

Projet HDMeet



Zakaria BELKACEMI
Formation Développeur Web
02/02/2024

Projet HDMeet (React, Node)



COMMENCER

ADMINISTRATION

HDM Meet

Table des matières

	Page #
1- Competences couvertes par le projet	3
2- Résumé du projet	3
3-Maquettage de l'application	4
4- Cahier des charges.....	5
5- Specifications techniques.....	6
6- La base de donnees et le MCD	7
7- Réalisation du projet	9
7.1- Vidéoconférence	9
7.2-Mécanisme du chat.....	23
7.3-Administration	25
7.4- CRUD	42
8- Utilisation du programme	45
9-Tests.....	49
10- Veille technique.....	50
11- Bilan.....	51

1. Compétences couvertes par le projet

- Les compétences couvertes sont :
 - Maquetter une application
 - Réaliser une interface utilisateur web statique et adaptable
 - Développer une interface utilisateur web dynamique
 - Créer une base de données
 - Développer les composants d'accès aux données
 - Développer la partie back-end d'une application web ou web mobile
 - Gérer l'authentification des utilisateurs et attribution de ROLES
 - Etablir une connexion websocket entre les peers

2. Résumé du Projet

HDM Network est une organisation à but non lucratif qui vise à briser la fracture numérique en offrant des opportunités pour les personnes qui cherchent à développer leurs compétences dans le monde numérique.

L'objectif du projet est de développer une application de vidéoconférence en utilisant les technologies React pour le front-end et Node/Express pour le back-end, afin de remplacer l'utilisation de la version gratuite de Zoom au sein d'HDM NETWORKS. Cette transition est motivée par la limitation de 45 minutes imposée par Zoom sur les sessions gratuites.

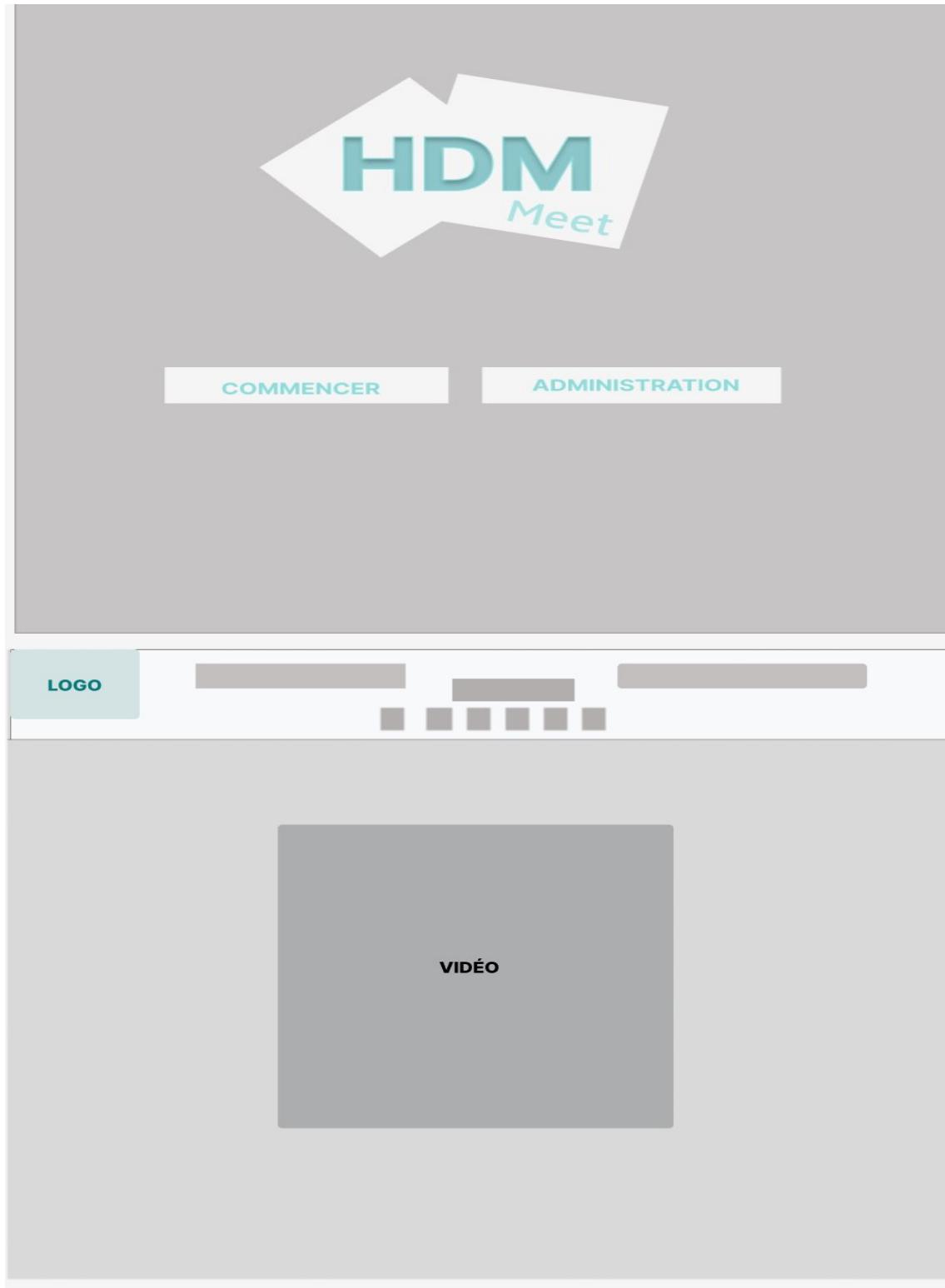
Le processus de développement a débuté par une analyse approfondie du cahier des charges, permettant de définir les besoins spécifiques de l'entreprise en matière de vidéoconférence. En se basant sur ces exigences, un Modèle Conceptuel de Données (MCD) a été élaboré pour comprendre la structure de la base de données MySQL qui sera utilisée.

Le front-end de l'application sera construit en utilisant React, assurant une interface utilisateur moderne et réactive. Du côté du back-end, Node.js avec Express sera employé pour créer un serveur robuste et efficace, permettant la gestion des sessions de vidéoconférence.

Le développement de cette application de vidéoconférence a suivi une approche Agile pour assurer une gestion flexible et itérative du projet. Une partie authentification a également été mise en place, au même titre qu'un panel Admin afin de gérer les droits des utilisateurs et les accès à l'application.

Les technologies Socket.IO et WebRTC ont été cruciales, permettant l'établissement de connexions en temps réel entre les pairs. Socket.IO a facilité la communication bidirectionnelle en temps réel, tandis que WebRTC a été essentiel pour la transmission de flux audio et vidéo, garantissant une expérience de vidéoconférence fluide et efficace. Cette combinaison de technologies modernes et bien établies aboutit à une application performante et adaptée aux besoins spécifiques de communication de l'entreprise.

3. MAQUETTAGE DE L'APPLICATION



4. CAHIER DES CHARGES

HDM souhaite se détacher de l'application Zoom pour ses vidéo conférence à cause de la contrainte suivante : les utilisateurs sont expulsés de la conférence au bout de 45 minutes. Cela est dû au fait qu'ils utilisent la version gratuite de Zoom. C'est dans ce contexte que HDM m'a demandé la création d'une application de vidéo conférence. Ainsi, les utilisateurs ne seraient alors plus expulsés durant les conférences.

Les utilisateurs doivent pouvoir s'identifier à l'aide de leurs adresse email. Une adresse email est lié à un rôle, un rôle est lié à un email.

A partir du moment où une adresse email est présente dans la base de données, cette adresse email a accès à l'application. Seuls les emails avec le rôle ADMIN ont la possibilité d'ajouter un nouvel accès ou de changer un rôle.

L'application doit être facile d'utilisation et le plus proche possible de l'application Zoom en termes de fonctionnalités (chat, partage d'écran, mute, et camera on/off)

Le développement des pages web dynamiques en utilisant les technologies MYSQL pour la gestion de la base de données, Réact en front-end et Node.js / Express en back-end.

Développement des CRUD à partir d'une base de données.

Le site doit être adaptatif, c'est-à-dire qu'il doit être compatible avec les appareils mobiles et les ordinateurs de bureau.

Il doit être sécurisé, avec des mécanismes de protection contre les vulnérabilités telles que les injections SQL et les failles XSS.

Les données sensibles des utilisateurs au même titre que les identifiants de connexion à la base de données doivent être stockés de manière sécurisée.

5. SPECIFICATIONS TECHNIQUES

HTML :

Afin de créer et de représenter le contenu d'une page web et sa structure.

CSS :

Permet de créer des pages web à l'apparence soignée.

REACT :

React est une bibliothèque Javascript open-source permettant la construction d'interfaces utilisateur interactives et réactives.

SOCKET.IO :

Socket.IO est une bibliothèque javascript qui permet la mise en œuvre de la communication en temps réel bidirectionnelle entre le serveur et le client, facilitant le développement d'application web interactives

ANTD :

Ant Design (antd) est une bibliothèque de conception de composants React qui offre un ensemble complet de composants d'interface utilisateur (UI) préconçus et stylisés. J'ai principalement utilisé cette bibliothèque pour mes alertes messages.

MATERIAL-UI :

Material-UI est une bibliothèque de composants React basée sur les principes du design Material, créée par Google. J'ai utilisé m-ui pour mes boutons et mes inputs.

WebRTC:

WebRTC (Web Real Time communication) est un ensemble de normes et de protocoles permettant la communication en temps réel directement entre navigateurs web (c'est pourquoi c'est utilisé seulement côté front) sans nécessiter de plugins ou d'installations tierces.

MYSQL :

Le système de gestion de base de données MySQL est un logiciel permettant d'introduire des données, de mettre à jour, de supprimer et d'accéder aux données stockées dans une base de données. Il utilise le langage SQL

EXPRESS :

Express est un framework web pour Node.js, conçu pour simplifier le processus de création d'applications web et d'API._

NodeJS :

Node.js permet d'exécuter du code JavaScript côté serveur, alors que traditionnellement Javascript était principalement utilisé côté client dans les navigateurs web. Cela unifie le langage de programmation entre le côté client et le côté serveur.

