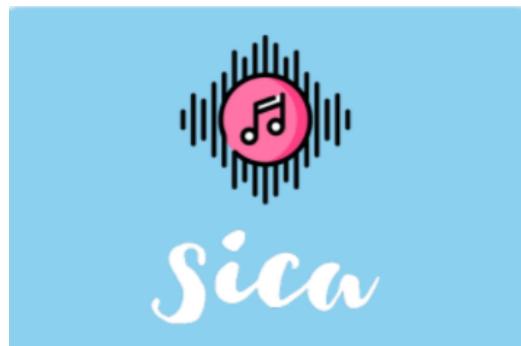


RAPPORT DE PROJET

Plateforme Sociale pour la musique

SICA



Projet réalisé par
Théo Barberot
Jean-François Castellani

Projet encadré par
Christophe Barès
Sylvain Reynal

Table des matières

1. Introduction
2. Exigences du projet
3. Analyse et conception
 - 3.1 Exigences du projet
 - 3.2 Conception fonctionnelle
 - 3.4 Conception technique
 - 3.5 Conception graphique
4. Réalisation technique
 - 4.1 API
 - 4.2 Client
 - 4.3 Socket
5. Organisation et bilan du projet
 - 5.1 Organisation de notre binôme
 - 5.2 Résultats
 - 5.3 Améliorations envisagées
6. Conclusion

1. Introduction

La musique est devenue un acteur majeur dans nos vies et notamment dans l'industrie du divertissement mondial. En effet, près de 96% de la population mondiale a écouté au moins un genre musical en 2019, et près de 40% des français savent jouer d'un instrument. Ainsi l'industrie musicale est amenée à évoluer avec le temps et les technologies dont elle dispose.

Lors des dernières années, avec la montée en puissance des réseaux sociaux et des plateformes de streaming, l'industrie musicale s'est également développée. En effet, de nombreux artistes ont totalement abandonné leur style de musique, leur signature et même leurs propres univers afin de produire des nouvelles musiques appelées "commerciales", destinées uniquement à plaire aux plus grand nombre, et à chercher le plus de "streams" (nombre d'écoutes).

C'est pour remettre la création artistique au centre de l'industrie musicale que l'on propose notre plateforme sociale 'SICA' dédiée au partage de musique, et qui encourage les créateurs à pousser la création, de partager, de communiquer, d'interagir et de collaborer, quelques soit leur niveau, leur goûts et la taille de leur communauté.

2. Exigences du projet

Tout d'abord nous retrouvons les exigences fonctionnelles d'une plateforme sociale classique :

Nous devons pouvoir nous créer un compte personnel, où l'on peut se connecter, se déconnecter, ajouter des informations personnelles (photos, bannières, données notamment sur nos goûts musicaux). Nous devons pouvoir suivre un utilisateur si l'on aime son contenu, lui envoyer un message et donc posséder un espace de messagerie entre les différents utilisateurs. Nous proposons également aux utilisateurs de suivre un ou plusieurs genres musicaux afin d'avoir des suggestions de sons, ou de profils accordés à leurs goûts.

Pour que le concept de 'SICA' soit efficace, il faut que l'on puisse avoir un flux (appelé "feed") de musique que l'on peut faire défiler à l'infini afin d'avoir de nombreuses propositions d'écoutes. De même, on doit avoir la possibilité de poster une musique, (accompagnée d'images, de commentaires ou même de vidéos) que les autres utilisateurs pourront potentiellement retrouver dans leur feed, ou sur notre profil.

Afin de lier les musiciens entre eux, on propose également un espace dédié à la collaboration, où vous pourrez soumettre une requête, une recherche de partenaires pour vous accompagner dans un nouveau projet, où simplement finir un projet, et ainsi engager la discussion directement, et de même créer des groupes.

SICA propose également des options pour les artistes amateurs afin de les aider à s'entourer de conseils et de personnes, c'est-à-dire des espaces dédiés à des questions pour pouvoir progresser, des espaces où certains cours seraient mis à disposition des artistes. Nous proposons également un espace pour pouvoir créer des événements en ligne, ou potentiellement des événements en direct.

3. Analyse et conception

Sica est une plateforme se voulant fonctionnelle et accessible. C'est pour cela que nous avons choisi de réduire le nombre de pages existantes. En effet, celles-ci sont au nombre de 6 :

- Page d'accueil
- Profil
- Messagerie instantanée
- Projets
- Connexion
- Inscription

Puis, ces pages sont remplies à l'aide de composants que l'on insère. Il y a 10 composants différents que nous allons énumérer :

- Amis en ligne
- Amis proches
- Conversations
- Le feed
- Les messages
- Post
- Partage
- Barre de droite
- Barre de gauche
- Barre du haut

Lors de notre premier jet du projet, nous avions des difficultés pour résoudre certaines erreurs car notre code n'était pas très lisible et compréhensible.

Pour cela, nous avons divisé notre projet sur VS Code en 3 parties : l'API, le client et le socket.

L'API (application programming interface), ou back-end, est une interface logicielle qui permet de « connecter » un logiciel ou un service à un autre logiciel ou service afin d'échanger des données et des fonctionnalités, ici notre API sera directement lié à MongoDB, le système de gestion de notre base de donnée que nous avons choisi.

Le client, ou front-end, est destiné à programmer toutes les fonctionnalités, affichages et design de notre plateforme, directement accessible par le client.

Le fichier socket correspond à l'interface de connexion. Il s'agit d'un modèle permettant la communication inter processus afin de permettre à divers processus de communiquer aussi bien sur une même machine qu'à travers un réseau TCP/IP. Dans l'API, nous avons distingué un dossier Models où nous avons mis les modèles pour une conversation, un utilisateur, un projet, un post et un message.

