IUT Villetaneuse
Université Sorbonne Paris Nord

UNIVERSITÉ SORBONNE PARIS NORD

SYNAPSE AUDIO

Synapse Audio : Plus qu'un son, une onde d'excellence.

RAPPORT Compétence 2

Développement d'Application Web

Projet : **SoundStream**

Équipe : Synapse Audio

Rédigé par :
- Kadir CASSEL
- Aboubakry SY
- Saif-eddine ALJANE
- Romain HEURTAUX
- Tristan COLLEN

Établissement : IUT de Villetaneuse - Université Sorbonne Paris Nord – BUT2 Informatique
Date : Janvier 2026

Introduction

Ce rapport expose notre réflexion sur l'optimisation et la sécurité de l'application web que nous avons développée dans le cadre de la SAE S3.01. Il a pour but d'expliquer les choix que nous avons faits pour garantir un certain niveau d'optimisation et de sécurité dans le développement de notre application. Pour cela, nous aborderons l'architecture de notre application, la sécurité des données, la sécurisation de l'authentification et enfin les avantages d'un Front-end modulaire.

1 Une architecture optimisée

Flask est très permissif, il n'impose pas de structure stricte, on pourrait tout mettre dans quelques fichiers très compacts cela marcherait. Mais pour des raisons pratiques et de cohérence du code, nous avons donc décidé de structurer notre application avec une architecture MVC (Modèle-Vue-Contrôleur). Cette structure vise à disposer un cadre clair dans la répartition des rôles de chaque partie du code, ce qui permet d'optimiser le maintien du site, le débogage et la mise à jour de celui-ci :

- **Le Modèle :** Assure l'intermédiaire entre le stockage concret (ici sous forme SQL) et la représentation abstraite utilisée par l'application. En résumé, elle convertit les données concrètes stockées dans la base de données en données abstraites, autrement dit le DAO (Data Access Object). Ainsi, la couche Service interagit seulement avec une instance du DAO, ce qui optimise l'accès et la gestion/manipulation des données, car cette responsabilité est isolée du reste.
- **Le Service :** Cette couche contient la logique métier. Elle récupère les données via le DAO et les manipulent selon la demande du contrôleur. En général sur notre projet, il va souvent renvoyer des dictionnaires et des listes pour que les templates puissent diffuser des lignes d'informations pour le dashboard et les autres pages.
- **Le Contrôleur :** C'est un peu le relais de l'application web, il va recevoir les requêtes HTTP du client, gérer les droits de celui-ci avec des décorateurs, traiter sa requête et lui renvoyer une réponse en Vue adaptée grâce à la bonne gestion des routes après avoir préalablement chercher toutes les données adéquates grâce aux services côté serveur.
- **Le Template (Vue) :** Il s'agit de la couche où se trouve le HTML, mais il contient les données qui ont été envoyées dans la requête HTTP envoyée par le contrôleur. Ces données ont été intégrées grâce au moteur Jinja. Cela permet d'optimiser la lisibilité du code car il n'y a pas de HTML mélangé à du python

(Voir Annexe 1 : arborescence de la structure de notre application)

2 Sécurité des données

L'injection SQL est une des vulnérabilités les plus critiques des applications web. Elle permet à un attaquant d'exécuter du code SQL malveillant en manipulant les entrées utilisateur.

Pour contrer cette menace, notre solution consiste à utiliser systématiquement des requêtes paramétrées pour séparer le code SQL des données utilisateur.

(Voir Annexe 2 : Comparaison entre code vulnérable et requêtes paramétrées)

Avec les requêtes paramétrées :

- La commande SQL est fixe et ne peut pas être modifiée.
- Les données utilisateur sont traitées comme des valeurs, pas comme du code.
- Le connecteur échappe automatiquement les caractères dangereux.
- Il est impossible d'injecter du SQL, même avec des entrées malveillantes.

Gain pour l'optimisation : Au-delà de la sécurité, les requêtes paramétrées permettent au moteur de la base de données de réutiliser les plans d'exécution, ce qui améliore légèrement les performances.

3 Sécurisation de l'authentification

Concernant les mots de passe, quelques règles essentielles sont appliquées. La première est de ne jamais enregistrer les mots de passe en clair dans la base de données. Il faut toujours chiffrer ou hacher l'accès côté serveur en utilisant la méthode **salt** qui sera choisie aléatoirement.