6. La base de données et le MCD

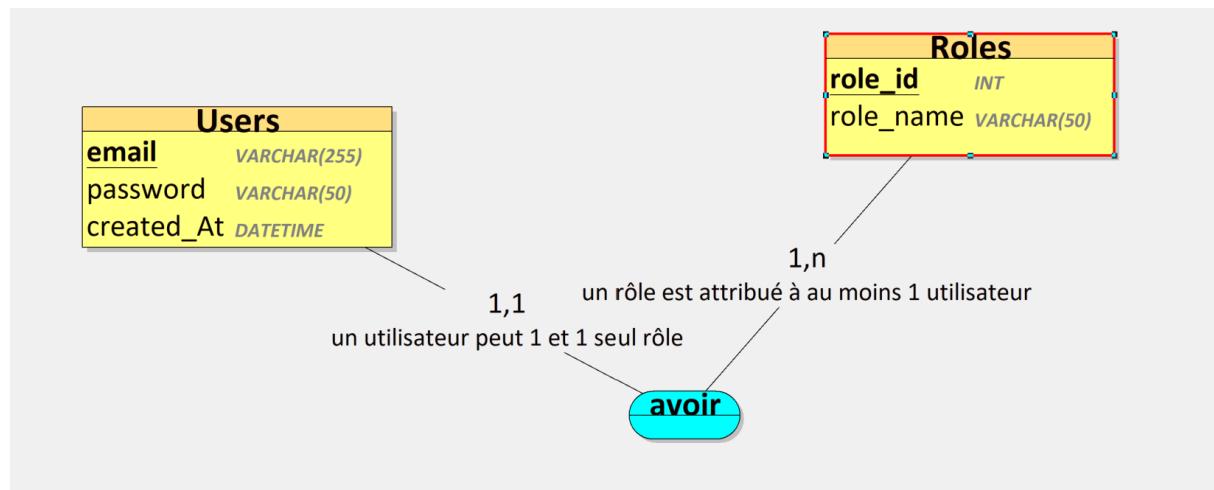
La base de données est simple. Elle comporte qu'une seule table (users).

Il a été déterminé par le tech lead qu'une seule table serait nécessaire. Du coup je n'ai pas mis en œuvre de liaison entre les tables.

La table dans l'application actuelle est telle que :

Users	
email	<i>VARCHAR(255)</i>
password	<i>VARCHAR(50)</i>
role	<i>ENUM ('ADMIN','USER')</i>
created_at	<i>DATETIME</i>

De ce fait, pour vous parler de cardinalité, voici le MCD qui illustre ce que j'aurai fait :



Cela signifie que chaque utilisateur peut être associé à un et un seul rôle, et chaque rôle peut appartenir à 1 ou plusieurs utilisateurs.

La connexion à la base de données se fait côté serveur (server.js).

```
const mysql = require('mysql2');

const connection = mysql.createConnection({
    host: process.env.DB_HOST,
    user: process.env.DB_USER,
    password: process.env.DB_PASSWORD,
    database: process.env.DB_DATABASE
});

connection.connect((err) => {
    if (err) {
        console.error('Erreur de connexion à la base de données MySQL :', err);
    } else {
        console.log('Connecté à la base de données hdmeet');
    }
});
```

J'importe le module **mysql2** dans le script. Ce module fournit des fonctionnalités pour interagir avec une base de données MySQL depuis une application Node.js.

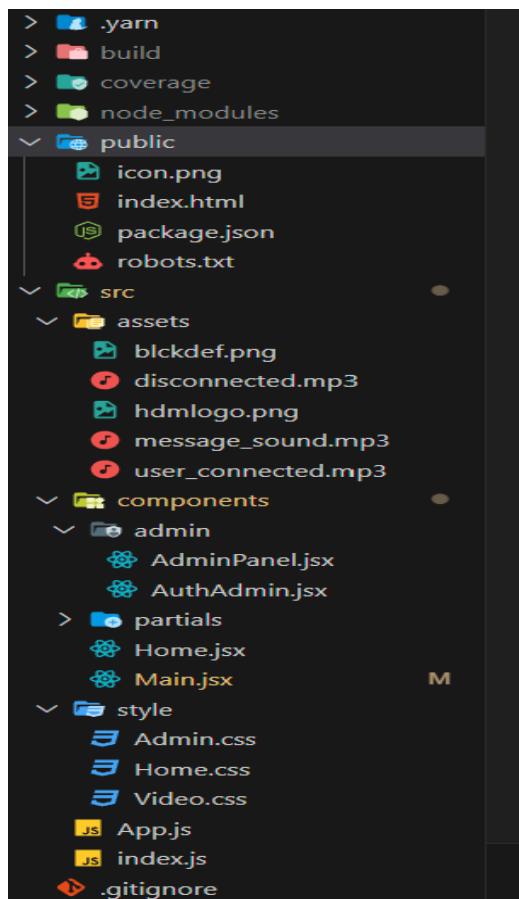
Ensuite, une connexion à la base de données est établie en utilisant la méthode **mysql.createConnection**. Les informations nécessaires pour établir la connexion, telles que l'hôte, l'utilisateur, le mot de passe et le nom de la base de données, sont récupérées à partir des variables d'environnement (**process.env**), ce qui permet une configuration flexible sans révéler les informations sensibles directement dans le code.

La connexion est ensuite établie avec la base de données en appelant la méthode **connect** sur l'objet de connexion. Cette méthode prend un callback en argument, qui est appelé une fois la connexion établie ou en cas d'erreur.

Dans le callback, on vérifie si une erreur est survenue lors de la connexion. Si c'est le cas, un message d'erreur est affiché dans la console, indiquant qu'il y a eu une erreur lors de la connexion à la base de données MySQL. Sinon, un message de confirmation est affiché, indiquant que la connexion à la base de données "hdmeet" a été établie avec succès.

7. REALISATION DU PROJET HDM Meet

Vidéo conférence :



L'arborescence se compose principalement d'un dossier src qui contient un dossier components qui lui-même contient un dossier 'admin' (composants liés à l'administration) et un sous dossier 'partials' (pour la sideBar et la controlBar incorporés dans Main) à la racine du dossier components il y a les principaux composants Home et Main.

App.js :

```

import React, { Component } from 'react';
import Main from './components/Main.jsx';
import Home from './components/Home.jsx';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import AuthAdmin from './components/admin/AuthAdmin.jsx';
import AdminPanel from './components/admin/AdminPanel.jsx';
import { AuthProvider } from './context/authContext.js';
import ProtectedRoute from './components/protectedRoute.jsx';

class App extends Component {

    render() {
        return (
            <AuthProvider>
                <div>
                    <Router>
                        <Routes>
                            <Route path="/" element={<Home />} />
                            <Route path="/authAdmin" element={<AuthAdmin />} />
                            <Route path="/adminPanel/*" element={<ProtectedRoute component={AdminPanel} />} />
                            <Route path="/:url" element={<Main />} />
                        </Routes>
                    </Router>
                </div>
            </AuthProvider>
        );
    }

    export default App;
}

```

Le fichier **app.js** constitue le point d'entrée principal de notre application React. Il orchestre le routage et la navigation au sein de l'application en utilisant la bibliothèque **react-router-dom**.

- **Composants Importés :** J'importe les composants principaux, tels que **Main**, **Home**, **AuthAdmin**, et **AdminPanel**, qui représentent les différentes sections de notre application.
- **Configuration du Routage :** En utilisant **BrowserRouter**, **Routes**, et **Route** de **react-router-dom**, nous définissons les différentes routes de notre application. Chaque route est associée à un chemin spécifique et rend un composant particulier.
- **Routes Définies :**
 - La route "/" est associée au composant **Home**.
 - La route "/AuthAdmin" est associée au composant **AuthAdmin**.
 - La route "/adminPanel" est associée au composant **AdminPanel**.
 - La route "/:url" est dynamique, associant le chemin à un composant **Main**.

Ce fichier centralise la structure de navigation de notre application, facilitant la compréhension et la gestion des différentes sections. Il offre une vision claire de la logique de routage qui guide le rendu des composants en fonction des chemins d'URL spécifiés.

A noter que je « wrappe » tous mes composants grâce à mon **AuthProvider** (context), ce qui me permet de gérer l'authentification à travers l'application.

Home.jsx :

```
import React from 'react';
import { Button } from '@material-ui/core';
import logo from '../assets/hdmlogo.png';
import './style/Home.css';
import { Link } from 'react-router-dom';
import { useAuth } from '../context/authContext';

const Home = () => {
  const { isAuthenticated } = useAuth();

  const createRoom = () => {
    const newRoom = Math.random().toString(36).substring(3, 25);
    window.location.href = `/ ${newRoom}`;
  }

  return (
    <div className="container2">
      <div>
        <img className='logo' src={logo} alt="" /> <br />
        <Button className='startBtn' variant="contained" color="primary" onClick={createRoom}>Commencer</Button>

        {isAuthenticated ? (
          <Link to="/adminPanel">
            <Button className='startBtn' variant="contained" color="primary">Administration</Button>
          </Link>
        ) : (
          <Link to="/authAdmin">
            <Button className='startBtn' variant="contained" color="primary">Administration</Button>
          </Link>
        )}
      </div>
    </div>
  );
}

export default Home;
```

Le fichier **Home.jsx** définit le composant **Home**, qui représente la page d'accueil.

Imports : J'importe les dépendances nécessaires, y compris le composant **Button** de **@material-ui/core**, une image de logo, des styles CSS à partir de **Home.css**, et le composant **Link** de **react-router-dom**.

- **Méthode createRoom :** Lorsque l'utilisateur clique sur le bouton "Commencer", cette méthode génère un identifiant de salle unique de manière aléatoire. Elle utilise un nombre aléatoire converti en une chaîne de caractères en base 36, puis extrait une sous-chaîne de cette valeur pour obtenir un identifiant unique. Ensuite, l'utilisateur est redirigé vers la nouvelle salle en utilisant **window.location.href**.
- **Rendu :** `return()` génère l'interface utilisateur de la page d'accueil, comprenant le logo, un bouton "Commencer" qui déclenche la création d'une nouvelle salle, et un bouton "Administration" qui redirige vers la page d'authentification pour les administrateurs ou directement dans le panel administrateur si l'utilisateur est déjà connecté.

Main.jsx :

Le composant **Main.jsx** gère la logique principale de l'application.