Exemple du modèle pour un post :

```
const moongoose = require("mongoose");

const PostSchema = new moongoose.Schema({
    userId : {
        type : String,
        require : true
    },
    desc : {
        type : String,
        min: 2,
        max : 500
    },
    img : {
        type : String,
    },
    audio : {
        type : String,
    },
    likes : {
        type: Array,
        default : []
    }
},
{timestamps : true}
);

module.exports = moongoose.model("Post", PostSchema);
```

Nous pouvons voir que pour réaliser un post, il faut obligatoirement un utilisateur, ce qui semble logique. On peut aussi avoir une description entre 2 et 500 caractères, une image et un audio. Nous expliquerons dans une partie ultérieure les exports et les différents outils utilisés.

Dans un autre dossier, nous avons mis les routes nécessaires afin de faire fonctionner notre API c'est-à-dire la route pour créer un utilisateur, le modifier ou celle pour ouvrir une conversation entre 2 utilisateurs.

Voici l'exemple de la première route écrite, celle pour s'inscrire sur notre plateforme :

```

//Register
router.post("/register", async (req, res) => {
  try {
    //Generer un nouveau mot de passe
    const salt = await bcrypt.genSalt(10);
    const hashedPassword = await bcrypt.hash(req.body.password, salt);

    //Creer un nouvel Utilisateur
    const newUser = new User({
      username: req.body.username,
      email: req.body.email,
      password: hashedPassword
    })
    //Enregister l'user et réponse
    const user = await newUser.save();
    res.status(200).json(user);
  }
  catch (err) {
    res.status(500).json(err);
  }
});

```

L'ajout de commentaires nous permet de rendre plus accessible notre code à une personne extérieure voulant continuer le projet.

Dans un second temps, nous avons développé le client, aussi appelé le front-end c'est-à-dire les pages avec lesquelles l'utilisateur interagit.

Le client est divisé en 3 sous parties :

- Composants
- Pages
- Context

Les composants comme dit auparavant sont très pratiques dans la construction de nos pages car une fois créés, il suffit de les importer.

Par exemple, le composant Topbar est importé dans toutes les pages du projet en dehors de la connexion et de l'inscription. Cela aurait été contre-productif de réécrire à chaque fois le code pour ce composant.

Exemple du code pour TopBar :

```

export default function Topbar() {

  const { user } = useContext(AuthContext);
  const DP = process.env.REACT_APP_DOSSIER_PUBLIC;

  const handleClick = () => {
    sessionStorage.removeItem('user');
    window.location.reload();
}

```

```

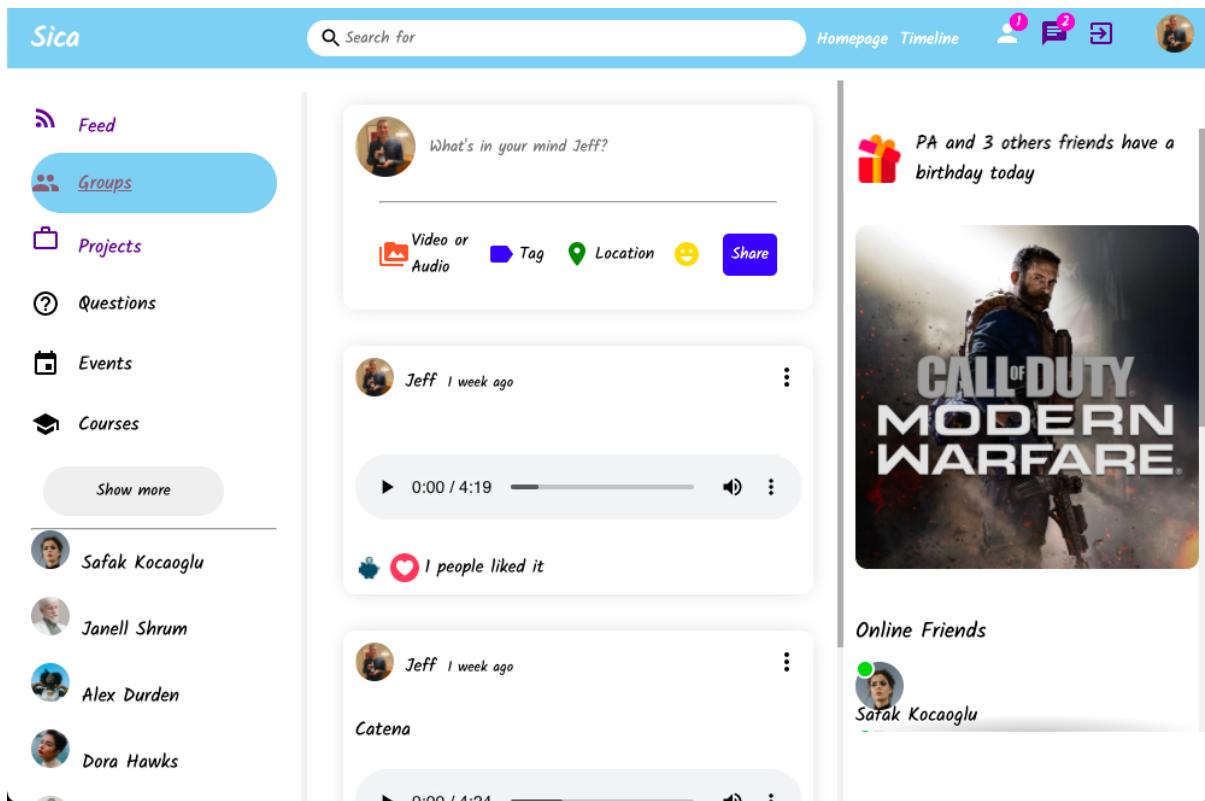
return (
  <div className="topbar-container">
    <div className="topbarLeft">
      <Link to="/" style={{ textDecoration: "none" }}>
        |   <span className="logo">Sica</span>
      </Link>
    </div>
    <div className="topbarCenter">
      <div className="searchbar">
        <Search className="searchIcon" />
        <input placeholder="Search for" className="searchInput" />
      </div>
    </div>
    <div className="topbarRight">
      <div className="topbarLinks">
        <span className="topbarLink">Homepage</span>
        <span className="topbarLink">Timeline</span>
      </div>
      <div className="topbarIcons">
        <div className="topbarIconItem">
          <Person />
          <span className="topbarIconBadge">1</span>
        </div>
        <div className="topbarIconItem">
          <Link to="/chat" style={{ textDecoration: "none" }}>
            <Chat />
            <span className="topbarIconBadge">2</span>
          </Link>
        </div>
        <div className="topbarIconItem">
          <Link to="/register" style={{ textDecoration: "none" }}>
            <ExitToApp onClick={handlerClick}>/>
          </Link>
        </div>
        <div>
          <Link to={`/profile/${user.username}`}>
            <img src={user.profilePicture ? DP + user.profilePicture : DP + "person/noAvatar.jpg"} alt="" />
          </Link>
        </div>
      </div>
    </div>
  </div>
)

