTime.Add((peak, timeInSeconds));
}
```

2

3

4

5

13 - Extrait de code du backend Services/Utils/AudioConverter.cs/ExtractPeaksWithTime

Une fois cette liste construite, on passe à l'étape de génération des empreintes :

Pour chaque pic, on le considère comme un **point d'ancrage** [1] et on va créer plusieurs paires avec d'autres pics situés à proximité dans le temps. Chaque paire donne une empreinte composée de : [2]

- La fréquence du point d'ancrage,
- La fréquence du point cible,
- La différence de temps entre les deux (deltaT),
- Et le temps initial t du point d'ancrage.

On crée ensuite un hash* à partir de ces 4 paramètres [3].

Après tout ça, on se retrouve seulement avec nos empreintes de l'audio, mais cela nous permet maintenant de les comparer afin de trouver l'audio qui lui ressemble le plus.

```
public List<FingerPrint> GenerateFingerprints(List<(int Frequency, double Time)> peaks)
{
    var fingerprints = new List<FingerPrint>();
    int fanOut = 5;
    for (int i = 0; i < peaks.Count; i++)
    {
        var anchor = peaks[i];
        for (int j = 1; j <= fanOut && (i + j) < peaks.Count; j++)
        {
            var target = peaks[i + j];
            double deltaT = target.Time - anchor.Time;
            if (deltaT > 0 && deltaT < 5)
            {
                var roundedDeltaT = Math.Round(deltaT, 2);
                var hashInput = $"{anchor.Frequency}|{target.Frequency}|{roundedDeltaT}";
                var hash = ComputeHash(anchor.Frequency, target.Frequency, deltaT);
                fingerprints.Add(new FingerPrint
                {
                    T1 = anchor.Time,
                    Hash = hash
                });
    }}}
    return fingerprints;
}
private string ComputeHash(int f1, int f2, double dt)
{
    var input = $"{f1}|{f2}|{Math.Round(dt, 2)}";
    using (var sha1 = System.Security.Cryptography.SHA1.Create())
    {
        var bytes = System.Text.Encoding.UTF8.GetBytes(input);
        var hashBytes = sha1.ComputeHash(bytes);
        return Convert.ToBase64String(hashBytes);
}}
```

1

2

3

14 - Extrait de code du backend Services/Utils/AudioConverter.cs/GenerateFingerprints

Maintenant, pour effectuer notre recherche :

On récupère toutes les empreintes de la base de données qui ont le même hash que ceux de l'extrait [1]. Ces hashes servent de raccourci pour repérer rapidement des similarités.

Pour chaque empreinte qui correspond [2]:

- On calcule la différence de temps entre le moment où elle apparaît dans la base (db.T1) et dans l'extrait (query.T1).
- On note cette différence comme un décalage (ou offset), arrondi à la seconde près.
- On compte ensuite combien de fois chaque décalage apparaît **pour chaque morceau audio** dans la base.

On sélectionne l'audio dont les empreintes ont un grand nombre de décalages identiques [3].

Cela signifie que beaucoup de paires (fréquence1, fréquence2, deltaT) sont présentes dans l'extrait au même moment que dans cet audio, ce qui est un signe fort de correspondance.

On vérifie maintenant que la correspondance est suffisante [4] (pas une valeur trop basse). Si c'est le cas, alors on considère qu'il n'y a pas de correspondance fiable. Sinon, on peut considérer qu'on a l'audio qui ressemble le plus à l'extrait envoyé à notre API.

```
public async Task<AudiosSearchDto> SearchAudioWithFingerprintsAsync(IFormFile file)
{
    var queryFingerprints = await GetFingerPrints(file);
    if (queryFingerprints == null || !queryFingerprints.Any())
        throw new Exception("No fingerprints");

    var queryHashes = queryFingerprints.Select(fp => fp.Hash).ToHashSet();
    // Fetch all matching fingerprints from DB
    var dbMatches = await context.FingerPrint
        .Where(fp => queryHashes.Contains(fp.Hash))
        .ToListAsync();

    if (!dbMatches.Any())
        throw new Exception("No match found.");

    // PartitionId -> Offset -> Count
    var offsetMap = new Dictionary<int, Dictionary<int, int>>();

    // Handle multiple query fingerprints with same hash
    foreach (var dbFp in dbMatches)
    {
        var matchingQueryFps = queryFingerprints.Where(q => q.Hash == dbFp.Hash);
        foreach (var queryFp in matchingQueryFps)
        {
            int offset = (int)Math.Round(dbFp.T1 - queryFp.T1);

            if (!offsetMap.ContainsKey(dbFp.PartitionId))
                offsetMap[dbFp.PartitionId] = new Dictionary<int, int>();
            if (!offsetMap[dbFp.PartitionId].ContainsKey(offset))
                offsetMap[dbFp.PartitionId][offset] = 0;
```

1

2

```
            offsetMap[dbFp.PartitionId][offset]++;
        }
    }

    // Find best match: Partition with most consistent time offset
    var bestMatchPartition = offsetMap
        .Select(entry => new
        {
            PartitionId = entry.Key,
            MaxCount = entry.Value.Values.Max()
        })
        .OrderByDescending(e => e.MaxCount)
        .FirstOrDefault();

    // Use a dynamic threshold based on input size
    int dynamicThreshold = Math.Max(10, queryFingerprints.Count / 20);

    if (bestMatchPartition == null || bestMatchPartition.MaxCount < dynamicThreshold)
        throw new Exception("No confident match.");

    // Get audio and partition info
    var partition = await
context.AudioPartition.FindAsync(bestMatchPartition.PartitionId);
    if (partition == null)
        throw new Exception("Partition not found.");

    var audio = await context.Audios.FindAsync(partition.AudioId);
    if (audio == null)
        throw new Exception("Audio not found.");

    return new AudiosSearchDto
    {
        Id = audio.Id,
        Name = audio.Name,
        Artist = audio.Artist,
        Duration = audio.Duration,
        MatchScore = bestMatchPartition.MaxCount,
    };
}
```

3

4

15 - Extrait de code du backend Services/AudioServices.cs/SearchAudioWithFingerprintsAsyn

c. L'authentification

Afin de permettre à l'utilisateur de retrouver les musiques détectées ou de poursuivre son expérience via des enregistrements personnalisés, il a été nécessaire d'introduire une gestion d'utilisateurs et d'authentification. Cette authentification constitue la base de toutes les interactions liées au stockage et à la personnalisation. Seule la fonctionnalité de détection sonore est accessible sans authentification.