Depuis la page home, lorsque l'utilisateur a cliqué sur « commencer », la room est générée et il atterrit dans une partie d'authentification

```
    <input
      placeholder="Nom d'utilisateur"
      type="text"
      name="text"
      onChange={(e) => this.handleUsername(e)}
      required
      style={{
        backgroundColor: "white",
        borderRadius: "5px",
        margin: "10px",
      }}
    />
    <Button
      className="startBtn"
      type="submit"
      variant="contained"
      color="primary"
    >
      Se connecter
    </Button>
  </form>
</div>
<div>
  <video
    id="myVideo"
    ref={this.myVideo}
    autoPlay
    muted
    style={{
      objectFit: "fill",
      width: "100",
      height: "30%",
      borderRadius: "25px",
    }}
    onClick={this.handleVideoClick}
  ></video>
</div>
</div>
render() {
  return (
    <div>
      {this.state.askForUsername ? (
        <div>
          {this.state.loadingCamera && (
            <div className="spinner">
              {" "}
              <Flex align="center" gap="middle">
                <Spin size="large" />
                Chargement...
              </Flex>
            </div>
          )}
          <Link to="/">
            <img
              className="logo"
              src={logo}
              alt=""
              style={{
                width: "150px",
                position: "absolute",
                top: "0",
                left: "0",
              }}
            />
          </Link>
          <br />
          <br />
        </div>
      ) : (
        <div className="askUsername">
          <form onSubmit={this.handleSubmit}>
            <input
              type="email"
              placeholder="Votre email"
              name="email"
              autoComplete="email"
              onChange={(e) => this.handleEmail(e)}
              required
              style={{
                backgroundColor: "white",
              }}
            />
          </form>
        </div>
      )}
    </div>
  )
}
```

- **Condition askForUsername :** La section est conditionnée par `this.state.askForUsername`, qui contrôle si l'application doit demander à l'utilisateur de fournir son email et son nom d'utilisateur.
- **Formulaire de Connexion :** Un formulaire est render avec deux champs : un champ d'email et un champ de nom d'utilisateur. Ces champs sont stylisés et obligatoires (`required`). Les valeurs saisies sont gérées par les méthodes `handleEmail` et `handleUsername`.
- **Bouton "Se connecter" :** Un bouton "Se connecter" déclenche la méthode `handleSubmit` lorsqu'il est cliqué.
- **Vidéo en Arrière-plan :** Une vidéo en arrière-plan est rendue, affichant le flux vidéo de l'utilisateur. La vidéo est également cliquable pour zoomer, déclenchant la méthode `handleVideoClick` qui utilise l'API fullscreen des navigateurs.
Vérification saisie : La méthode `handleSubmit` est déclenchée lors de la soumission du formulaire où l'utilisateur entre son email et son nom d'utilisateur.
event.preventDefault() : Cette ligne empêche le comportement par défaut du formulaire, évitant ainsi le recharge de la page lors de la soumission.

Petit bonus : j'ai ajouté une petite roue de chargement pendant que l'api webRTC vérifie les permissions de caméra/audio (on y reviendra un peu plus tard) une fois les permissions obtenues, la state affichant la roue de chargement passera à false pour ensuite laisser place à la caméra de l'utilisateur.

```
handleSubmit = (event) => {
  event.preventDefault()

  const { currentUserEmail, authorizedUsers } = this.state

  let isAuthorized = false

  for (let i = 0; i < authorizedUsers.length; i++) {
    if (authorizedUsers[i].email === currentUserEmail) {
      isAuthorized = true
      break
    }
  }

  if (isAuthorized) {
    this.connect()
  } else {
    message.error(
      "Hop hop hop, vous n'êtes pas autorisé à entrer ici, allez zou!"
    )
  }
}
```

- **Extraction des Variables** : Les variables `currentUserEmail` et `authorizedUsers` sont extraites de l'état actuel (`this.state`).
- **Vérification de l'Autorisation** : La méthode itère à travers la liste des utilisateurs autorisés (`authorizedUsers`) pour vérifier si l'email actuel (`currentUserEmail`) est présent parmi les utilisateurs autorisés. Si une correspondance est trouvée, la variable `isAuthorized` est définie sur `true`.
- **Connexion Autorisée** : Si l'utilisateur est autorisé (si `isAuthorized` est `true`), la méthode `connect()` est appelée, impliquant probablement la transition vers la phase suivante de la vidéoconférence ou l'activation de fonctionnalités spécifiques.
- **Refus d'Accès** : Si l'utilisateur n'est pas autorisé, un message d'erreur est affiché à l'aide de la bibliothèque "antd" (`message.error`). Cela informe l'utilisateur qu'il n'a pas l'autorisation d'accéder à la vidéoconférence.

Tout cela garantie que seuls les **utilisateurs autorisés** peuvent accéder à la vidéoconférence. La vérification se fait en comparant l'email actuel avec la liste des utilisateurs autorisés, renforçant ainsi la sécurité et la gestion des accès à l'application.

Lorsque l'utilisateur entre ses informations et clique sur "Se connecter", la logique de l'application passe à la deuxième partie où la vidéoconférence est activée et d'autres fonctionnalités sont disponibles.

Déterminer si un utilisateur est autorisé ou non implique un call API afin d'alimenter une « state » qui sera ensuite traitée dans handleSubmit.

```

class Main extends Component {
  constructor(props) {
    super(props)
    this.myVideo = React.createRef()
    this.iceCandidatesQueue = {}
    this.videoAvailable = false
    this.audioAvailable = false

    this.state = {
      video: false,
      audio: false,
      screen: false,
      showModal: false,
      screenAvailable: false,
      messages: [],
      message: '',
      newmessages: 0,
      askForUsername: true,
      username: '',
      usernames: {},
      isSidebarOpen: false,
      playUserConnectedSound: false,
      requestingSpeech: false,
      speechRequestMessage: '',
      password: '',
      authorizedUsers: [], ⚡ [←
      connectedEmails: [],
      currentUserEmail: '',
      isAdmin: false,
      loadingCamera: true,
      isSpeakingStates: {}
    }

    axios
      .get("http://localhost:4001/users")
      .then((response) => {
        this.setState({ authorizedUsers: response.data })
      })
      .catch((error) => {
        console.error(
          "Erreur lors de la récupération des utilisateurs :",
          error
        )
      })
  }
}

```

- Une requête HTTP GET est effectuée sur mon endpoint :"<http://localhost:4001/users>" pour obtenir la liste des utilisateurs autorisés.
- En cas de succès (**then**), la réponse est utilisée pour mettre à jour l'état **authorizedUsers** avec les données récupérées depuis l'API.
- En cas d'erreur (**catch**), un message d'erreur est affiché dans la console.

A noter que cet événement se produit dès l'initialisation du composant étant donné que mon call API est situé dans le constructeur du composant (j'aurai également pu mettre ce call dans un componentDidMount certes...).

Concernant la partie back-end de mon endpoint « users », il s'agit d'une requête SQL à ma base de données afin de récupérer tout de la table « users ».

```
app.get('/users', (req, res) => {
  console.log('Requête vers /users reçue.');

  connection.query('SELECT * FROM users ORDER BY created_At DESC', (err, results) => {
    if (err) {
      res.status(500).json({ error: 'Erreur lors de la récupération des utilisateurs' });
    } else {
      const sanitizedEmails = results.map(user => sanitizeString(user.email));
      res.json(sanitizedEmails);
    }
  });
});
```

1. **Requête SQL :** La route effectue une requête SQL pour sélectionner tous les utilisateurs dans la table 'users', les triant par ordre décroissant de création.
2. **Gestion des Erreurs :** En cas d'erreur lors de l'exécution de la requête SQL, un message d'erreur est affiché dans la console, et une réponse JSON avec le statut 500 (Internal Server Error) est renvoyée au client.
3. **Sanitisation des Données :** Les résultats de la requête sont traités pour la sanitisation des données avant d'être renvoyés au client. La sanitisation implique le nettoyage des valeurs sensibles, telles que les emails et les mots de passe, pour prévenir les attaques d'injection de code malveillant.
4. **Réponse JSON :** En cas de succès, les résultats traités sont renvoyés au client sous forme de réponse JSON.

Donc on peut dire qu'au final, cette route permet de récupérer la liste des utilisateurs, en veillant à nettoyer les données sensibles avant de les transmettre au côté client.

A présent, permettez-moi de vous décrire les étapes pour établir une connexion entre pairs à l'aide de Socket.IO et WebRTC

Dans un premier temps, lorsque l'utilisateur va créer sa « room » nous allons récupérer sa permission afin de pouvoir utiliser sa source audio/video. Pour cela j'ai créé la méthode getPermissions. Cette méthode est appelée dans le constructeur.

```
getPermissions = async () => {
  try {
    const videoStream = await navigator.mediaDevices.getUserMedia({
      video: true,
    })
    const audioStream = await navigator.mediaDevices.getUserMedia({
      audio: true,
    })

    // en faisant "!!navigator.mediaDevices.getDisplayMedia" je vérifie si la méthode getDisplayMedia est dispo dans le navigateur
    // si c'est le cas ça veut dire que le navigateur supporte le partage d'écran
    // donc ça me retourne true et donc je met à jour ma state
    // (state qui va déterminer si j'affiche le bouton de partage ou non)
    const screenAvailable = !!navigator.mediaDevices.getDisplayMedia
    this.setState({ screenAvailable })

    if (videoStream || audioStream) {
      // si on a l'autorisation pour l'audio ou la vidéo de l'utilisateur
      window.localStream = videoStream || audioStream // je récupère le flux autorisé par l'utilisateur dans window.localStream
      this.myVideo.current.srcObject = window.localStream // j'affiche ce flux dans mon élément vidéo
      this.setState({ loadingCamera: false })
    }

    this.videoAvailable = !!videoStream // videoAvailable à true si videoStream est true et vice versa
    this.audioAvailable = !!audioStream // même délire
    this.screenAvailable = screenAvailable
  } catch (error) {
    console.error(error)
    this.setState({ loadingCamera: false })
  }
}
```

`getPermissions` permet d'obtenir les autorisations de l'utilisateur pour accéder à sa caméra et à son microphone, et détermine également si le partage d'écran est disponible dans le navigateur.

- Autorisations de la Caméra et du Microphone :** La méthode utilise l'API webRTC `navigator.mediaDevices.getUserMedia` pour demander à l'utilisateur l'autorisation d'accéder à sa caméra (`video: true`) et à son microphone (`audio: true`).
- Vérification de la Disponibilité du Partage d'Écran :** La méthode vérifie la disponibilité de la fonction `navigator.mediaDevicesgetDisplayMedia` pour déterminer si le navigateur supporte le partage d'écran. La variable `screenAvailable` est mise à jour en conséquence.
- Mise à Jour de l'État :** Les résultats des autorisations pour la caméra et le microphone sont vérifiés. Si au moins l'un d'entre eux est autorisé, le flux autorisé est récupéré (`window.localStream`) et affiché dans un élément vidéo (`this.myVideo.current`). L'état de chargement de la caméra est également mis à jour.
- Mise à Jour de l'Indicateur de Disponibilité :** Les indicateurs `videoAvailable` et `audioAvailable` sont mis à jour en fonction de la disponibilité des flux vidéo et audio.
- Gestion des Erreurs :** En cas d'erreur lors de la demande d'autorisations, un message d'erreur est affiché dans la console, et l'état de chargement de la caméra est mis à jour pour refléter le souci.