```

Notre composant est divisé en 2 parties, la première étant la partie React et la seconde la partie HTML/CSS. Pour gagner en visibilité encore une fois, nous avons choisi de mettre des className avec des noms très explicites comme TopBarRight ou TopBarCenter. Cela rend le fichier CSS plus facile à gérer.

Les pages sont les aspects centraux de notre plateforme, c'est ce qu'il apparaît dans votre navigateur.

Voici la page d'accueil de notre site :



Et le code de cette page:

```

import Feed from "../../components/feed/Feed";
import Rightbar from "../../components/rightbar/Rightbar";
import Sidebar from "../../components/sidebar/Sidebar";
import Topbar from "../../components/topbar/Topbar";
import "./home.css"

export default function Home() {
  return (
    <>
      <Topbar />
      <div className="homeContainer">
        <Sidebar />
        <Feed/>
        <Rightbar/>
      </div>
    </>
  )
}
  
```

Comme dit au-dessus, la page d'accueil est extrêmement accessible, en effet le code tient en 10 lignes. On importe les composants utiles et on les réutilise dans le corps de notre page Home.

Enfin, nous avons la partie Context, ces 3 fichiers permettent de suivre ou de ne plus suivre un utilisateur mais aussi de différencier les différents cas d'une connexion sur la plateforme.

Le Login se divise en 3 parties :

- LOGIN_START : il s'agit de l'état par défaut lorsque l'on ouvre l'application, l'utilisateur prend la valeur nulle.
- LOGIN_SUCCESS : comme son nom l'indique, on passe dans cet état lorsque l'email et le mot de passe correspondent aux informations présentes dans la base de données. La variable user du stockage de session prendra comme valeur l'Id de l'utilisateur
- LOGIN_FAILURE : il s'agit du cas où la connexion n'est pas possible ou pas permise. Rien ne se passe, on reste sur la même page.

Ci-dessous, voici les différents cas que l'on a traité :

```
import { Reducer } from "react";

const AuthReducer = (state, action) => {
  switch (action.type) {
    case "LOGIN_START":
      return {
        user: null,
        isFetching: true,
        error: false
      };
    case "LOGIN_SUCCESS":
      return {
        user: action.payload,
        isFetching: false,
        error: false
      };
    case "LOGIN_FAILURE":
      return {
        user: null,
        isFetching: false,
        error: action.payload
      };
    case "FOLLOW":
      return {
        ...state,
        user: {
          ...state.user,
          following: [...state.user.following, action.payload],
        },
      };
    case "UNFOLLOW":
      return {
        ...state,
        user: {
          ...state.user,
          following: state.user.following.filter(following=>following !== action.payload),
        },
      };
    default:
      return state
  }
}

export default AuthReducer;
```

Après avoir fait fonctionner l'API et le client indépendamment puis ensemble, on a décidé de rajouter une partie que l'on a nommée "socket", indispensable lorsque l'on veut créer une messagerie instantanée.

Nous reviendrons sur toutes ses parties en détail dans la partie suivante.

4. Réalisation technique

A. L'API

L'API (Application Programming Interface) est la partie immergée de l'iceberg, elle permet de paramétrer notre plateforme et d'effectuer la connexion entre la plateforme et la base de données. Sans elle, aucun utilisateur ne pourrait se connecter, ni voir les posts qui lui sont destinés.

Nous avons choisi d'utiliser la base de données MongoDB qui, contrairement au langage SQL classique, permet de gérer des systèmes de données complexes. On peut avoir des listes, des objets encapsulés sans avoir de soucis.

Ce fonctionnement facilite grandement le développement d'applications qui gèrent beaucoup de données, et permet de développer un système totalement dynamique, avec un système de recherche aussi performant que le SQL. De plus son système de gestion est en JSON (Javascript Object Notation) ce qui permet de le relier aux langages React utilisés par la suite.

Sans plus tarder voici les outils nécessaires à la réalisation de l'API:

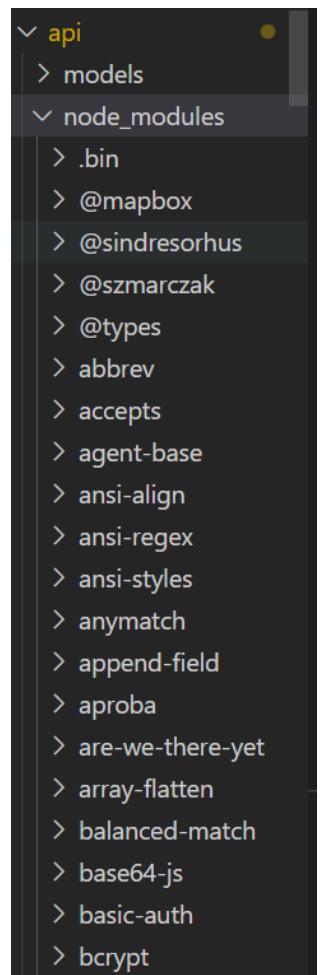
- Node.js et NPM :

NodeJS est un environnement d'exécution permettant d'utiliser le JavaScript côté serveur. Il permet de concevoir des applications en réseau performantes, telles qu'un serveur web, ou justement une API. Nous avons fait ce choix car Node.js est uniquement basé sur l'exécution de Javascript et est très populaire dans le monde des développeurs. Node.js permet également l'utilisation de nombreux frameworks dont certains nous seront utiles (Mongoose, Socket.IO...).