L'authentification a été mise en place en deux volets : l'un côté application (frontend), l'autre côté API (backend).

L'application permet à l'utilisateur de créer un compte, de se connecter, de consulter son profil, son historique et sa bibliothèque. L'interface utilisateur repose sur des services Angular responsables de l'envoi de requêtes HTTP vers l'API pour la gestion des données. Les composants sont réactifs et s'abonnent à l'état d'authentification pour s'adapter dynamiquement aux changements (par exemple, redirection vers la page de connexion si l'utilisateur est déconnecté ou l'onglet Administrateur qui apparait).

Une fois connecté, un token d'authentification est reçu et stocké temporairement. Ce token est attaché à toutes les requêtes sécurisées vers l'API afin d'authentifier l'utilisateur.

De son côté, l'API reçoit les requêtes HTTP et gère le stockage des utilisateurs ainsi que de leurs données associées (musiques aimées, historique, etc.). Lors de l'authentification, elle génère un token de session à durée limitée qui contient également des informations sur les rôles (ex. : si l'utilisateur est administrateur). Ce token est ensuite renvoyé à l'application, permettant d'accéder aux autres fonctionnalités protégées de l'API.

En matière de sécurité des mots de passe, nous avons suivi les préconisations de l'ANSSI* pour éviter toute exposition ou compromission. Les mots de passe ne circulent jamais ni ne sont stockés en clair : avant même d'être transmis, ils sont hachés côté client à l'aide de SHA-256, de sorte qu'aucune donnée saisie par l'utilisateur ne voyage sur le réseau sans protection.

Dès réception, l'API reprend cet haché, le concatène à un sel unique généré lors de l'inscription, puis le hache à nouveau, on a alors :

$$SHA\_256(SHA\_256(password) + salt)$$

Le résultat ainsi obtenu est comparé à celui enregistré en base, ce qui empêche les attaques par rainbow tables ou pré-calculs.

Enfin, tous les tokens d'authentification émis sont à durée de vie limitée et contrôlés systématiquement à chaque requête ; seuls ceux comportant une information de rôle administrateur autorisent l'accès aux routes sensibles de l'API.

Un utilisateur en base de données est donc identifié de la manière suivante :

	Mail	PasswordHash	Salt	IsAdmin
1	merlin.yann63(	Rf7uvJN/gcptiKwZkmeg69K7GNH+K6Z2uUbWa2U/cT0=	IgMiCgeMeAxEzT/Pgt3hFMQeMXx1GI9gWWeEEIX3j9g=	1

16 - Stockage d'un user en BD

iii. Base de données

a. Technologies utilisées

Pour notre application, nous avons choisi d'utiliser PostgreSQL. Ce choix s'est fait assez naturellement : nous étions déjà familiers avec ce système de gestion de base de données, et certains d'entre nous l'utilisent régulièrement en entreprise.

Nous avons également utilisé Entity Framework dans notre API. Cette technologie, que nous détaillerons davantage dans la partie suivante, permet d'automatiser la création et la mise à jour de la base de données à partir du code.

b. Création et modification automatique

Comme évoqué précédemment, grâce à Entity Framework, nous n'avons jamais eu besoin d'écrire manuellement des scripts SQL pour créer ou modifier la base de données. Une fois correctement configuré (notamment en définissant des relations comme les clés étrangères), Entity Framework est capable d'analyser les classes de notre projet ainsi que l'historique des migrations (des fichiers générés à chaque évolution du modèle de données).

Il peut alors automatiquement générer les scripts nécessaires pour ajouter, modifier ou supprimer des tables, des colonnes ou des contraintes dans la base de données.

De plus, lors de l'installation d'Entity Framework, nous devons choisir une version compatible avec le système de gestion de base de données utilisé (SQL Server, Oracle, PostgreSQL, etc.). Ainsi, toutes les petites différences de syntaxe entre ces langages qu'on oublie parfois en écrivant nos propres scripts SQL ne posent ici aucun problème, puisque ce n'est pas à nous de les gérer : c'est Entity Framework qui s'en charge automatiquement.

17 - Les bibliothèques utilisées pour la BD

iv. Serveur

Afin de pouvoir accéder à notre base de données et à notre API depuis n'importe quel poste disposant d'une connexion Internet, nous avons choisi d'utiliser un serveur cloud*. Ce serveur a été loué chez DigitalOcean, principalement en raison de son bon rapport qualité/prix et des nombreux avis positifs.

Il s'agit plus précisément d'un Docker Droplet, c'est-à-dire une machine virtuelle préconfigurée pour exécuter des conteneurs Docker. Docker permet de faire tourner des applications dans des environnements isolés appelés conteneurs, ce qui facilite leur déploiement, leur mise à jour et leur portabilité.

Notre Docker Droplet héberge quatre conteneurs :

- Un conteneur **PostgreSQL**, pour la gestion de notre base de données.
- Un conteneur **CaddyServer**, chargé des redirections et de la génération automatique des certificats HTTPS.
- Un conteneur **.NET**, pour faire tourner notre API.
- Un conteneur **pgAdmin**, offrant une interface web pour administrer PostgreSQL.

Le déploiement de l'API depuis un poste de développement vers le serveur est rapide. En utilisant la fonction « Publier » de Visual Studio, nous avons configuré une connexion FTP vers le serveur. Une fois l'API buildée et envoyée, un script de déploiement se charge d'arrêter le conteneur existant, de le reconstruire, puis de le relancer automatiquement.

La seule contrainte rencontrée concerne l'accès à la base de données lorsque nous sommes connectés à Eduroam : pour une raison inconnue, cette connexion bloque l'accès à notre serveur. Il est alors nécessaire d'utiliser un VPN ou une connexion personnelle pour pouvoir exécuter des requêtes vers la base de données.

III. Bilan

i. Bilan technique

Le bilan technique de ce projet est très fourni puisqu'on a eu l'opportunité de réaliser une application FullStack, du front-end au back-end, en passant par la base de données et le serveur.

Premièrement, nous avons appris à utiliser le framework Angular, qui lui-même utilise les langages HTML, CSS et TypeScript. Nous avons donc énormément renforcé nos connaissances en développement web.