Donc cette méthode gère la demande d'autorisations pour la caméra et le microphone de l'utilisateur, détermine la disponibilité du partage d'écran, et met à jour les états en conséquence. Elle assure une initialisation appropriée des flux vidéo et audio pour la vidéoconférence.

Une fois les permissions récupérées on se connecte à la room (avec son email autorisé et son username) et getMedia est appelée :

```
getMedia = () => {
  this.setState(
    {
      video: this.videoAvailable, // videoAvailable = acces video autorisé par user donc this.state.video sera true
      audio: this.audioAvailable, // ...
    },
    () => {
      this.getUserMedia() // lorsque l'utilisateur arrivera dans la room sa camera/son audio sera activée ou désactivée en fonction des permissions
      this.connectToSocketServer() // ensuite j'enclenche la logique de connexion server notamment en recuperant l'username, l'email, signal pour SDP/iceCandidates etc...
    }
  )
}
```

getMedia est appelée une fois que l'on vient de se connecter à la room. Cette méthode va set les states video et audio en fonction de ce qu'a permis l'utilisateur (audioAvailable et videoAvailable sont set dans getPermissions).

Ensuite, il appellera getUserMedia afin que l'utilisateur arrive dans la room, sa caméra et son audio seront activés ou désactivés en fonction des permissions. J'initialise ensuite la connexion avec mon serveur pour la logique de sockets.

getUserMedia :

```
getUserMedia = () => {
  if (
    (this.state.video && this.videoAvailable) ||
    (this.state.audio && this.audioAvailable)
  ) {
    navigator.mediaDevices
      // en param je mets ce que je veux récupérer
      .getUserMedia({ video: this.state.video, audio: this.state.audio })
      // ce machin va me retourner un obj "MediaStream" qui est simplement le flux que j'ai récupéré (audio + video ou audio/video)
      .then(this.getUserMediaSuccess) // si c'est good j'appelle getUserMediaSuccess qui récupérera le MediaStream en param
      .then((stream) => {})
      .catch((e) => console.log(e))
  } else { // sinon si ni l'audio ni la vidéo sont activés je stop tout en même temps :
    try {
      let tracks = this.myVideo.current.srcObject.getTracks()
      tracks.forEach((track) => track.stop())
    } catch (e) {
      console.log(e)
    }
  }
  // à savoir que getUserMedia est appelé à chaque fois qu'un utilisateur désactive et ou réactive son audio ou sa caméra afin de stopper ou réactiver le media correspondant
  // en rappelant getUserMedia cette fois si avec les states mises à jour (voir handleAudio et handleVideo)
}
```

- Cette méthode sert à récupérer du flux média à partir de la caméra et/ou du microphone en fonction des états actuels de la vidéo et de l'audio (donc cela peut être les permissions qui sont refusées ou un mute/demute désactivation caméra/activation caméra)
- Si l'un des médias est activé ou désactivé, j'appelle **navigator.mediaDevices.getUserMedia** avec les options appropriées et chaîne plusieurs actions avec **.then** et **.catch**.
- Si ni la vidéo ni l'audio ne sont activés, je stop tous les tracks du flux média actuel. Ensuite, j'appelle la méthode **getUserMediaSuccess**.

getUserMediaSuccess :

```

getUserMediaSuccess = (stream) => {
    // "stream" contient mon objet MediaStream qui m'est retourné quand l'user a accepté qu'on ait accès à sa cam + micro..
    // (vois plus haut dans getUserMedia)
    // Met à jour le flux local avec le nouveau flux de la caméra/microphone.
    window.localStream = stream
    this.myVideo.current.srcObject = stream
    this.setupAudioAnalyser(stream, socketId); // appel à cette fonction pour la gestion de l'affichage de la frame verte autour de la vidéo lorsqu'un utilisateur parle

    // ici je boucle dans toutes les connexions actuelles..
    for (let id in connections) {
        if (id === socketId) continue // la jdis que si dans la liste des sockets ya une id qui correspond à MA socketId(moi)
        // alors je saute l'itération, j'ui dis de pas calculer et de continuer son petit bonhomme de chemin

        // je stream la petite bouille du streamer à tous les users dans la room en ajoutant le flux actuel à toutes les connexions webRTC
        connections[id].addStream(window.localStream)

        // ici je DOIS créer une offre qui contient un "SDP" (go check https://developer.mozilla.org/fr/docs/Glossary/SDP)
        // et ce sera envoyé aux autres users
        connections[id]
            .createOffer()
            .then((description) => connections[id].setLocalDescription(description))
            .then(() => {
                // du coup j'envoie tout ça via une websocket..
                // mon émission "signal" contiendra mon offer pour que tous les autres users puissent la recevoir
                // createOffer qui va créer une SDP est un processus OBLIGATOIRE pour établir une connexion WebRTC
                // c'est comme si t'allais à la banque pour ouvrir un compte et tu signes aucun papier..
                socket.emit(
                    "signal",
                    id,
                    JSON.stringify({ sdp: connections[id].localDescription })
                )
            })
            .catch((e) => console.log(e))
    }
}
    
```

- Cette méthode est appelée en cas de succès de l'obtention du flux média via `getUserMedia`.
- Elle met à jour le flux local (`window.localStream`) avec le nouveau flux de la caméra/microphone et configure l'élément vidéo (`this.myVideo.current`, **current car myVideo est une ref. Cela me permet d'accéder à l'élément du dom sans pour autant provoquer un rerender**).
- J'itére ensuite sur les connexions existantes, ajoutant le nouveau flux à chaque connexion, créant une offre (**SDP ou Session Description Protocol**), et émettant un signal via la WebSocket pour informer les autres utilisateurs de l'offre.

La transmission du SDP entre les pairs est indispensable pour établir une connexion webRTC.

`createOffer` va créer une `localDescription` qui contiendra entre autres le fameux SDP.

Que contient un SDP ? Il contient différentes informations telles que le codec, l'adresse source, et les informations temporelles pour l'audio et la vidéo.

Une fois la `localDescription` prête, je l'envoie avec l'id de mes peers connectés à la room via websocket (j'utilise `socket.io`).

Comment ça marche exactement ?

```
// C'est comme si t'allais à la banque pour ouvrir un compte et
socket.emit(
  "signal",
  id,
  JSON.stringify({ sdp: connections[id].localDescription })
)
  .catch(error => console.error(error))
```

J'effectue ce que l'on appelle une émission. Personnellement, je vois ça un peu comme un mix de call api et d'event listener personnalisé où l'on peut envoyer des données dans le corps de la requête (payload ici en l'occurrence) et le nom de l'émission (« signal » qui s'apparente un peu à un event) fait office d'endpoint. Une fois l'émission faite, on peut « l'écouter » côté serveur !

Côté serveur :

```
socket.on('signal', (toId, message) => { // message contient le SDP généré avec createOffer côté front
  io.to(toId).emit('signal', socket.id, message) // j'émite le signal du socket.id (moi) vers les autres sockets
})
```

Du côté serveur, j'écoute l'émission (ou l'event, c'est comme vous voulez !) 'signal' à l'aide de **socket.on**. Le paramètre **toId** représente l'identifiant du destinataire de ce signal, et le paramètre **message** contient le Session Description Protocol (SDP) généré avec **createOffer** du côté client.

En réponse à cet événement, une nouvelle émission est effectuée vers la socket spécifiée par **toId** à l'aide de **io.to(toId).emit**. Cette émission contient l'identifiant (**socket.id**) de la socket émettrice (moi) et le SDP associé. Ces informations seront ensuite reçues côté client dans la méthode 'signalFromServer'.

Comme mentionné précédemment, l'échange de SDP (Session Description Protocol) via la création d'une offre (**createOffer**) est crucial pour établir une connexion WebRTC entre les pairs. Cependant, ce n'est pas la seule étape nécessaire pour assurer une communication WebRTC réussie.

IceCandidates :

En plus de l'échange de SDP, les Ice Candidates jouent un rôle essentiel. Les Ice Candidates sont des informations qui spécifient comment les deux pairs peuvent se connecter. Ces candidats sont générés localement pour chaque pair et doivent être partagés entre les pairs pour permettre une connexion peer-to-peer réussie.

Qu'est-ce qu'un IceCandidate ? En gros, un Ice Candidate contient des informations sur la connectivité réseau d'un pair (meilleurs path réseau entre les peers par exemple), et ces candidats sont échangés entre les pairs pour faciliter l'établissement d'une connexion peer-to-peer robuste, même à travers des configurations réseau complexes.

Pour implémenter cela, chaque pair génère ses Ice Candidates localement et les transmet à l'autre pair via des émissions WebSocket. Le serveur agit comme un médiateur pour relayer ces informations entre les pairs. Du côté serveur, la réception d'Ice Candidates déclenche une émission vers le pair cible.

Comment l'iceCandidate est généré ? Vous vous souvenez de setLocalDescription ? il contient non pas le SDP seulement mais également les iceCandidates !

Si vous m'avez bien suivi, vous savez que j'ai envoyé setLocalDescription via websocket (« signal ») et donc cela est renvoyé à tous les peers présents dans la room.