Npm est le gestionnaire de paquets par défaut pour l'environnement d'exécution JavaScript Node.js de Node.js, et va donc permettre d'installer tous les modules requis pour notre API.

- Mongoose :

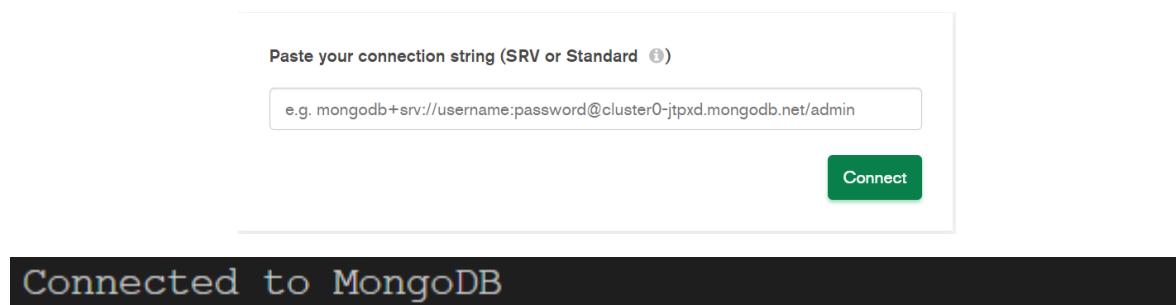
```
const moongoose = require("mongoose");
```



Mongoose est une bibliothèque de programmation JavaScript orientée objet qui crée une connexion entre MongoDB et le framework d'application, ici Node.js.

```
mongoose.connect(process.env.MONGO_URL, { useNewUrlParser: true, useUnifiedTopology: true }, () => {
  console.log("Connected to MongoDB");
})
```

Grâce à la fonction ci-dessus, la connexion à la base de données est simplifiée. Il suffit juste de rentrer l'URL de connexion de la base de données créée sur MongoDB dans un fichier .env (chargé par le process.env).



- Express :

Express est le framework actuellement le plus populaire dans Node et est la bibliothèque sous-jacente pour un grand nombre d'autres cadres applicatifs web pour Node. Express fournit des méthodes pour spécifier quelle fonction est appelée pour une méthode HTTP particulière (GET, POST, SET, etc.) et un modèle d'URL ("Route").

```
const express = require("express");
const app = express();
```

Comme dit précédemment, on définit nos paramètres d'utilisation en JSON.

```
app.use(express.json());
```

La méthode express permet aussi de démarrer le serveur sur le port 8800 et affiche un commentaire de journal dans la console. Avec le serveur en cours d'exécution, nous allons sur localhost:8800 dans notre navigateur pour voir l'exemple de réponse renvoyée :

```
app.listen(8800, () => {
  console.log("Backend server is running !")
})
```

- Dotenv :

Dotenv est un module qui charge les variables d'environnement d'un fichier .env dans process.env. Comme évoqué précédemment, cette méthode va nous servir à établir la connexion avec l'URL MongoDB.

```
const dotenv = require("dotenv");  
dotenv.config();
```

- Helmet :

Helmet est la solution préconisée pour le middleware Express.js pour protéger son application des vulnérabilités les plus courantes. Helmet est une collection de plusieurs « middlewares » ou intergiciels. Il est possible d'inclure un certain nombre d'entêtes en une fois en incluant directement Helmet :

```
app.use(helmet());
```

Par exemple, dans le cas du middleware xssFilter (inclus dans le code ci-dessus), celui-ci intervient lors d'un appel d'une URL gérée par notre serveur. Le middleware modifie alors la requête en y incluant l'en-tête X-XSS-Protection. Certains navigateurs comme IE8 bloquent des requêtes valables si cet en-tête est activé. Le middleware xssFilter gère la version du navigateur.

- Morgan :

Morgan est un middleware de niveau requête HTTP. C'est un excellent outil qui enregistre les demandes ainsi que d'autres informations en fonction de sa configuration et du préréglage utilisé. Cela s'avère très utile lors du débogage et également si vous souhaitez créer des fichiers journaux. Dans notre plateforme, Morgan servira à définir les paramètres “communs” de projet.

```
app.use(morgan("common"));
```

Par exemple, la commande ci-dessus permet de définir la date et l'heure actuelles en UTC, la version HTTP, les URL des requêtes etc...

- Multer :

Multer est un middleware node.js pour la gestion des données multipart/form-data, qui est principalement utilisé pour le téléchargement de fichiers.

Ce middleware permet donc dans notre exemple de définir les chemins et les noms de destinations des fichiers téléchargés et importés des utilisateurs.

```
const storage = multer.diskStorage({  
  destination: (req, file, cb) => {  
    cb(null, "public/images");  
  },  
  filename: (req, file, cb) => {  
    cb(null, file.originalname);  
  },  
});
```

```
const upload = multer({ storage });  
app.post("/api/upload", upload.single("file"), (req, res) => {  
  try {  
    return res.status(200).json("File uploaded successfully");  
  } catch (err) {  
    console.log(err);  
  }  
});
```

- Path : Le module path fournit des utilitaires pour travailler avec les chemins de fichiers et de répertoires.

```
app.use("/images", express.static(path.join(__dirname, "public/images")));
```

Cet exemple ci-dessus définit donc le chemin de base associé aux images de notre projet.

B. Le client

Le client ou front-end est la partie non-immégrée de l'iceberg, c'est ce que l'utilisateur voit, manipule : c'est l'interface utilisateur.

En ce qui concerne les outils, nous avons utilisé React.js, l'HTML et le CSS. React.js est un framework JavaScript extrêmement populaire parmi les développeurs. C'est un framework front open source qui a été développé par Facebook depuis 2013.