Cela parait étrange d'avoir développé des compétences de développement web, alors que notre solution finale est une application de bureau, mais l'explication est simple. Nous avons utilisé le framework Electron.js pour convertir notre application web en application native. Electron est un outil qu'aucun d'entre nous dans le groupe n'avait déjà utilisé. Nous avons donc beaucoup appris à ce niveau là.

Deuxièmement, sur la partie back-end, nos compétences en C# avec le framework .NET ont été mises à l'épreuve. Nous nous en sommes très bien sorties sur ce point puisque ce sont des technologies que l'on avait déjà utilisées, et que le principe d'API est déjà un sujet qu'on maitrise. Malgré tout, nous avons beaucoup appris sur le plan logique et mathématique, avec l'utilisation de librairies de traitement numérique du signal. Nous n'en avions utilisé en cours seulement en Python, c'était donc une nouveauté en C#.

Ensuite, nous avons utilisé nos compétences techniques en base de données pour mettre en place une BD PostgreSQL simple, avec quelques tables et quelques associations.

Pour finir, nous avons réalisé des Dockerfiles pour déployer des conteneurs sur notre serveur, pour le back et la bd.

Grâce à ce projet nous avons donc déployé une stack complète et appris énormément sur le plan technique.

ii. Bilan scolaire et humain

Mais les compétences techniques ne sont pas le seul aspect que ce projet nous a permis de développer. En effet, nous avons également beaucoup appris sur le plan humain.

Tout d'abord, ce projet nous a obligés à collaborer en équipe, avec des tâches réparties et des dépendances fortes entre les parties développées. Cela nous a appris à communiquer de façon claire et régulière, notamment lorsque certaines décisions impactaient plusieurs parties du projet (par exemple : les formats des données échangées entre le front-end et l'API, ou la gestion des utilisateurs dans la base de données). Nous avons aussi dû faire face à des moments de désaccord technique ou à des contraintes de temps, ce qui nous a appris à trancher rapidement, tout en respectant les idées de chacun.

Ensuite, le fonctionnement en mode agile, avec des sprints de 10 jours, nous a forcés à organiser notre travail, à anticiper les blocages, et à suivre une discipline collective, même lorsque nous n'étions pas tous disponibles en même temps. Le suivi du Kanban sur GitLab a permis de garder une vision claire de l'avancement et d'éviter que certaines tâches ne tombent dans l'oubli.

Enfin, nous avons pris conscience de l'importance de la transmission de l'information : ce rapport en est un bon exemple, mais aussi les nombreux commentaires dans le code et les schémas de modélisation. Ce sont des aspects qui sont essentiels dans une vraie démarche d'ingénieur.

Ce projet a donc été l'occasion de développer non seulement nos compétences techniques, mais aussi nos compétences humaines et professionnelles : travail en équipe, rigueur, communication, organisation et capacité d'adaptation.

iii. Difficultés rencontrées

Dorian : Les difficultés que j'ai rencontrées de mon coté on été au niveau de la reconnaissance musicale, faire en sorte de générer les empreintes, de parcourir la base de données à la recherche d'empreintes similaires et faire en sorte de pouvoir sauvegarder les musiques/audios dans la base de données aussi.

Yann : Personnellement, la difficulté principale que j'ai rencontré était une difficulté technique. Elle concernait la récupération du flux audio du système de l'utilisateur. En effet , Angular est un framework sécurisé qui utilise le navigateur pour demander à l'utilisateur s'il est bien d'accord pour que le programme accède à son flux audio. Malheureusement en encapsulant le code Angular dans une application Electron, le navigateur n'est plus utilisé, et l'utilisateur n'a donc plus aucun moyen d'autoriser l'accès aux flux audio de sa machine. Pour pallier cette difficulté technique, j'ai effectué des recherches et je suis parvenu à trouver une manière de la contourner. On peut avec Electron définir ce qu'on appelle des « handlers » dans le app.js. Ce sont des fonctions qui auront des accès sur la machine similaires à ceux d'une application native, donc qui n'ont pas besoin de navigateur. J'ai donc configuré un handler pour récupérer les sources audio du pc de l'utilisateur.

Laurian : Au début, j'ai rencontré des difficultés à utiliser Git dans un cadre collaboratif, car je n'avais jamais travaillé avec cet outil auparavant et j'avais peu d'expérience en travail d'équipe avec un système de versioning. De même, découvrir Electron a été un défi, car je n'avais que peu travaillé sur des projets web et je ne maîtrisais pas encore TypeScript. Un autre point difficile a été la gestion du temps et des tâches au sein du groupe. J'avais du mal à évaluer correctement le temps nécessaire pour accomplir mes missions tout en restant synchronisé avec mon groupe. Avec le temps, j'ai pu me familiariser avec Git en l'utilisant régulièrement pour mettre à jour le front-end, même si je rencontre encore des difficultés avec les merges. J'ai également approfondi mes connaissances en développement web, notamment en TypeScript à force de coder et de faire des recherches internet sur des docs forum, et j'ai amélioré ma gestion du temps et des priorités.

Corentin : L'un des premiers défis du projet a été d'évaluer précisément les tâches à réaliser et d'en répartir la charge de manière réaliste dans le temps imparti. Il ne s'agissait pas seulement de découper le projet en étapes logiques, mais aussi d'anticiper le temps que chaque fonctionnalité ou bloc technique allait réellement nécessiter. Cette planification s'est révélée plus complexe que prévu, notamment face aux imprévus techniques ou à la découverte de contraintes tardives. Une autre difficulté importante a été de faire coïncider la conception initiale — souvent idéale et ambitieuse — avec les limites concrètes liées aux outils, aux technologies utilisées ou aux contraintes du serveur, notamment en matière de stockage ou de taille de fichiers. Il a fallu plusieurs fois revoir ou adapter certaines approches, non pas par manque d'idées, mais parce que leur faisabilité technique n'était pas assurée dans le cadre et le temps du projet. Trouver cet équilibre entre une architecture propre, une logique métier claire, et la nécessité d'avoir quelque chose de fonctionnel dans un délai court a été un vrai travail d'ajustement. Cela m'a obligé à faire

des compromis, à repenser certaines priorités, et à accepter que tout ne pouvait pas être implémenté comme imaginé au départ.

iv. Conclusion

Ce projet nous a permis de concevoir et développer une application complète de reconnaissance musicale, en mobilisant beaucoup de compétences techniques et organisationnelles. De la capture d'audio à son analyse, en passant par la gestion de comptes utilisateurs et le déploiement serveur, chaque membre de l'équipe a pu intervenir sur une ou plusieurs briques essentielles du système.