SignalFromServer :

Maintenant que le SDP et les iceCandidates sont générés côté client et transmis via websockets pour qu'en suite le serveur joue son rôle de médiateur pour les redistribuer à tous les autres peers, on peut maintenant les réceptionner côté client car forcément, on est susceptible nous aussi de recevoir autant d'offers que d'utilisateurs dans une room.

```
4 // ici je vais réceptionner tout ce qui est SDP/iceCandidates
5 signalFromServer = (fromId, body) => {
6   let signal = JSON.parse(body)
7
8 >   if (signal.speaking !== undefined) {
9     }
10
11   if (fromId !== socketId) {
12     // j'assure que l'id du client (fromId) est différent du mien (socketId)
13     if (signal.sdp) {
14       // si ya une prop "sdp" dans signal (prop générée lors du createOffer)
15       connections[fromId]
16         .setRemoteDescription(new RTCSessionDescription(signal.sdp)) // alors je contrôle le sdp de l'autre peer
17         .then(() => {
18           // si le type du sdp est "offer" ça veut dire que c'est une offre qui vient d'un autre client btw
19           if (signal.sdp.type === "offer") {
20             connections[fromId]
21               .createAnswer() // du coup je créer une réponse (answer) à l'offer que j'ai reçu,
22               // c'est obligatoire
23               .then((description) => {
24                 connections[fromId]
25                   // une fois arrivé à l'appel de setLocalDescription(),
26                   // webRTC va commencer le processus de collecte des IceCandidates (fourni par navigateurs)
27                   .setLocalDescription(description)
28                   .then(() => {
29                     // déjà vu ..
30                     socket.emit(
31                       // déjà vu ..
32                       "signal", // déjà vu ..
33                       fromId, // déjà vu ..
34                       JSON.stringify({
35                         | sdp: connections[fromId].localDescription, // déjà vu ..
36                         | })
37                       )
38                     })
39                   .catch((e) => console.log(e))
40
41                   .catch((e) => console.log(e))
42                   .catch((e) => console.log(e))
43
44               }
45
46             // pour comprendre un peu ICE(Interactive Connectivity Establishment),
47             // jte conseille : https://developer.mozilla.org/en-US/docs/Web/API/RTCIceCandidate ou alors chatGPT of course
48             // Si tu veux d'un côté t'as les SDP (contenant des infos type codec vidéo/audio ou tout autre paramètres)
49             // Ce décris COMMENT les médias doivent être échangés entre les peers
50             // d'un autre côté tu as les iceCandidates qui est un peu dans le même principe mais
51             // fournit plutôt des infos sur la connectivité réseau (ip, routing, ports, protocols..)
52             // afin de pouvoir choisir le meilleur chemin réseau pour communiquer entre les peers
53             // ICE et l'offre SDP doivent toujours être utilisés ensemble pour pouvoir établir une connection webrtc
54             if (this.iceCandidatesQueue[fromId]) {
55               this.iceCandidatesQueue[fromId].forEach((candidate) => {
56                 // un client envoie son message (fromId)
57                 connections[fromId].addIceCandidate(
58                   // j'ajoute mes iceCandidates à ma connection p2p
59                   new RTCIceCandidate(candidate)
60                 )
61               }
62             }
63             // quand tous les candidats ont été ajoutés à la classe RTCIceCandidate, ils sont delete car y'en a plus besoin
64             delete this.iceCandidatesQueue[fromId]
65           }
66         }
67       .catch((e) => console.log(e))
68
69       // du coup logiquement après un createOffer ou createAnswer tu as des iceCandidates
70       // ça veut dire qu'un peer a trouvé un bon chemin de connexion réseau et il l'envoie pour qu'un autre peer puisse l'essayer
71       if (signal.ice) {
72         let iceCandidate = new RTCIceCandidate(signal.ice) // du coup je créer mon obj RTCIceCandidate à partir du ice reçu
73         if (connections[fromId].remoteDescription) {
74           // si setRemoteDescription s'est déroulé comme il faut
75           connections[fromId]
76             .addIceCandidate(iceCandidate) // j'ajoute ENFIN l'icecandidate
77             .catch((e) => console.log(e))
78         } else {
79           if (!this.iceCandidatesQueue[fromId]) {
80             // Si pas de remoteDescription,
81             // alors je stock les iceCandidates en attendant la remoteDescription
82             this.iceCandidatesQueue[fromId] = []
83           }
84           // du coup en attendant je met les iceCandidate dans une file d'attente
85           this.iceCandidatesQueue[fromId].push(iceCandidate)
86         }
87       }
88     }
89   }
90 }
```

Dans la fonction **signalFromServer**, je gère la réception des signaux provenant du serveur, notamment les SDP (Session Description Protocol) et les Ice Candidates.

1. Analyse du signal :

- La fonction commence par « parser » le body du signal JSON reçu du serveur pour extraire les informations nécessaires. Le signal contient SDP (**sdp**), ou des Ice Candidates (**ice**).

2. Traitement des SDP :

- Si le signal contient un SDP (**sdp**), je vérifie si l'identifiant de l'émetteur (**fromId**) est différent de l'identifiant local (**socketId**). Cela garantit que l'on ne traite pas les signaux provenant de soi-même.
- La méthode utilise **setRemoteDescription** pour définir la description distante avec le SDP reçu.
- Si le type du SDP est de type "offer", cela signifie qu'il s'agit d'une offre provenant d'un autre client. Dans ce cas, une "answer" est créée (c'est mandatory..) à l'aide de **createAnswer**, et la description locale est définie avec **setLocalDescription**.
- Enfin, la description locale mise à jour est envoyée au serveur via une émission WebSocket.

3. Traitement des Ice Candidates :

- Si le signal contient des Ice Candidates (**ice**), je créer un nouvel objet **RTCIceCandidate** à partir des données Ice reçues.
- Je vérifie ensuite si la description distante a déjà été définie (**remoteDescription**). Si oui, l'Ice Candidate est ajouté à la connexion avec **addIceCandidate**.
- Si la description distante n'est pas encore définie, l'Ice Candidate est mis en file d'attente dans **iceCandidatesQueue** (une state qui est en fait un tableau vide qui va stocker les autres iceCandidates et ils repasseront plus tard) en attendant la réception de la description distante.

Maintenant que l'on a accompli ces étapes, la connexion webRTC entre pairs est enfin établie. Les flux vidéo et audio des pairs peuvent être exploités dans le DOM.

Parlons un peu du mécanisme de chat :

```
sendMessage = () => {
  if (this.state.message.trim() !== "") {
    socket.emit("chat-message", this.state.message, this.state.username); // j'émite les states username et message
    // une fois le message envoyé, j'erset l'input à vide et je laisse this.username as sender of course
    this.setState({ message: "", sender: this.state.username })
  }
}
```

Lorsqu'un utilisateur envoie un message, la méthode **sendMessage** est déclenchée du côté client. Elle émet alors un signal "chat-message" avec pour payload le message et le nom de l'utilisateur. Du côté serveur, ce signal est capturé par la fonction associée à l'événement "chat-message".

```

socket.on('chat-message', (data, sender) => {
  data = sanitizeString(data); // on rend safe vs failles xss
  sender = sanitizeString(sender); // idem

  for (const key in connections) {
    if (connections[key].includes(socket.id)) { // je vérifie dans quel room mon socket est présent

      connections[key].forEach((key) => { // dans la room de mon socket id j'émite les messages et le sender
        // ainsi que le socket.id car côté front je vérifie que la personne qui envoie le message n'est pas moi même avant d'envoyer la notification et le son de la notif
        io.to(key).emit("chat-message", data, sender, socket.id);
      })
      break; // doit stop la boucle quand je trouve la room
    }
  }
});
    
```

Le serveur effectue une étape de ‘sanitization’ pour prévenir les failles XSS, garantissant ainsi la sécurité des données. Ensuite, il parcourt les connexions pour déterminer dans quelle "room" (salle de discussion) l'émetteur du message est actuellement présent. Une fois la room identifiée, le serveur émet le message à tous les utilisateurs présents dans cette room.

```
socket.on("chat-message", this.addMessage) // je récupère les messages émis côté serveur pour les display
```

```

addMessage = (data, sender, socketIdSender) => {
  this.setState((prevState) => ({
    messages: [...prevState.messages, { sender: sender, data: data }], // je prend le tableau messages
    // et y ajoute sender et data sans y écraser les autres msgs dans le tableau messages
  }));

  if (socketIdSender !== socketId) {
    // si c'est pas moi qui envoie le msg, j'incrémente le chiffre de la notif d'un new msg
    this.setState({ newmessages: this.state.newmessages + 1 })
    this.playMessageSound()
  }
}
    
```

Enfin, côté client, la méthode **addMessage** réceptionne les messages provenant du serveur. Elle met à jour l'état des messages en ajoutant le nouveau message sans écraser les précédents. De plus, si le message provient d'un autre utilisateur, le compteur de nouveaux messages est incrémenté, déclenchant une notification sonore.

Résumons :

Tout commence par une phase d'authentification où l'utilisateur entre son email et son nom d'utilisateur. Un appel à une API récupère la liste des utilisateurs autorisés. La connexion n'est autorisée que pour les utilisateurs autorisés.

Après l'authentification, l'utilisateur obtient les permissions pour la caméra et le microphone avec la méthode `getPermissions`, déterminant également la disponibilité du partage d'écran. Ensuite, la méthode `getMedia` est appelée, configurant les flux vidéo et audio en fonction des autorisations. Une connexion à la room est établie, et `getUserMediaSuccess` est déclenchée.

getUserMediaSuccess met à jour le flux local et émet des offres (SDP) via WebSocket à tous les pairs dans la room. La réception des SDP côté serveur entraîne la création d'une réponse (answer) et l'émission vers les pairs concernés. De plus, les Ice Candidates, générés avec setLocalDescription, sont échangés pour une connexion robuste.

La réception des signaux côté client est gérée par signalFromServer, qui analyse les SDP et les Ice Candidates. Les SDP sont utilisés pour définir les descriptions locales et distantes (codec audio/video etc..), tandis que les Ice Candidates assurent une connectivité réseau optimale (meilleurs path réseau possible etc..). Les connexions entre pairs sont ainsi établies.

Donc le processus dans l'ensemble met en évidence la synchronisation des SDP, la gestion des Ice Candidates, et l'utilisation de Socket.IO pour la signalisation.

Administration :

```
1  import React, { useState, useEffect } from "react"
2  import axios from "axios"
3  import { Link, useNavigate } from "react-router-dom"
4  import { Button } from "@material-ui/core"
5  import { message } from "antd" // superbe bibliothèque..
6  import logo from "../../assets/hdmlogo.png"
7  import { useAuth } from "../../context/authContext"
```

AuthAdmin gère l'authentification des administrateurs. Il utilise les fonctionnalités de React, telles **useState** et **useEffect**, ainsi qu'Axios pour effectuer des requêtes HTTP, Material-UI pour les composants d'interface utilisateur, et Ant Design pour les messages d'alerte. J'utilise aussi le contexte useAuth afin de stocker un token dans un localStorage (plus d'informations à venir après..)

```
const [adminEmail, setAdminEmail] = useState("")
const [adminPassword, setAdminPassword] = useState("")
const { login } = useAuth() // fonction login de authContext
const navigate = useNavigate()

useEffect(() => {
  const params = new URLSearchParams(window.location.search);
  const sessionExpired = params.get('sessionExpired');

  if (sessionExpired === 'true') {
    message.warning('Votre session a expiré. Veuillez vous reconnecter.');
  }
}, []);
```

Je commence par définir des états pour l'email et le mot de passe de l'administrateur grâce à **useState**. Je vérifie la présence du paramètre **sessionExpired** dans l'URL puis j'affiche un avertissement approprié si nécessaire et ce dans un **useEffect**.

useEffect avec un tableau de dépendances vide (comme vous le voyez sur l'illustration) est utilisé pour effectuer des tâches qui ne nécessitent une exécution qu'une seule fois lors du rendu initial du composant. Ici cela inclue la vérification d'URL pour le token expiré et l'affichage du message si besoin.

```

const handleAdminEmailChange = (e) => {
  setAdminEmail(e.target.value.toLowerCase())
}

const handleAdminPasswordChange = (e) => {
  setAdminPassword(e.target.value)
}

```

Les fonctions **handleAdminEmailChange** et **handleAdminPasswordChange** sont des gestionnaires d'événements qui mettent à jour les states respectifs en fonction des changements dans les champs de saisie.

```

const loginAsAdmin = () => {
  if (adminEmail === "" || adminPassword === "") {
    message.warning("Merci de remplir tous les champs svp ! ")
    return
  }

  axios
    .post("http://localhost:4001/login", {
      email: adminEmail,
      password: adminPassword,
    })
    .then((response) => {
      const { token } = response.data

      login(token) // j'envoie le token en localstorage via la methode login de authContext !

      localStorage.setItem("adminEmail", adminEmail)
      navigate("/adminPanel")

    })
    .catch((error) => {
      if(error.response.status === 403){
        message.error("Vous n'êtes pas admin, zou !")
      }
      if(error.response.status === 401){
        message.error("Mot de passe incorrect !")
      }
      console.error("erreur! " + error)
    })
}

```

La fonction **loginAsAdmin** est appelée lorsqu'un administrateur tente de se connecter. Elle vérifie que les champs d'email et de mot de passe sont remplis, effectue une requête HTTP POST vers le serveur avec ces informations, et traite la réponse. En cas de succès, un token est extrait de la réponse, stocké localement, et l'utilisateur est redirigé vers la page du panneau d'administration.