Nous avons choisi React car il est simple et facile à apprendre. En effet, le développement du web est actif car les fonctions régulières sont optimisées, cela nous permet de rentrer uniquement les données appropriées.

a) Fonctions utilisées

- useEffect : Cela fait partie des Hooks, une nouvelle mise à jour de React permettant de bénéficier d'un état local sans avoir à écrire de classe. Cela va déclencher une fonction de manière asynchrone lorsque l'état d'un composant change. Voici un exemple :

```
useEffect(() => {
  const getFriends = async () => {
    try {
      const friendList = await axios.get("/users/friends/" + user._id);
      setFriends(friendList.data);
    }
    catch (err) {
      console.log(err);
    }
  }
  getFriends();
}, [user]);
```

Cette fonction comme son nom l'indique permet de renvoyer les amis d'un utilisateur. Notre useEffect admet ici un paramètre étant l'utilisateur, en effet on veut uniquement les amis de l'utilisateur qui s'est connecté.

- Async/await : Positionner async devant une fonction signifie une chose simple
⇒ une fonction renvoie toujours une promesse
On utilise après await afin d'attendre que la promesse se réalise et renvoie son résultat.
Pour l'exemple précédent, on attend que l'API nous renvoie les amis de notre utilisateurs, on est donc obligé d'utiliser async/await.

- useState : Après le Hook d'effet, on va parler du Hook d'état. Il est très pratique et permet de bénéficier d'un état local ici aussi sans écrire de classes. Appeler useState déclare une variable d'état c'est-à-dire que les appels de fonctions ne touchent pas à cette dernière.

On passe comme argument l'état initial de la variable, et cette fonction nous renvoie une paire de valeurs, l'état et une fonction pour le modifier.

Pour l'exemple précédent :

```
const [friends, setFriends] = useState([]);
```

Initialement, notre liste d'amis est vide, logique car nous n'avons encore rien récupéré. Il faut pour cela attendre l'async/await pour le faire. Cela réalisé, on remplace l'état initial avec setFriends en lui mettant comme argument friendList.data, les amis de l'utilisateur connecté.

- Axios : il s'agit de la bibliothèque JavaScript centrale du projet. Cette dernière fonctionne comme un client HTTP. Elle permet de communiquer avec l'API en utilisant des requêtes. En résumé, elle permet de créer une passerelle entre le back-end et le front-end, entre le serveur et le client.

Toujours avec l'exemple des amis de l'utilisateur :

```
axios.get("/users/friends/" + user._id);
```

Axios prends toutes les requêtes : get, post, put, delete

Ici, il s'agit d'un get car on veut récupérer des informations dans la base de données. En argument de axios.get, on passe notre route définie dans l'API détaillée dans la partie correspondante.

- React Context

Nous allons à présent aborder la façon de se connecter et se déconnecter de Sica, il s'agit de la partie la plus complexe du code. Mais aussi la partie la plus importante car sans connexion il est impossible de pouvoir récupérer les informations sur les amis, les posts et les messages.

Pour la connexion, le travail est différent car il ne s'agit pas d'une requête ordinaire, en effet c'est une comparaison entre l'email et la mot de passe tapé et ceux qui existent déjà dans la base de données.

Pour cela on va créer un “reducer” : il s’agit d’une fonction qui est chargée de construire un nouvel état à chaque action.

Voici les différentes actions que l’on a pour le moment sur notre plateforme :

```
export const LoginStart = (userCredentials) =>({
  type: "LOGIN_START",
});

export const LoginSuccess = (user) =>({
  type: "LOGIN_SUCCESS",
  payload: user,
});

export const LoginFailure = (error) =>({
  type: "LOGIN_FAILURE",
  payload: error,
});

export const Follow = (userId) =>({
  type: "FOLLOW",
  payload: userId,
});

export const Unfollow = (userId) =>([
  type: "UNFOLLOW",
  payload: userId,
]);
```

Voici le reducer pour les actions précédentes :

```

import { Reducer } from "react";

const AuthReducer = (state, action) => {
  switch (action.type) {
    case "LOGIN_START":
      return {
        user: null,
        isFetching: true,
        error: false
      };
    case "LOGIN_SUCCESS":
      return {
        user: action.payload,
        isFetching: false,
        error: false
      };
    case "LOGIN_FAILURE":
      return {
        user: null,
        isFetching: false,
        error: action.payload
      };
    case "FOLLOW":
      return {
        ...state,
        user: {
          ...state.user,
          following: [...state.user.following, action.payload],
        },
      };
    case "UNFOLLOW":
      return {
        ...state,
        user: {
          ...state.user,
          following: state.user.following.filter(following => following !== action.payload),
        },
      };
    default:
      return state
  }
}

export default AuthReducer;

```

Après avoir réalisé le reducer et les actions il faut utiliser un autre outil très pratique de React. Il s'agit de React Context. Cet outil permet de partager les propriétés à un composant. Le fait de créer un context permet de récupérer simplement nos datas sans avoir à tout passer manuellement. C'est assez analogue à la notion d'héritage que l'on retrouve dans la programmation orientée objet. En effet, on englobe le composant “parent” le plus haut dans l’arborescence de composants avec le pattern Provider. Tous les composants “enfants” pourront alors se connecter au Provider et ainsi accéder aux datas sans passer par tous les composants intermédiaires.