L'enjeu principal n'était pas seulement de réussir à créer une application fonctionnelle, mais aussi de comprendre les technologies et la partie traitement du signal mobilisées et de savoir les faire cohabiter efficacement. Nous avons ainsi appris à utiliser des frameworks modernes comme Angular ou Electron, à développer une API robuste en ASP.NET Core, à manipuler PostgreSQL avec Entity Framework, et à déployer une stack conteneurisée sur un serveur.

Ce projet a également été une expérience concrète de travail en équipe, avec toutes les difficultés et ajustements que l'on peut rencontrer. La coordination, la gestion des imprévus, et les choix techniques à trancher collectivement, ou encore la nécessité de planifier sont des éléments qui nous ont fait progresser en tant que futurs ingénieurs.

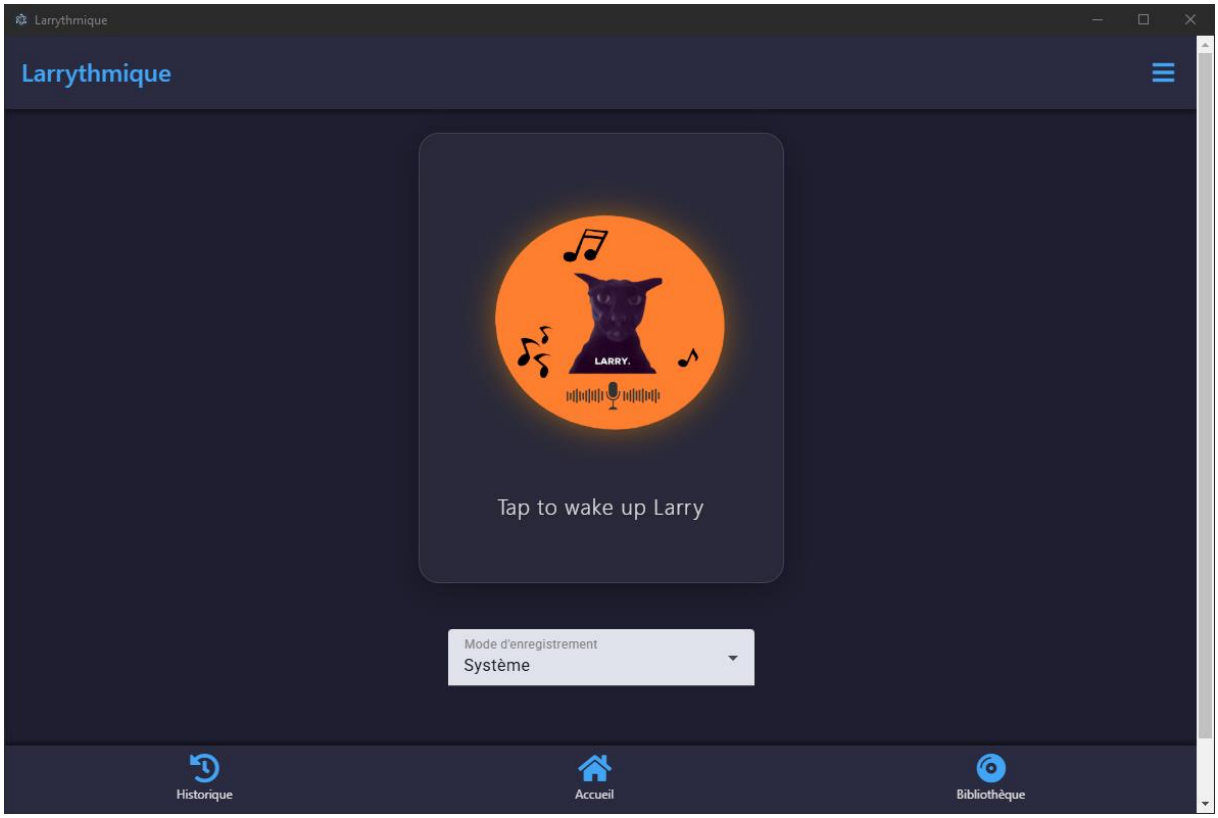
Ce projet nous a prouvé qu'un bon ingénieur ne se définit pas uniquement par ses compétences techniques, mais aussi par sa capacité à apprendre, à collaborer, à s'adapter, et à construire des solutions viables dans un cadre donné.

Nous sommes fiers du résultat obtenu. L'application que nous livrons est stable, fonctionnelle, et répond aux objectifs fixés.