Concernant login, parlons de la route /login côté serveur :

```
app.post('/login', (req, res) => {
  const { email, password } = req.body;

  const cleanEmail = sanitizeString(email)
  const cleanPassword = sanitizeString(password)

  const query = 'SELECT * FROM users WHERE email = ?';
  connection.query(query, [cleanEmail], (err, results) => {

    if (results.length === 0) {
      return res.status(401).json({ error: 'Utilisateur non trouvé.' });
    }

    const user = results[0];

    if(results[0].role !== "ADMIN"){
      return res.status(403).json({error : "Vous n'êtes pas admin, zou !"})
    }

    // jcompare le mdp fourni avec celui qui est haché en bdd
    bcrypt.compare(cleanPassword, user.password, (bcryptErr, passwordMatch) => {
      if (bcryptErr) {
        return res.status(500).json({ error: 'Erreur lors de l\'authentification.' });
      }

      if (passwordMatch) {
        // le password a match alors je genere un token
        const token = jwt.sign({ email: user.email, role: user.role }, process.env.JWT_SECRET, { expiresIn: "30m" });
        console.log('Token généré :', token);
        res.json({ token });
      } else {
        res.status(401).json({ error: 'Mot de passe incorrect.' });
      }
    });
  });
});
```

La route est définie en tant que point d'entrée pour les requêtes HTTP POST à l'adresse **/login**. Elle récupère les informations d'authentification de l'utilisateur, à savoir l'email et le mot de passe, à partir du corps de la requête.

Avant d'effectuer les vérifications d'authentification, la route utilise le module **express-session** pour configurer la gestion des sessions. Cette configuration inclut un secret (que je définis dans une variable d'environnement par souci de sécurité) pour signer les cookies de session, et elle est utilisée pour stocker temporairement des informations liées à la session de l'utilisateur.

Ensuite, la route vérifie si l'email et le mot de passe ont été fournis dans la requête. Si l'un de ces champs est manquant, la route renvoie une réponse avec un code d'erreur 400 (Bad Request, un truc qui cloche avec la query par exemple) indiquant que l'email et le mot de passe sont requis.

Si les informations d'authentification sont fournies, la route effectue une requête SQL pour récupérer l'utilisateur correspondant à l'email fourni. En cas d'erreur lors de la recherche dans la base de données, la route renvoie une réponse avec un code d'erreur 500 (Internal Server Error, connexion avec le server qui cloche en général).

Si aucun utilisateur n'est trouvé avec l'email fourni, la route renvoie une réponse avec un code d'erreur 401 (Unauthorized), indiquant qu'aucun utilisateur correspondant n'a été trouvé.

Si un utilisateur est trouvé, la route utilise **bcrypt.compare** pour comparer le mot de passe fourni avec le mot de passe haché stocké en base de données. En cas d'erreur lors de la comparaison, la route renvoie une réponse avec un code d'erreur 500.

Si les mots de passe correspondent, la route génère un token JWT (JSON Web Token) contenant des informations telles que l'email de l'utilisateur et son rôle. Ce token est signé avec une clé secrète, et sa validité est limitée à 30 minutes. Le token est ensuite renvoyé dans la réponse JSON.

Si les mots de passe ne correspondent pas, la route renvoie une réponse avec un code d'erreur 401, indiquant que le mot de passe est incorrect.

Revenons à notre composant AuthAdmin :

```
const handleKeyDown = (e) => {
  if (e.key === "Enter") {
    loginAsAdmin()
  }
}
```

La fonction **handleKeyDown** permet à l'utilisateur de déclencher la connexion en appuyant sur la touche "Enter" lors de la saisie (oui je suis feignant je préfère appuyer sur enter plutôt que de prendre ma souris et cliquer sur connexion !).

```
return (
  <div>
    <Link to="/">
      <img
        className="logo"
        src={logo}
        alt=""
        style={{ width: "150px", position: "absolute", top: "0", left: "0" }}
      />
    </Link>
    <br />
    <div className="content">
      <input
        type="email"
        name="email"
        placeholder="Email administrateur"
        onChange={handleAdminEmailChange}
        onKeyDown={handleKeyDown}
      />{" "}
      <br />
      <br />
      <input
        type="password"
        name="password"
        placeholder="Mot de passe"
        onChange={handleAdminPasswordChange}
        onKeyDown={handleKeyDown}
      />
      <br />
      <br />
      <Button
        className="startBtn"
        variant="contained"
        color="primary"
        onClick={loginAsAdmin}
      >
        Se connecter
      </Button>
    </div>
  </div>
)
```

En termes d'interface utilisateur, le composant affiche des champs d'entrée pour l'email et le mot de passe, ainsi qu'un bouton "Se connecter" stylisé à l'aide de la balise Button de Material-UI. Une image du logo du projet est également affichée avec un lien vers la page d'accueil.

Revenons côté front.

Une fois authentifié, on est redirigé vers AdminPanel. Mais avant..

Bon à savoir : AdminPanel est une route protégée. Cela signifie que l'accès à cette section est réservé aux utilisateurs authentifiés.

Pour garantir cette sécurité, j'ai mis en place un composant **ProtectedRoute** qui englobe notre composant AdminPanel.

Jetons un coup d'œil à la structure du fichier **App.jsx** :

```

import React, { Component } from 'react';
import Main from './components/Main.jsx';
import Home from './components/Home.jsx';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import AuthAdmin from './components/admin/AuthAdmin.jsx';
import AdminPanel from './components/admin/AdminPanel.jsx';
import { AuthProvider } from './context/authContext.jsx';
import ProtectedRoute from './components/admin/protectedRoute.jsx';

class App extends Component {
    render() {
        return (
            <AuthProvider>
                <div>
                    <Router>
                        <Routes>
                            <Route path="/" element={<Home />} />
                            <Route path="/authAdmin" element={<AuthAdmin />} />
                            <Route path="/adminPanel/*" element={<ProtectedRoute component={AdminPanel} />} />
                            <Route path="/:url" element={<Main />} />
                        </Routes>
                    </Router>
                </div>
            </AuthProvider>
        );
    }
}

export default App;

```



La route **/adminPanel/*** est associée au composant **ProtectedRoute**, qui elle-même enveloppe le composant **AdminPanel**. La fonction **ProtectedRoute** vérifie si l'utilisateur est authentifié avant de lui permettre l'accès à la page d'administration. Cela offre une couche supplémentaire de sécurité pour s'assurer que seuls les utilisateurs autorisés peuvent accéder à AdminPanel.

```

import React from 'react';
import { Route, Routes, Navigate } from 'react-router-dom';
import { useAuth } from '../../../../../context/authContext';

const ProtectedRoute = ({ component: Component }) => { // dans app.js j'utilise le component protectedRoute et en param je fou le composant adminPanel
    const { token } = useAuth(); // mon authcontext me retourne un token

    return token ? ( // si ya un token in ze building alors je dirige vers la route qu'il faut (adminPanel en occurrence)
        <Routes>
            | <Route path="/*" element={<Component />} />
        </Routes>
    ) : (
        <Navigate to="/authAdmin" /> // sinon (pas connecté) faut se login
    );
};

export default ProtectedRoute;

```

Le composant **ProtectedRoute** prend en compte le contexte d'authentification **useAuth()** (**on y reviendra**) pour vérifier la présence du token. Si un token est présent, l'utilisateur est redirigé vers la route spécifiée (dans ce cas, AdminPanel). Si aucun token n'est présent (c'est-à-dire si l'utilisateur n'est pas authentifié), il est redirigé vers la page d'authentification (**/authAdmin**).

AuthContext :

```
// CONTEXT = permet de wrapper des composants enfants pour pouvoir leurs transmettre des states ou méthodes du coup,
// plus besoin de transmettre tout ça via props à chaque niveau
const AuthContext = createContext();

// du coup mon authProvider va être utilisé en tant que wrapper pour gérer l'état d'authentification des composants enfants
export const AuthProvider = ({ children }) => {
  const [token, setToken] = useState(localStorage.getItem('authToken') || null); // en fonction de là où on se situe dans l'app et du moment, il peut y avoir un token ou non .
  // Faut savoir que authContext est utilisé comme wrapper (via le component protectedRoute) donc à tout moment l'app est soumise au context
  const [isTokenExpired, setIsTokenExpired] = useState(false);

  useEffect(() => { // gestion expiration token
    const checkTokenExpiration = () => {
      if (token) {
        try {
          const decodedToken = jwtDecode(token);
          const currentTime = Math.floor(Date.now() / 1000); // nombre de secondes écoulées depuis 1970 LOL
          setIsTokenExpired(decodedToken.exp < currentTime); // decodedToken.exp == heure à laquelle je set sign le token côté endpoint /login (server) + 30 mn (expiration)
        } catch (error) {
          console.error("Erreur JWT :", error);
        }
      }
    };
    checkTokenExpiration();
  }, [token, setIsTokenExpired]);

  const login = (newToken) => { // cette méthode va prendre le token que j'ai récup dans le call api d'adminPanel pi va le foutre dans localstorage
    setToken(newToken);
    localStorage.setItem('authToken', newToken);
  };

  // je gère le logout directement dans adminPanel..

  // grâce à ce que je return j'avais pouvoir recuperer token, isAuthenticated et login en faisant un useAuth dans mes autres components
  // je convertis token en boolean pour set isAuthenticated comme il se doit
  return (
    <AuthContext.Provider value={({ token, isAuthenticated: Boolean(token), isTokenExpired, login })}>
      {children}
    </AuthContext.Provider>
  );
};

// j'creer un hook perso "useAuth" qui permet de faire en sorte que les composants enfants pourront utiliser useAuth pour avoir accès à ce que je return dans mon authProvider (token, isAuthenticated et login)
export const useAuth = () => {
  return useContext(AuthContext);
};
```

```
import React, { Component } from 'react';
import Main from './components/Main.jsx';
import Home from './components/Home.jsx';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';
import AuthAdmin from './components/admin/AuthAdmin.jsx';
import AdminPanel from './components/admin/AdminPanel.jsx';
import { AuthProvider } from './context/authContext.js';
import ProtectedRoute from './components/admin/protectedRoute.jsx';

class App extends Component {

  render() {
    return (
      <AuthProvider> ←
        <div>
          <Router>
            <Routes>
              <Route path="/" element={<Home />} />
              <Route path="/authAdmin" element={<AuthAdmin />} />
              <Route path="/adminPanel/*" element={<ProtectedRoute component={AdminPanel} />} />
              <Route path="/:url" element={<Main />} />
            </Routes>
          </Router>
        </div>
      </AuthProvider>
    );
  }

  export default App;
}
```