Voici le context créé pour l'authentification :

```

import { createContext, useEffect, useReducer } from "react";
import AuthReducer from "./AuthReducer";

const INITIAL_STATE = {
  user: JSON.parse(sessionStorage.getItem("user")) || null,
  isFetching: false,
  error: false
};

export const AuthContext = createContext(INITIAL_STATE);

export const AuthContextProvider = ({ children }) => {
  const [state, dispatch] = useReducer(AuthReducer, INITIAL_STATE);

  useEffect(() => {
    sessionStorage.setItem("user", JSON.stringify(state.user));
  }, [state.user])

  return (
    <AuthContext.Provider value={{
      user: state.user,
      isFetching: state.isFetching,
      error: state.error,
      dispatch
    }}>
      {children}
    </AuthContext.Provider>
  )
}

```

INITIAL_STATE : si c'est une nouvelle session alors l'utilisateur est null donc on le renvoie sur la page par défaut c'est-à-dire sur le register. Sinon, le fait d'avoir sessionStorage.getItem("user") nous permet de ne pas se reconnecter à chaque fois que l'on relance la page.

La ligne suivante sert à créer notre context à partir de l'état initial. Puis, l'on écrit notre Provider afin de réaliser des mises à jour sur nos données. En effet, si l'on se connecte pour la première fois sur une session, sessionStorage.setItem permettra d'enregistrer les informations de ce nouvel utilisateur afin de créer un nouvel état.

Cela n'est toujours pas suffisant pour réussir à se connecter à la plateforme car maintenant il faut encore créer la logique pour passer la page Login à la page d'accueil. On réalise donc une fonction loginCall qui comme son nom l'indique permet d'appeler l'API.

```

import axios from "axios";

export const loginCall = async (userCredential, dispatch) => {
  dispatch({type: "LOGIN_START"});
  try{
    const res = await axios.post("auth/login", userCredential);
    dispatch({type: "LOGIN_SUCCESS", payload: res.data});
  }
  catch(err){
    dispatch({type: "LOGIN_FAILURE", payload: err})
  }
}

```

Cette fonction permet de déterminer si les données transmises sont en adéquation avec ce qu'il y a dans MongoDb, si c'est vrai on est dans l'état LOGIN_SUCCESS et passe à la page d'accueil sinon on reste sur la page de connexion avec LOGIN_FAILURE.

On utilise cette fonction dans notre fichier login.jsx avec comme userCredential le mail et le mot de passe entrés dans les inputs, le tout dans une fonction handleClick.

```

const handleClick = (e) => {
  e.preventDefault();
  loginCall({ email: email.current.value, password: password.current.value }, dispatch);
}

```

Puis pour finir on appelle cette fonction pour valider le formulaire :

```

<form className="loginBox" onSubmit={handleClick}>
  <input placeholder="Email" type="email" className="loginInput" ref={email} required />
  <input placeholder="Password" type="password" className="loginInput" ref={password} minLength="4" required />
  <button className="loginButton"> {isFetching ? <CircularProgress color="white" size="20px" /> : "Log In"}</button>
  <span className="loginForgot">Forgot Password</span>
  <Link to="/register" style={{textDecoration:"none"}}>
    <button className="loginRegisterButton">{isFetching ? <CircularProgress color="white" size="20px" /> : "Create a new account"}</button>
  </Link>
</form>

```

Nous avons importé pas mal de composants de material-ui comme CircularProgress pour rendre la plateforme plus esthétique.

Pour finir avec les outils utilisés sur la partie clients nous allons étudier les 2 derniers Hooks utiles :

- useRef : il va renvoyer un objet modifiable qui n'impacte pas le cycle de vie du composant. On va référer un ou des composants :

```

const email = useRef();
const password = useRef();

```

Ici email et password sont des objets avec la propriété current. L'argument dans le useRef est optionnel.

- useContext : Il s'agit d'un Hook qui permet d'accéder aux informations partagées.

```
const { user } = useContext(AuthContext);
```

Plus simplement dans notre cas d'utilisation, c'est le composant qui va nous permettre de récupérer toutes les informations concernant l'utilisateur (Id, username, post, projet,...).

C. Socket

Socket.io est une bibliothèque JavaScript pilotée par les évènements pour les applications web en temps réel. Elle permet une communication à faible latence, bidirectionnelle et basée sur les évènements entre un client et un serveur.

Socket.io repose sur le protocole WebSocket et offre des garanties supplémentaires telles qu'un mode dégradé en HTTP long-polling ou la reconnexion automatique.

Dans le fichier index.js, on va écrire toutes les fonctions dont on va avoir besoin pour envoyer et recevoir un message, ouvrir une conversation ou en supprimer une.

Voici les fonctions pour ajouter, supprimer et obtenir un utilisateur :

```
const addUser = (userId, socketId) => {
  !users.some(user => user.userId === userId) && users.push({ userId, socketId });
}

const removeUser = (socketId) => {
  users = users.filter((user) => user.socketId !== socketId);
}

const getUser = (userId)=>{
  return users.find(user=>userId === user.userId)
}
```

AddUser permet de rechercher un utilisateur avec some puis de l'ajouter avec push. RemoveUser avec filter permet de remplacer une liste par une autre mais sans l'utilisateur que l'on veut supprimer.

GetUser va nous renvoyer l'utilisateur correspondant à l'identifiant que l'on cherche.

Lorsqu'il y a une connection au socket :

```
io.on("connection", (socket) => [
  //when there is a connection
  console.log("a user connected");

  // take userId and socketId from user
  socket.on("addUser", userId => {
    addUser(userId, socket.id);
    io.emit("getUsers", users);
  })

  //send and get message
  socket.on("sendMessage", ({senderId, receiverId, text})=>{
    const user = getUser(receiverId);
    io.to(user.socketId).emit("getMessage", {
      senderId, text
    })
  });
]);
```

Socket.on est un outil créant une connexion WebSocket. Donc, chaque utilisateur se connectant à "localhost:3000/" déclenche une connexion via WebSocket avec notre app Node.js, il va s'afficher dans la console "a user connected". Le ".on" permet de recevoir une réponse.

Socket.emit va permettre comme son nom l'indique d'émettre un message.

5. Organisation et bilan du projet

A) Organisation du projet et du binôme.