Annexes

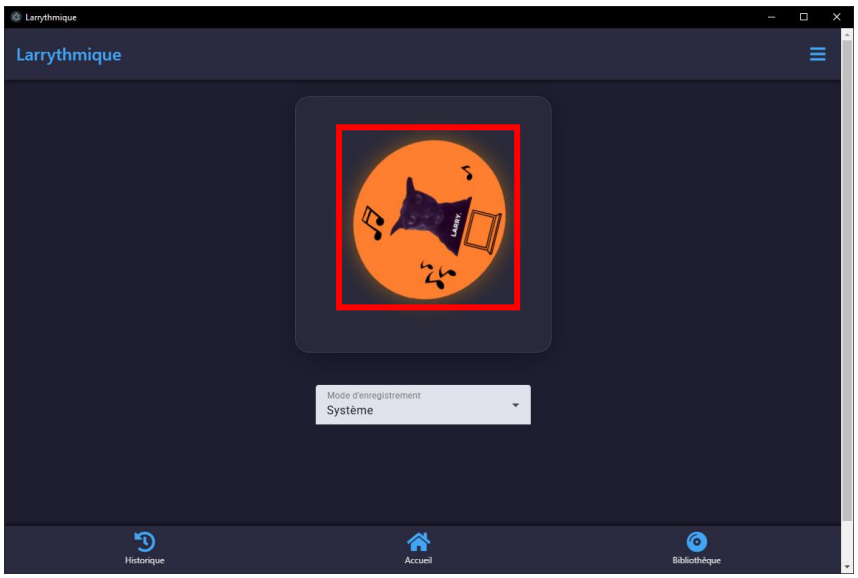
i. Démonstration de l'application

Au lancement de l'application, l'utilisateur arrive sur la page d'accueil :

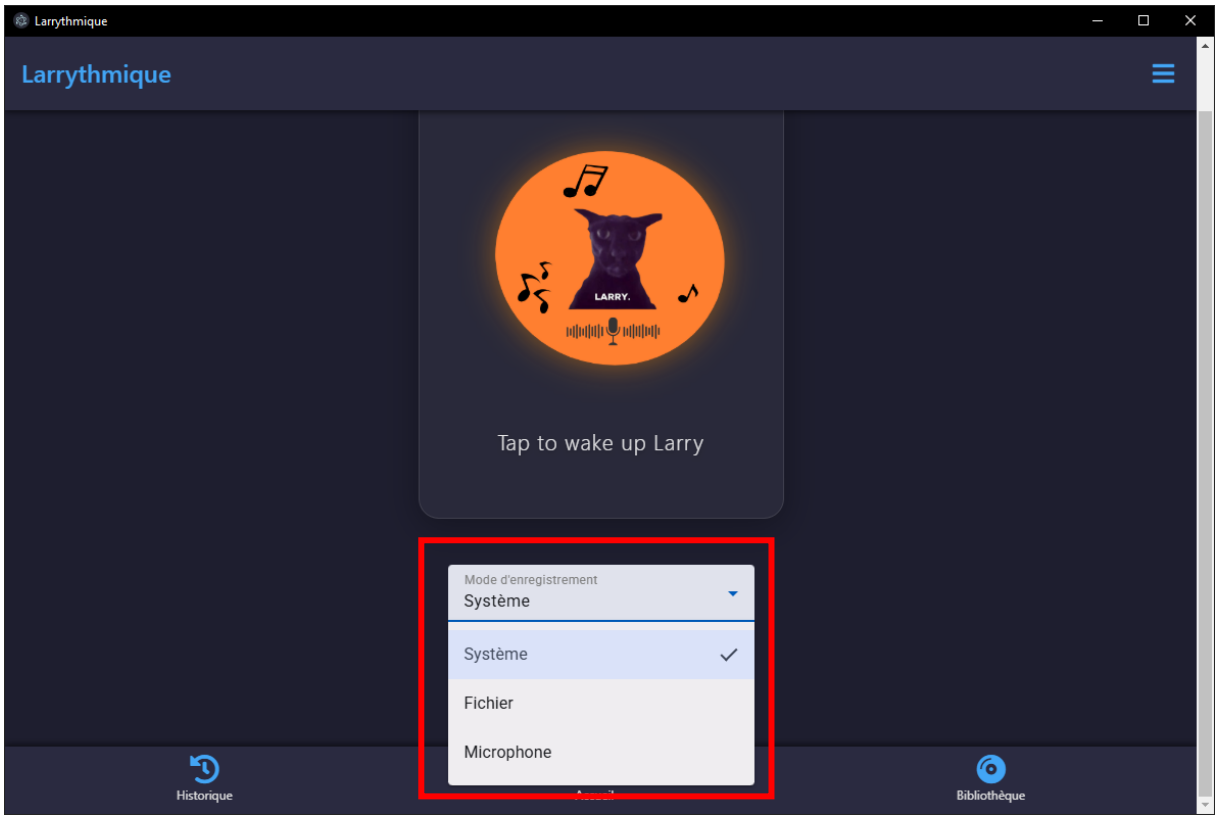


Plusieurs options s'offrent alors à lui.

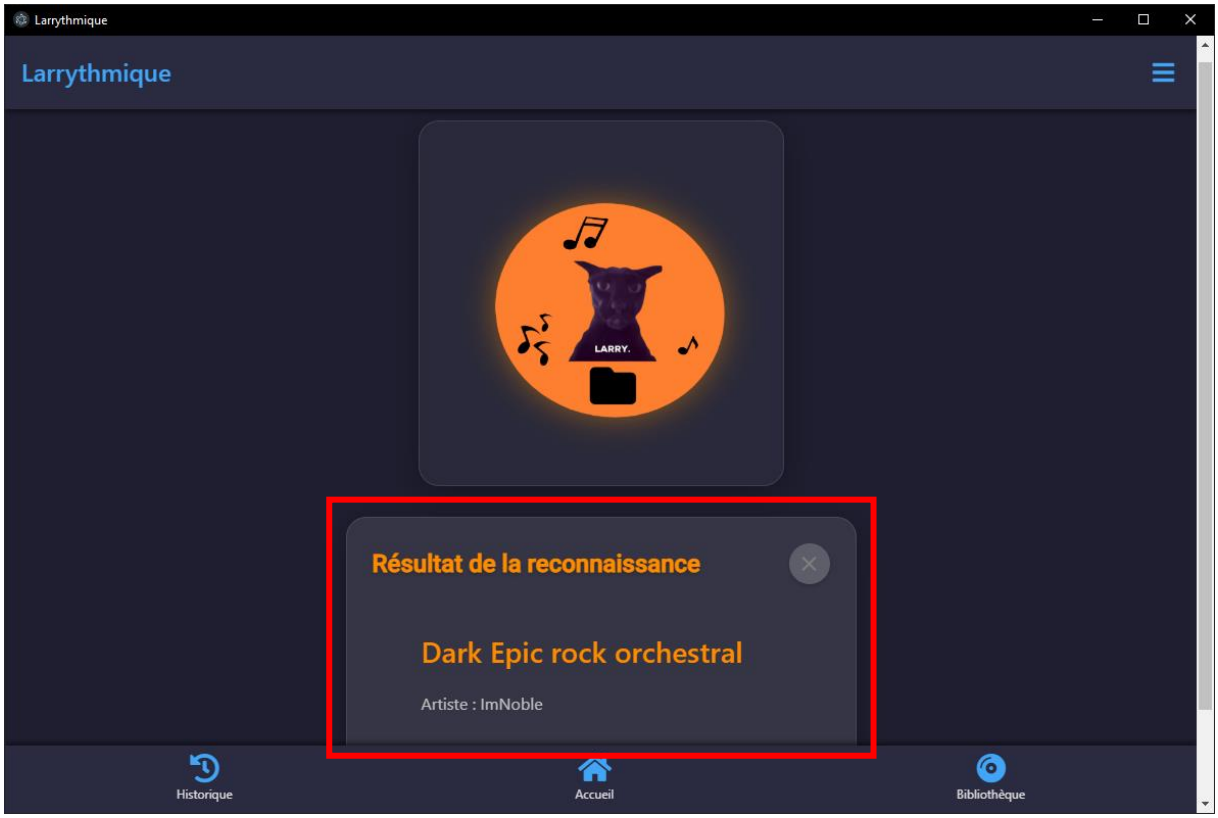
Premièrement, il peut, en tant qu'utilisateur non authentifié, effectuer une reconnaissance musicale via le son système de son ordinateur, c'est-à-dire la piste audio en cours de lecture. Pour cela, il lui suffit de cliquer sur l'icone de Larry au centre de son écran :