Utilisé pour gérer l'état d'authentification des composants enfants de l'application, ce contexte permet de transmettre des informations telles que le token, l'état d'authentification (**isAuthenticated**), et la méthode de connexion (**login**) à travers l'application sans avoir besoin de passer ces valeurs via les props à chaque niveau.

Le composant **AuthProvider** est utilisée comme un wrapper pour englober les composants enfants et gérer l'état d'authentification. Il utilise le hook **useState** pour définir et mettre à jour le token, en récupérant éventuellement une valeur depuis le **localStorage** au chargement de l'application.

La méthode **login** est définie pour mettre à jour le token en cas d'authentification réussie, stockant également ce token dans le **localStorage** pour une persistance à travers les rechargements de page.

Le composant **AuthProvider** expose le contexte d'authentification à ses enfants via la méthode **useContext**. Ainsi, tout composant enfant peut utiliser le hook **useAuth** pour accéder aux informations d'authentification fournies par le contexte (**login**, **isAuthenticated**, **token**).

AdminPanel :

```
1 // Import des dépendances
2 import React, { Component } from "react"
3 import { Button } from "@material-ui/core"
4 import { Link } from "react-router-dom"
5 import { message } from "antd"
6 import logo from "../../assets/hdmlogo.png"
7 import Rodal from "roodal"
8 import "roodal/lib/roodal.css"
9 import axios from "axios"
10 import "../../style/Admin.css"
11 import { jwtDecode } from "jwt-decode"
12
13 class AdminPanel extends Component {
14   constructor(props) {
15     super(props)
16     this.state = {
17       userEmail: "",
18       role: "ADMIN",
19       showCreateUserForm: false,
20       showRoleUpdateForm: false,
21       newUserEmail: "",
22       newUserRole: "USER",
23       users: [],
24       showUserTable: false,
25       selectedUserForRoleUpdate: null,
26       showPasswordInputForUpdate: false,
27       showPasswordInputForCreate: false,
28       password: "",
29       newPassword: ""
30     }
31   }
32   componentDidMount() {
33     this.fetchUsers()
34     const adminEmail = localStorage.getItem("adminEmail")
35     this.setState({ userEmail: adminEmail })
36     message.info("Bienvenue dans votre espace administrateur")
37     const authToken = localStorage.getItem("authToken")
38
39     if (authToken) {
40       try {
41         const decodedToken = jwtDecode(authToken)
42         console.log("Decoded Token:", decodedToken)
43         const currentTime = Math.floor(Date.now() / 1000)
44         if (decodedToken.exp && decodedToken.exp < currentTime) {
45           localStorage.removeItem("authToken")
46           window.location.href = "/authAdmin?sessionExpired=true"
47           return
48         }
49       } catch (error) {
50         console.error("Error decoding JWT:", error)
51       }
52     }
53   }
54 }
```

```
handleLogout = () => {
  localStorage.removeItem("authToken")
  window.location.href = "/authAdmin"
}

fetchUsers = () => {
  axios
    .get(`http://localhost:4001/users`)
    .then((response) => {
      this.setState({ users: response.data })
    })
    .catch((error) => {
      message.error("Erreur lors de la récupération des utilisateurs")
    })
}

openRoleUpdateModal = (user) => {
  this.setState({
    selectedUserForRoleUpdate: user,
    role: user.role,
  })
}

closeRoleUpdateModal = () => {
  this.setState({
    selectedUserForRoleUpdate: null,
    showPasswordInputForUpdate: false,
  })
}

toggleRoleUpdateForm = () => {
  this.setState((prevState) => ({
    showRoleUpdateForm: !prevState.showRoleUpdateForm,
    showPasswordInputForUpdate: prevState.role === "ADMIN",
  }))
}

toggleCreateUserForm = () => {
  this.setState((prevState) => ({
    showCreateUserForm: !prevState.showCreateUserForm,
    showPasswordInputForCreate: prevState.newUserRole === "ADMIN",
  }))
}
```

.... Autres togglers ...

```

handleCreateUser = (e) => {
  e.preventDefault()
  const { newUserEmail, newUserRole, password } = this.state

  axios
    .post(`http://localhost:4001/insertUser`, {
      email: newUserEmail,
      role: newUserRole,
      password: newUserRole === "ADMIN" ? password : "",
    })
    .then((response) => {
      message.success("Nouvel accès créé pour " + newUserEmail + " !")
      this.setState({ newUserEmail: "", password: "" })
      this.fetchUsers()
    })
    .catch((error) => {
      message.error(error.response.data.error)
    })
}

handleUpdateRole = (e) => {
  e.preventDefault()
  const { role, newPassword } = this.state
  const userEmail = this.state.selectedUserForRoleUpdate.email

  axios
    .put(`http://localhost:4001/updateRoles`, {
      email: userEmail,
      newRole: role,
      newPassword: role === "ADMIN" ? newPassword : "",
    })
    .then((response) => {
      message.success("Rôle mis à jour !")
      console.log("new pass " + newPassword)
      this.fetchUsers()
    })
    .catch((error) => {
      message.error(error.response.data.error)
    })
}
    
```

```

handleDeleteUser = (email) => {
  axios
    .delete(`http://localhost:4001/deleteUser/${email}`)
    .then(() => {
      message.success("Utilisateur supprimé avec succès")
      this.fetchUsers()
    })
    .catch((error) => {
      message.error("Erreur lors de la suppression de l'utilisateur")
    })
}
    
```

```
render() {
  const {
    showCreateUserForm,
    newUserEmail,
    newUserRole,
    users,
    showUserTable,
    selectedUserForRoleUpdate,
    showPasswordInputForUpdate,
    showPasswordInputForCreate,
    password,
  } = this.state

  return (
    <div>
      <Link to="/">
        <img
          className="logo"
          src={logo}
          alt=""
          style={{
            width: "150px",
            position: "absolute",
            top: "0",
            left: "0",
          }}
        />
      </Link>
      <div className="content">
        <p
          style={{
            color: "black",
            position: "absolute",
            top: "1%",
            right: "15%",
          }}
        >
          {" "}
          <span className="online-indicator"></span>{" "}
          <b>{this.state.userEmail}</b>{" "}
        </p>
      </div>
    </div>
  )
}
```

```
<Button
  style={{
    backgroundColor: "red",
    color: "white",
    position: "absolute",
    top: "0",
    right: "0",
  }}
  onClick={this.handleLogout}
  variant="contained"
>
  Se déconnecter
</Button>
<br />
<br />
<Button onClick={this.toggleCreateUserForm} variant="contained">
  {showCreateUserForm
    ? "Cacher le formulaire"
    : "Créer un nouvel accès"}
</Button>
<br />
{showCreateUserForm && (
  <form onSubmit={this.handleCreateUser}>
    <div>
      <label>
        <b>Email du nouvel utilisateur: </b>
      </label>
      <input
        type="email"
        value={newUserEmail}
        onChange={this.handleNewUserEmailChange}
        required
        style={{
          backgroundColor: "white",
          borderRadius: "8px",
          margin: "10px",
        }}
      />
    </div>
    <div>
      <label>
        <b>Rôle du nouvel utilisateur:</b>
      </label>
      <select
        value={newUserRole}
        onChange={this.handleNewUserRoleChange}
        style={{ borderRadius: "8px", margin: "10px" }}
      >
        <option value="ADMIN">ADMIN</option>
        <option value="USER">USER</option>
      
```

```
</div>
{showPasswordInputForCreate && (
  <div>
    <label>
      <b>Mot de passe:</b>
    </label>
    <input
      type="password"
      value={password}
      onChange={(e) =>
        this.setState({ password: e.target.value })
      }
      required
      style={{
        backgroundColor: "white",
        borderRadius: "8px",
        margin: "10px",
      }}
    />
  </div>
)
<button
  className="green-button"
  type="submit"
  variant="contained"
  color="primary"
>
  Créer l'utilisateur
</button>
</form>
)}
</div>

<br />
<Button onClick={this.toggleUserTable} variant="contained">
  {showUserTable
    ? "Cacher le tableau"
    : "Afficher le tableau des accès"}

```

```
{showUserTable && (
  <table className="user-table">
    <thead>
      <tr>
        <th>Email</th>
        <th>Rôle</th>
        <th>Date de création</th>
        <th>Action</th>
      </tr>
    </thead>
    <tbody>
      {users.map((user) => (
        <tr key={user.email}>
          <td>
            <b>{user.email}</b>
          </td>
          <td>
            <b>{user.role}</b>
          </td>
          <td>
            <b>{this.formatDate(user.created_At)}</b>
          </td>
          <td>
            <button
              className="red-button"
              onClick={() => this.handleDeleteUser(user.email)}
            >
              Supprimer
            </button>
            <button
              className="blue-button"
              onClick={() => this.openRoleUpdateModal(user)}
            >
              Modifier le rôle
            </button>
          </td>
        </tr>
      ))}
    </tbody>
  </table>
)}
```

```
<Modal
  visible={selectedUserForRoleUpdate !== null}
  onClose={this.closeRoleUpdateModal}
  animation="zoom"
  customStyles={{
    width: "300px",
  }}
>
  <h2>Modifier le rôle de l'utilisateur</h2>
```

```
<h2>Modifier le rôle de l'utilisateur</h2>
{selectedUserForRoleUpdate && (
  <div>
    <br />
    <i>
      {" "}
      <b>{selectedUserForRoleUpdate.email}</b>
    </i>{" "}
    <br />
    <input type="hidden" value={selectedUserForRoleUpdate.email} />
    <br />
    <label>
      {" "}
      <b>Nouveau rôle : &nbsp;</b>
    </label>
    <select value={this.state.role} onChange={this.handleRoleChange}>
      <option value="ADMIN">ADMIN</option>
      <option value="USER">USER</option>
    </select>
    {showPasswordInputForUpdate && (
      <div>
        <label>
          <b>Mot de passe:</b>
        </label>
        <input
          type="password"
          value={this.state.newPassword}
          onChange={(e) =>
            this.setState({ newPassword: e.target.value })
          }
          required
          style={{
            backgroundColor: "white",
            borderRadius: "8px",
            margin: "10px",
          }}
        />
      </div>
    )}
    <br /> <br />
    <button
      onClick={this.handleUpdateRole}
      variant="contained"
      className="green-button"
      type="submit"
    >
      Modifier le rôle
    </button>
    <br />
```

Le composant **AdminPanel** est la partie centrale de l'espace administrateur de l'application.