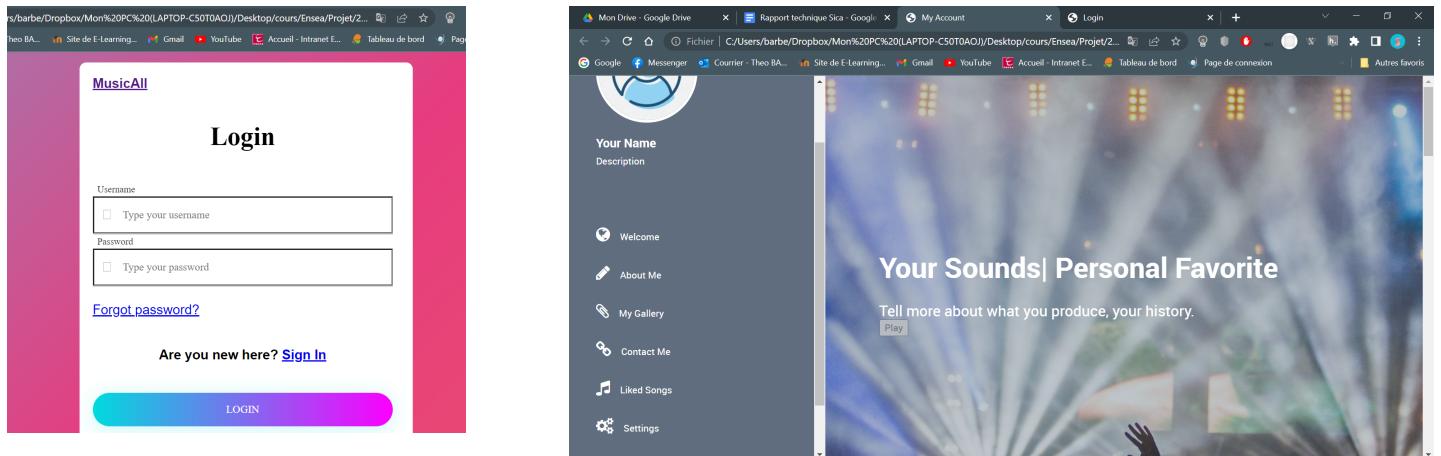
Dès le début du projet, nous avions conscience de la difficulté technique et du temps qu'il nous faudrait afin de réaliser un prototype opérationnel d'ici la fin de l'année et nous avons cherché à jouer sur nos compétences personnelles que nous avions partiellement acquise lors de notre stage d'été, Jean-François maîtrisant le langage PHP, JavaScript et Théo maîtrisant le HTML5 et le CSS.

On a donc conçu une première méthode de travail, Jean-François allait donc gérer la gestion de la plateforme à travers des fonctions PHP/JS et une base de données MySQL, et moi-même gérant le dossier client en assurant le contenu des pages et leur design.

On s'est très vite rendu compte que le projet manquait fortement de dynamisme, chaque option et chaque fonction nécessitait de rafraîchir la page, ce qui demandait un temps de réponse trop long pour toutes les fonctionnalités que l'on voulait développer. On s'est donc tourner vers une solution plus moderne, et plus dynamique grâce à l'utilisation de frameworks, et en créant l'application en partie grâce à React et Node.js. On a donc gardé nos rôles respectifs, Jean-François développant les fonctions et les composants Reacts, et Théo s'occupant du design de la plateforme, afin d'avoir une apparence claire et précise.

B) Résultats

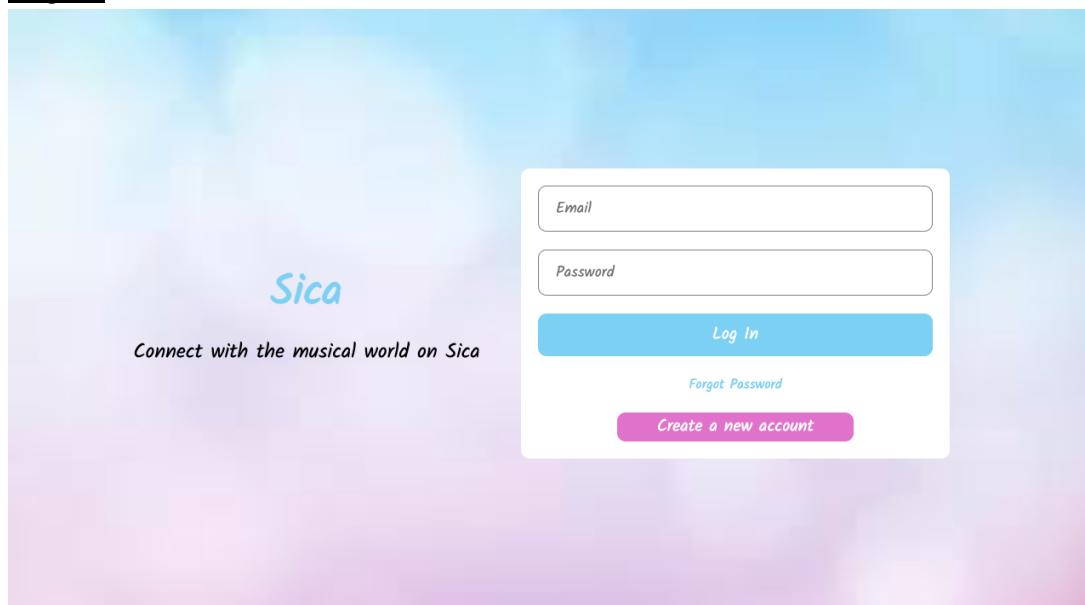
Voici les pages de la première version de la plateforme, avec la connexion en première et enfin le squelette du compte :