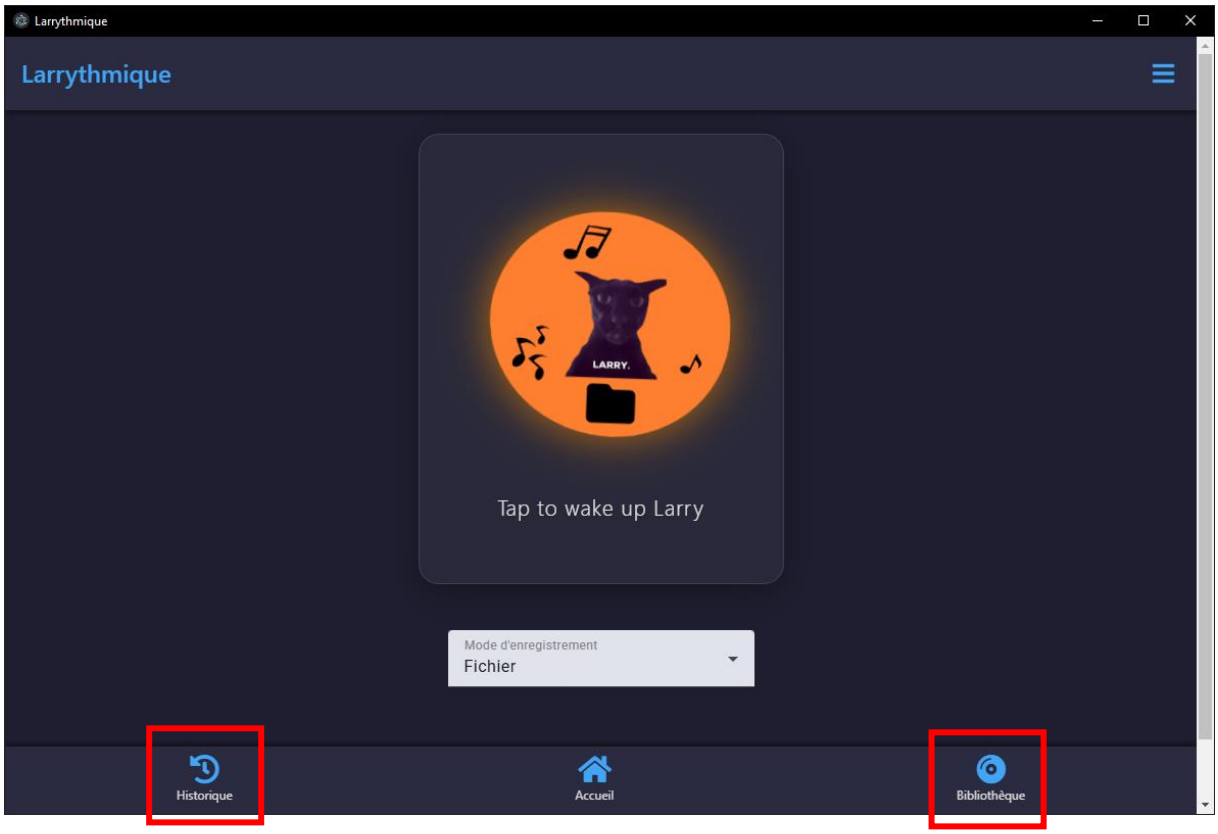
Via un menu déroulant, l'utilisateur peut également choisir entre trois mode d'enregistrement pour son audio :



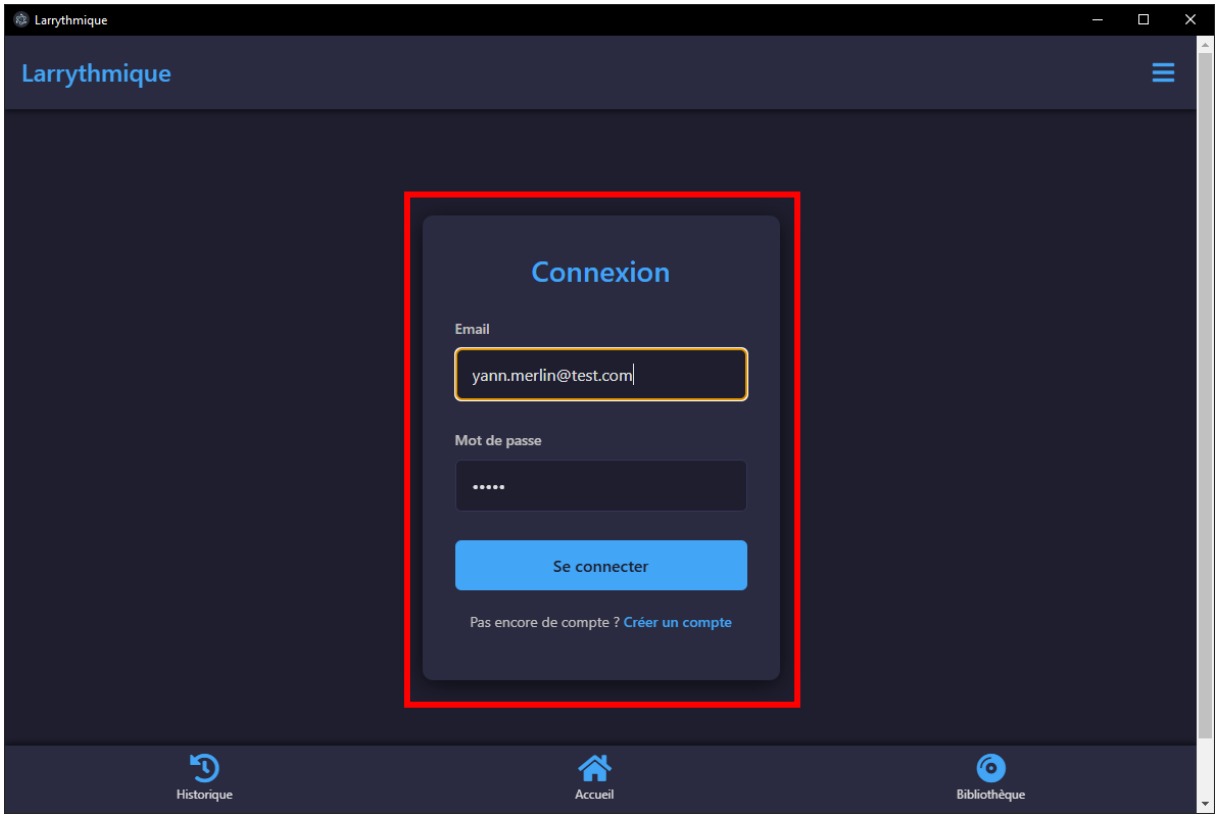
Une fois son mode d'enregistrement choisit, et le bouton Larry central cliqué, son audio est analysé, et il reçoit une réponse lui indiquant si une musique présente en base de données a été identifiée :



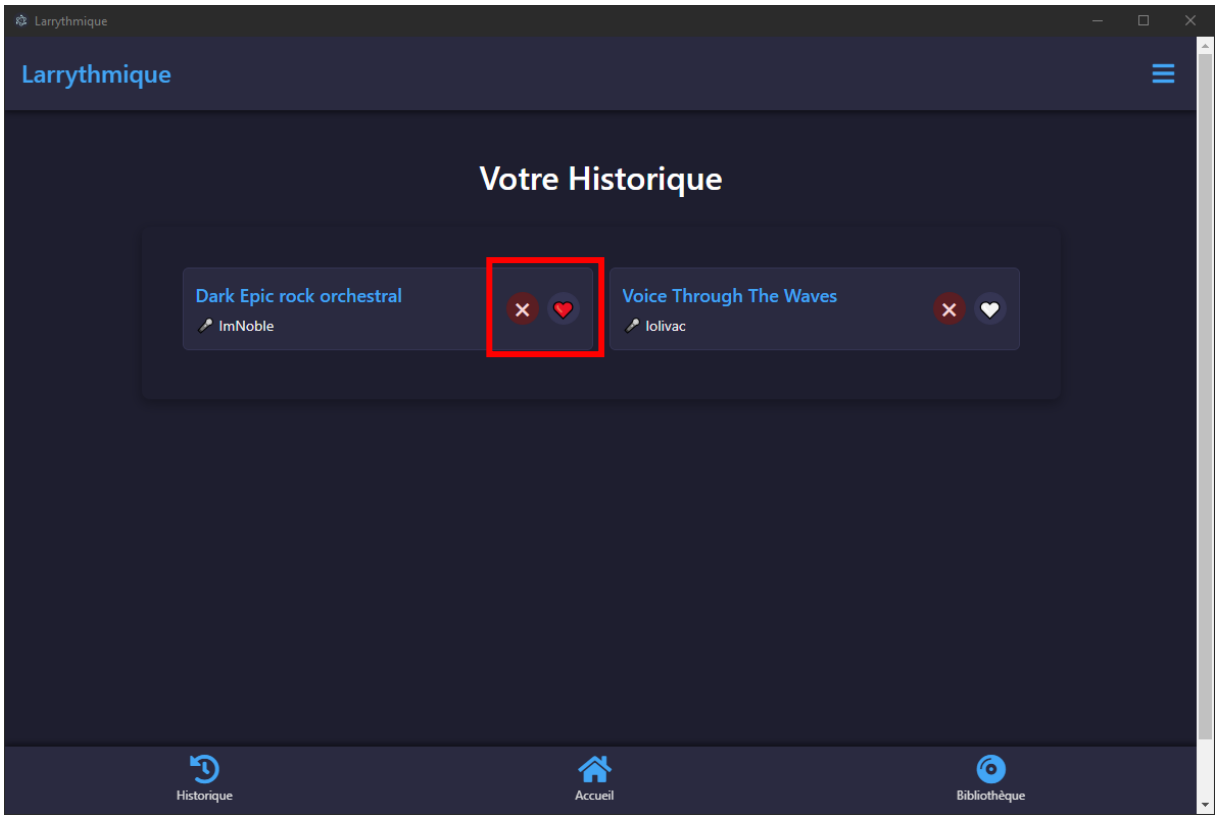
Ensuite, l'utilisateur a à sa disposition deux autres fonctionnalité, la bibliothèque et l'historique :