La méthode salt est une chaîne de données aléatoire et unique qui va être ajoutée au mot de passe d'un utilisateur avant que celui-ci ne soit haché. Son objectif est de rendre chaque hash unique, même pour des mots de passe identiques. Il y aura obligatoirement un salt généré avant le hash de chaque mot de passe, et il doit être unique pour éviter des problèmes. En effet, si deux utilisateurs utilisent le même mot de passe et que le système utilise le même code salt,

alors le code haché sera le même. Si un utilisateur malveillant accède à la base de données, il verra deux codes hachés similaires et saura immédiatement que ces utilisateurs ont le même mot de passe. C'est pour cela que, en rendant le salt unique, un même mot de passe « azerty » aura deux codes hachés différents !

En complément, nous utilisons la méthode **pepper**. C'est une chaîne unique, comme le salt, mais la clé est sauvegardée en dehors de l'application, ce qui évite de la rendre accessible aisément par un utilisateur malveillant. Il est important de prendre conscience que, si le pepper est perdu, alors tous les mots de passe deviendront inutilisables. La logique de la requête finale suit ce principe : `hash(salt, pepper, mot_de_passe)`.

Pour l'implémentation technique, nous avons choisi **Bcrypt**. C'est celui qui va gérer le salt pour nous : il génère automatiquement le salt et l'inclut directement dans le code haché correspondant au mot de passe de l'utilisateur. Dans ce cas, il faut concaténer la chaîne du mot de passe et du pepper secret avant de la mettre dans la méthode bcrypt. La requête finale correspond alors à : `bcrypt(mot_de_passe + pepper)`.

4 Avantages d'un Front-end modulaire

Le front-end répond à une logique de modularité grâce à la technologie de template **Jinja**. Face à la répétition de multiples patterns HTML, CSS et JavaScript, nous appliquons le principe « Bento » (ou *Atomic Design*). Cette méthode utilise des modules pour injecter des fragments de code via une ligne unique. Ce principe harmonise également le design, où l'on retrouve des blocs similaires entre les différents rôles utilisateurs.

L'utilisation de Jinja évite de réécrire systématiquement la structure de base du document (`<DOCTYPE>`, `<head>`, etc.) grâce à l'instruction `extends` (ex : `{% extends layout.html %}`). Elle facilite aussi l'inclusion d'éléments récurrents, comme un en-tête, via l'instruction `include` (ex : `{% include 'header.html' %}`).

Cette approche offre un gain majeur en maintenance. En respectant le principe de non-répétition, nous évitons les erreurs de redondance : une modification sur un fichier parent se répercute instantanément sur toute l'application, sans édition manuelle de chaque fichier.

Le modèle Jinja simplifie aussi l'interaction entre le HTML et les données en s'appuyant sur l'architecture MVC. Il permet d'intégrer des **boucles** et des **conditions** directement dans le template pour gérer les listes (logs, playlists) ou adapter l'affichage selon le rôle de l'utilisateur.

Enfin, cette gestion modulaire facilite l'implémentation des formulaires pour communiquer avec le contrôleur, permettant ainsi de créer des playlists ou de gérer la connexion de manière sécurisée.

5 Annexe :

5.1 Annexe 1 : arborescence de la structure de notre application

```
Code/app/
│
├── controllers/
│   ├── [Entité]Controller.py     --> Un contrôleur dédié pour chaque table de la BDD
│   └── (ex: UserController, PlaylistController...)
│
├── models/
│   ├── [Entité].py               --> L'Objet : Transforme la ligne SQL en Objet Python
│   ├── [Entité]DAO.py            --> Le DAO : Contient les requêtes SQL d'accès
│   └── (Cette structure est répliquée pour chaque entité(Tables de la BD) : User, Log, Playlist...)
│
├── services/
│   └── [Entité]Service.py        --> Logique métier  pour chaque entité de la base de données
│
├── static/
│   ├── database/
│   │   ├── database.db
│   │   ├── initdb.py
│   │   └── schema.sql
│   ├── css/
│   ├── js/
│   └── images/
│
└── templates/
    └── template.html             ---> Tous nos fichiers HTML permettant d'afficher les données envoyées par les controllers
```

5.2 Annexe 2 : Comparaison entre code vulnérable et requêtes paramétrées

1. Approche Vulnérable (Interdite)

```
query = f"SELECT * FROM users WHERE username = '{username}'"
```

Si username vaut "' OR '1'='1", la requête retourne toute la base.

2. Approche Sécurisée (Notre solution)

```
cursor.execute("SELECT * FROM users WHERE username = ?", (username,))
```
Les données sont traitées séparément. L'injection est impossible.