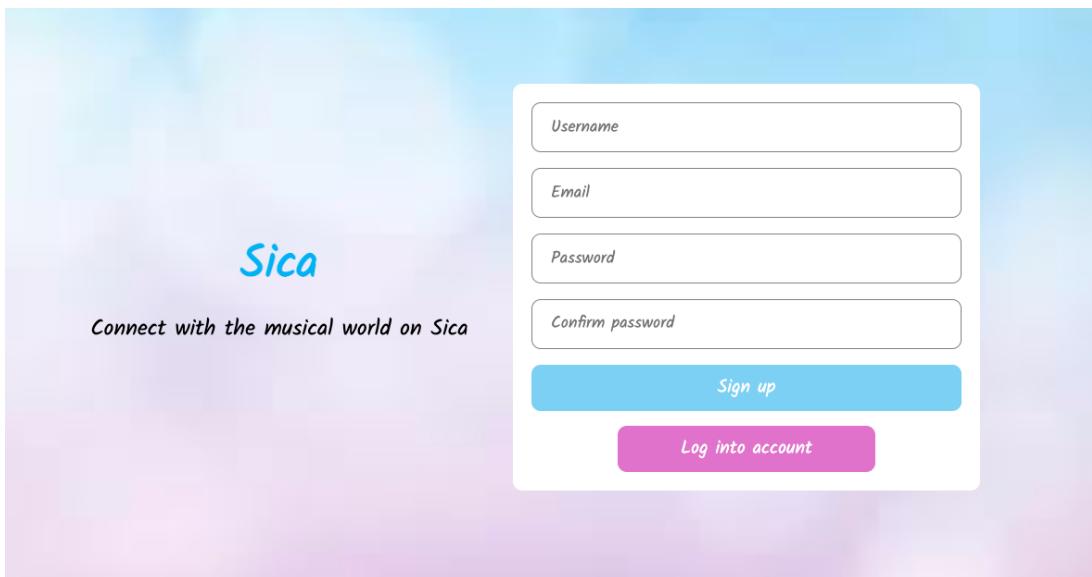
Après cette première version en web 1.0, nous avons appris les langages d'un web de niveau plus élevé : un web plus dynamique et plus accessible.

Voici une vue d'ensemble des pages de Sica :

Login :



Register :



Home :

The image displays the Sica home feed interface. On the left, a sidebar lists navigation options: Feed, Groups, Projects, Questions, Events, Courses, and a "Show more" button. Below this is a "User Friends" section. The main area shows a timeline of posts from users Jeff and Catena. Each post includes a video thumbnail, a timestamp (e.g., "2 weeks ago"), a play button, a progress bar, and a "Share" button. Below each video is a "1 people liked it" button. To the right of the posts is a promotional banner for "TOMORROWLAND ALPE D'HUEZ" with the text "MARCH 19-26 2022 ALPE D'HUEZ - FRANCE".

Projet :

La page projet est la même que la page d'accueil sauf que l'on affiche les projets à la place des posts.

Sica

Search for

Homepage Timeline

Feed Groups Projects Questions Events Courses Show more

User Friends

What's your project Jeff? Share

test 5 days ago Hello les gars ! Qui pour faire de l'électro ?

Jeff 5 days ago Freestyle Qui pour un freestyle ?

Jeff 2 weeks ago Association pour un projet de Jazz Recherche de 6 personnes disponibles pour un grand travail autour de ce grand genre ! Envoyez un message en MP

TOMORROWLAND WINTER
MARCH 19-26 2022 ALPE D'HUEZ - FRANCE

Profil :

Sica

Search for

Homepage Timeline

Feed Groups Projects Questions Events Courses Show more

User Friends

What's in your mind Jeff? Share

Jeff 2 weeks ago

▶ 0:00 / 4:19

User information

City: Bastia
From: Niolu

Description: I make electronic music, and i love progressive house and future bass

Update your profile

Chat :

The screenshot shows a messaging interface with a blue header bar. On the left is the user's profile picture and name "Sica". To the right are search, homepage, timeline, and notifications icons. Below the header is a search bar labeled "Search for friends". A list of contacts follows, with "Clara" and "test" visible. The main area displays a conversation between Clara and test. Clara sent a message "Salut jeff c'est le test" 5 days ago. test responded with "weu weu ceci est une descente de police" 5 days ago. At the bottom is a text input field with placeholder "Write something..." and a "Send" button.

Open a conversation to start a chat

Salut jeff c'est le test

5 days ago

weu weu ceci est une descente de police

5 days ago

Write something...

Send

C) Améliorations possibles

Tout d'abord il y a plein de détails à terminer, afin que toutes les options de la plateforme soient complètes, mais cela demande beaucoup de temps (Mot de passe oublié, mise à jour des informations personnelles, design des spectres des musiques, espace de groupes, événements, cours etc....).

Beaucoup de ces détails sont déjà écrits dans le back-end mais pas dans le front-end comme le mot de passe oublié, les modifications de profil, la suppression de publications, de messages,....

De plus, nous avons pensé à insérer un système de NFT (avec un système d'enchères) , afin de renforcer l'unicité des musiques et encourager la création musicale.

Enfin, on pourrait créer un algorithme (potentiellement une intelligence artificielle) qui permettrait de rediriger les utilisateurs vers de nouvelles personnes en adéquation avec leurs goûts musicaux, et un système de classification de sons très affinés.

Vous aurez donc compris, un projet de ce type n'est jamais réellement fini, de nouvelles idées sont toujours envisageables.

6. Conclusion

Tout d'abord nous sommes très ravis d'avoir travaillé dans ce projet, qui nous a permis de développer de nouvelles compétences très utiles, et de comprendre l'architecture complète d'une plateforme sociale. De plus, nous avons réalisé et pu constater la puissance, le dynamisme et la clarté qu'apportent les frameworks. Les compétences acquises cet année grâce à notre projet sont un atout extrêmement important pour notre future carrière. Le développement full-stack étant de plus en recherché dans les entreprises. De plus, cette année nous a permis d'améliorer nos capacités à apprendre de nouvelles capacités par nous-mêmes.

Enfin, nous avons adoré travailler dans ce projet, nos personnalités se complétant afin d'arriver à travailler toujours dans l'harmonie, et efficacement. Nous sommes assez fiers de nos résultats actuels et nous pensons peut-être pouvoir terminer le projet dans le futur, de manière personnelle ou même professionnelle.