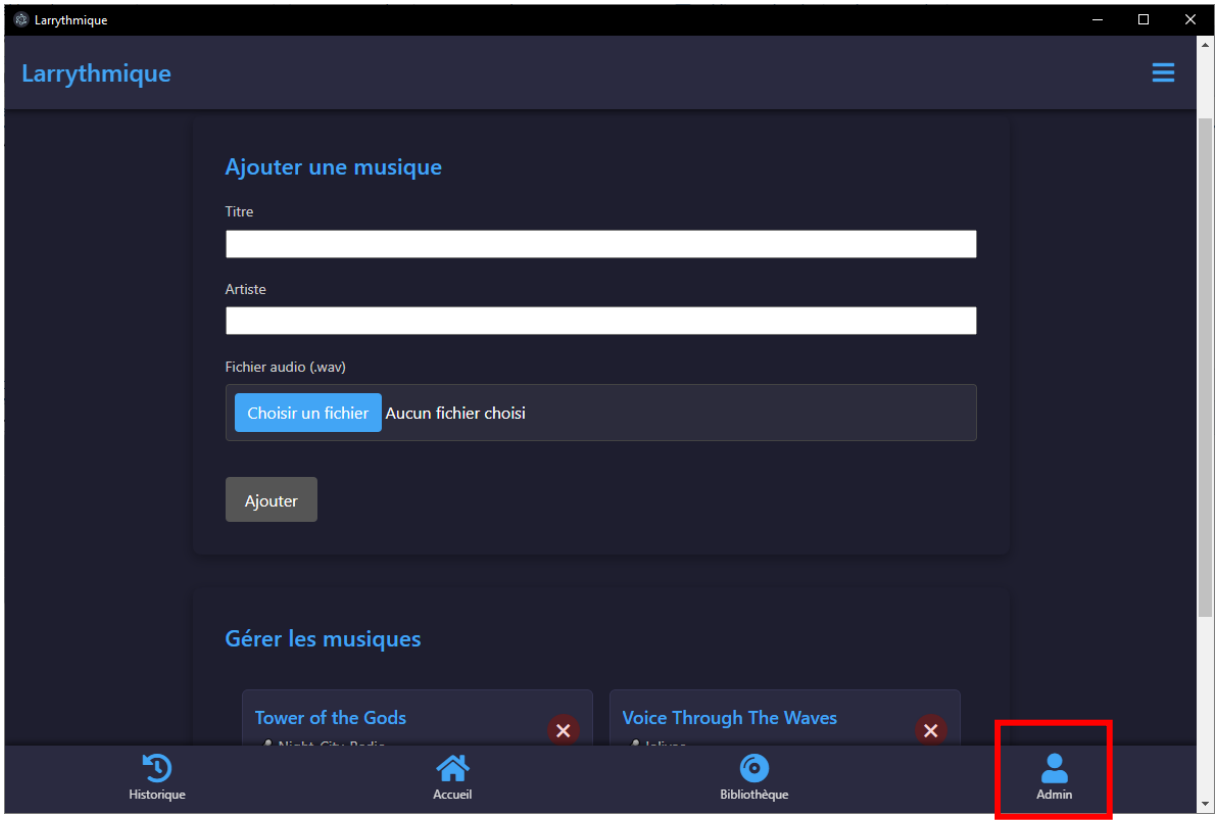
Pour les utiliser, il lui sera demandé de s'authentifier ou de créer un compte :



Ainsi il peut consulter les musiques identifiées, et en ajouter à sa bibliothèque de favoris en cliquant sur l'icone de cœur :



Pour finir, si l'utilisateur est admin, il dispose d'une page supplémentaire pour la gestion de l'application :



ii. Glossaire

Algorithme : Suite d'instructions logiques permettant de résoudre un problème ou d'effectuer une tâche.

Angular : Framework JavaScript utilisé pour créer des applications web dynamiques.

ANSSI : Agence française qui définit des règles de sécurité informatique (Agence nationale de la sécurité des systèmes d'information).

Api : Interface permettant à deux programmes de communiquer entre eux.

Application de bureau : Logiciel installé et utilisé directement sur un ordinateur, sans passer par un navigateur.

Asp.net core : Framework de Microsoft pour créer des applications web et des API en C#.

Back-end : Partie serveur d'une application qui gère la logique, les données et les traitements.

Base de données : Système permettant de stocker, organiser et retrouver des informations.

C# : Langage de programmation orienté objet développé par Microsoft.

Chromium : Moteur de rendu open source utilisé par des navigateurs comme Google Chrome.

Cloud : Ensemble de serveurs accessibles à distance via Internet pour stocker ou exécuter des services.

Conteneur : Environnement isolé contenant tout ce qu'il faut pour faire fonctionner une application.

CSS : Langage utilisé pour définir le style visuel d'une page web (couleurs, marges, polices, etc.).

DigitalOcean : Fournisseur de services cloud proposant des serveurs à louer.

Docker : Outil qui permet de créer, déployer et exécuter des applications dans des conteneurs.

Electron.js : Framework permettant de créer des applications de bureau avec des technologies web.

Empreintes audio : Représentations numériques caractéristiques d'un extrait sonore, utilisées pour l'identifier.

Fenêtre de Hanning : Fonction mathématique utilisée pour adoucir les bords d'un signal avant la FFT.

Framework : Ensemble d'outils et de règles facilitant le développement d'un type d'application.

Front-end : Partie visible d'une application, avec laquelle l'utilisateur interagit (interface).

GitLab : Plateforme de gestion de code source avec outils de collaboration et de suivi de projet.

Haché : Donnée transformée par une fonction mathématique irréversible, souvent pour des raisons de sécurité.

Hash : Code court obtenu en appliquant une fonction de hachage sur une donnée.

HTML : Langage utilisé pour structurer le contenu d'une page web.

IDE : Logiciel regroupant tous les outils nécessaires pour programmer (ex : Visual Studio, VS Code).

Java : Langage de programmation largement utilisé pour les applications web et mobiles.

Javaspringboot : Framework Java pour créer des API et des applications web rapidement.

Mode agile : Méthode de gestion de projet basée sur des itérations courtes et une adaptation continue.

Modélisation : Représentation graphique ou logique d'un système pour le comprendre ou le concevoir.

Node.js : Environnement permettant d'exécuter du JavaScript côté serveur.

Open source : Logiciel dont le code source est librement accessible et modifiable.

oracle : Entreprise proposant des logiciels, notamment un système de base de données propriétaire.

PCM (pour un audio) : Format brut de signal audio, sans compression.

PostgreSQL : Système de gestion de base de données relationnelle open source.

Rainbow tables : Tables utilisées pour retrouver des mots de passe à partir de leurs versions hachées.

React : Librairie JavaScript pour créer des interfaces utilisateur interactives.

Requête HTTP : Message envoyé par un client (navigateur ou appli) pour demander des données à un serveur.

Sel : Donnée aléatoire ajoutée à un mot de passe avant de le hacher, pour renforcer la sécurité.

Serveur : Ordinateur distant qui fournit des services ou des données à d'autres machines.

Spectrogramme : Représentation visuelle de la fréquence d'un son au cours du temps.

SQL : Langage permettant d'interagir avec une base de données (lecture, écriture, requêtes, etc.).

Tableau Kanban : Outil visuel pour suivre l'avancement des tâches dans un projet.

Tauri : Alternative à Electron permettant de créer des applications de bureau plus légères.

Token : Jeton d'identification temporaire utilisé pour authentifier un utilisateur.

Transformée de Fourier : Méthode mathématique pour analyser les fréquences contenues dans un signal.

TypeScript : Langage basé sur JavaScript avec des types pour sécuriser le code.

Windows : Système d'exploitation développé par Microsoft, utilisé sur la plupart des PC.

iii. Ressources utilisées

Recommandations pour l'authentification des utilisateurs :

Sécurité : Authentifier les utilisateurs | CNIL

Outil pour convertir et tester des tokens JWT :

https://jwt.io/

Vidéo explicative d'un projet visant à recréer Shazam en utilisant le langage GO :

https://www.youtube.com/watch?v=a0CVCcb0RJM&t=62s

Tutoriel simple pour connaitre les bases pour utiliser Electron avec Angular :

https://buddy.works/tutorials/building-a-desktop-app-with-electron-and-angular

Outil de test pour hasher des mot de passe :

Hash, hashing, and encryption toolkit

Le brevet de l'application Shazam :

https://moodle2024.uca.fr/pluginfile.php/591177/mod_resource/content/2/US7627477.pdf