1. Initialisation du State :

- adminPanel utilise plusieurs states pour gérer plusieurs aspects tels que l'e-mail de l'administrateur, le rôle actuel, l'affichage du formulaire de création d'utilisateur, la liste des utilisateurs, etc.

2. componentDidMount :

- Lorsque le composant est monté, il récupère l'e-mail de l'administrateur à partir du stockage local et affiche un message de bienvenue.
- Il vérifie également si le jeton d'authentification est toujours valide. Si ce n'est pas le cas, il déconnecte l'utilisateur et le redirige vers la page d'authentification avec un message de session expirée.

3. Gestion de l'authentification et Déconnexion :

- adminPanel permet à l'administrateur de se déconnecter en cliquant sur un bouton.
- La déconnexion entraîne la suppression du jeton d'authentification du stockage local (grâce à l'authContext via useAuth !) et redirige l'utilisateur vers la page d'authentification.

4. Récupération des Utilisateurs :

- Le composant utilise la fonction `fetchUsers` pour récupérer la liste des utilisateurs depuis l'API.

5. Création d'un Nouvel Utilisateur :

- L'administrateur peut créer un nouvel utilisateur en fournissant un e-mail, un rôle, et éventuellement un mot de passe (si le rôle choisi est ADMIN).
- adminPanel utilise `axios` pour effectuer une requête POST à l'API à l'endpoint '/insertUser'.

6. Affichage et Suppression des Utilisateurs :

- Le composant affiche une liste des utilisateurs avec leurs e-mails, rôles, et dates de création.
- L'administrateur peut supprimer un utilisateur en cliquant sur un bouton associé. Cela se fait grâce à handleDeleteUser via l'endpoint '/deleteUser'

7. Mise à Jour du Rôle d'un Utilisateur :

- L'administrateur peut modifier le rôle d'un utilisateur en cliquant sur un bouton.
- Cela ouvre une fenêtre modale (**Rodal**) permettant à l'administrateur de sélectionner un nouveau rôle et, éventuellement, un nouveau mot de passe.
- La mise à jour du rôle est effectuée à l'aide d'une requête PUT à l'API. Via l'endpoint '/updateRoles'

8. Affichage Dynamique :

- L'affichage du formulaire de création d'utilisateur, du tableau des utilisateurs, et de la fenêtre modale de mise à jour du rôle peut être activé/désactivé dynamiquement en fonction des actions de l'administrateur.

adminPanel utilise également des bibliothèques tierces telles que `@material-ui`, `antd`, et `rodal` pour améliorer l'expérience utilisateur.

Passons maintenant côté serveur pour expliquer le CRUD :

Comme démontré plus tôt, mon composant adminPanel effectue des call api à différents endpoints côté serveur afin de créer, voir, modifier ou supprimer des données.

• Récupération des Utilisateurs :

```
app.get('/users', (req, res) => {
  console.log('Requête vers /users reçue.');

  connection.query('SELECT * FROM users ORDER BY created_At DESC', (err, results) => {
    if (err) {
      res.status(500).json({ error: 'Erreur lors de la récupération des utilisateurs' });
    } else {
      const sanitizedEmails = results.map(user => sanitizeString(user.email));
      res.json(sanitizedEmails);
    }
  });
});
```

L'endpoint **GET /users** permet de récupérer la liste des utilisateurs depuis la base de données. Les résultats sont triés par date de création décroissante. Avant de les envoyer au client, les données sont également nettoyées à l'aide de la fonction **sanitizeString** pour éviter toute injection de code malveillant. **sanitizeString** est simplement une fonction qui retourne la bibliothèque **xss**.

- Mise à Jour du Rôle d'un Utilisateur :

```
app.put('/updateRoles', (req, res) => {
  let { email, newRole, newPassword } = req.body;

  email = sanitizeString(email);
  newRole = sanitizeString(newRole);
  newPassword = sanitizeString(newPassword);

  if (!email || !newRole) {
    return res.status(400).json({ error: 'Email et nouveau rôle sont requis.' });
  }

  let query = 'UPDATE users SET role = ?';
  let queryParams = [newRole, email];

  // Si j'ai select "ADMIN" dans le form
  if (newRole === "ADMIN" && newPassword) {

    bcrypt.hash(newPassword, 10, (err, hashedPassword) => {
      if (err) {
        console.error('Erreur lors du hachage du mot de passe :', err);
        return res.status(500).json({ error: 'Erreur lors du hachage du mot de passe.' });
      }

      query += ', password = ?'; // je concatene ", password = ?" à ma query
      queryParams = [newRole, hashedPassword, email];

      console.log("email " + email)
      console.log("newRole " + newRole)
      console.log("newPassword " + newPassword)
      console.log("queryParams " + queryParams)
      console.log('query ' + query)

      // à ce stade j'ai query qui est égal à UPDATE users set role = ? (ce sera newRole) ensuite je concatene ', password ?' (ce sera hashedPass)
      // pareil pour email à la fin de la query. C'est pour ça que l'ordre est important dans queryParams
      connection.query(query + ' WHERE email = ?', queryParams, (err, results) => {
        if (err) {
          console.error('Erreur lors de la mise à jour du rôle et du mot de passe de l\'utilisateur :', err);
          return res.status(500).json({ error: 'Erreur lors de la mise à jour du rôle et du mot de passe de l\'utilisateur.' });
        }

        if (results.affectedRows === 0) {
          return res.status(404).json({ error: 'Utilisateur non trouvé.' });
        }

        res.json({ message: 'Rôle et mot de passe de l\'utilisateur mis à jour avec succès.' });
      });
    });
  }
} else {
  // Sinon je met juste à jour son rôle (si j'ai pas select admin dans le form)
  console.log("QUERY : " + query, "queryParams : " + queryParams)
  connection.query(query + ' WHERE email = ?', queryParams, (err, results) => {
    if (err) {
      console.error('Erreur lors de la mise à jour du rôle de l\'utilisateur :', err);
      return res.status(500).json({ error: 'Erreur lors de la mise à jour du rôle de l\'utilisateur.' });
    }

    if (results.affectedRows === 0) {
      return res.status(404).json({ error: 'Utilisateur non trouvé.' });
    }

    res.json({ message: 'Rôle de l\'utilisateur mis à jour avec succès.' });
  });
}
});
```

L'endpoint **PUT /updateRoles** gère la mise à jour du rôle d'un utilisateur. Les données envoyées depuis le formulaire du client incluent l'e-mail de l'utilisateur, le nouveau rôle, et éventuellement le nouveau mot de passe si le rôle choisi est 'ADMIN' grâce à une query dynamique. Le processus inclut la mise à jour du rôle dans la base de données, et si nécessaire, le hachage du nouveau mot de passe à l'aide de bcrypt.

- **Création d'un Nouvel Utilisateur :**

```
app.post('/insertUser', (req, res) => {
  let { email, role, password } = req.body;
  console.log(email, role)
  email = sanitizeString(email);
  role = sanitizeString(role);
  password = sanitizeString(password)

  if (!email || !role) {
    return res.status(400).json({ error: 'Email et rôle sont requis.' });
  }

  bcrypt.hash(password, 10, (err, hashedPassword) => {
    if (err) {
      console.error('Erreur lors du hachage du mot de passe :', err);
      return res.status(500).json({ error: 'Erreur lors du hachage du mot de passe.' });
    }

    const query = 'INSERT INTO users (email, role, password) VALUES (?, ?, ?)';
    connection.query(query, [email, role, hashedPassword], (dbErr) => {
      if (dbErr) {
        console.error('Erreur lors de l\'insertion de l\'utilisateur :', dbErr);
        return res.status(500).json({ error: 'Erreur lors de l\'insertion de l\'utilisateur.' });
      }

      res.json({ message: 'Utilisateur inséré avec succès!' });
    });
  });
});
```

L'endpoint **POST /insertUser** est utilisé pour créer un nouvel utilisateur. Les données telles que l'e-mail, le rôle et le mot de passe (haché) sont incluses dans le corps de la requête. Avant l'insertion, le mot de passe est haché à l'aide de bcrypt.

- **Suppression d'un Utilisateur :**

```
0
1 app.delete('/deleteUser/:email', (req, res) => {
2   const email = sanitizeString(req.params.email);
3
4   if (!email) {
5     return res.status(400).json({ error: 'Email de l\'utilisateur à supprimer requis.' });
6   }
7
8   const query = 'DELETE FROM users WHERE email = ?';
9   connection.query(query, [email], (err, results) => {
0
1     if (err) {
2       console.error('Erreur lors de la suppression de l\'utilisateur :', err);
3       return res.status(500).json({ error: 'Erreur lors de la suppression de l\'utilisateur.' });
4     }
5
6     res.json({ message: 'Utilisateur supprimé avec succès!' });
7   });
8 });
9
```

L'endpoint **DELETE /deleteUser/:email** permet de supprimer un utilisateur spécifié par son e-mail. L'e-mail est extrait des paramètres de l'URL. La suppression est effectuée dans la base de données.

Chaque endpoint gère les erreurs potentielles telles que des données manquantes, des erreurs de base de données, et assure une réponse appropriée au client. De plus, des mesures de sécurité supplémentaires sont mises en place pour prévenir les attaques SQL/XSS. Les requêtes SQL utilisent des paramètres dynamiques représentés par des '?' dans les query et les données provenant du DTO (Data Transfer Object) sont « nettoyées » via la fonction 'sanitizedString', réduisant ainsi le risque d'injections SQL/XSS en empêchant l'interprétation malveillante des entrées utilisateur.

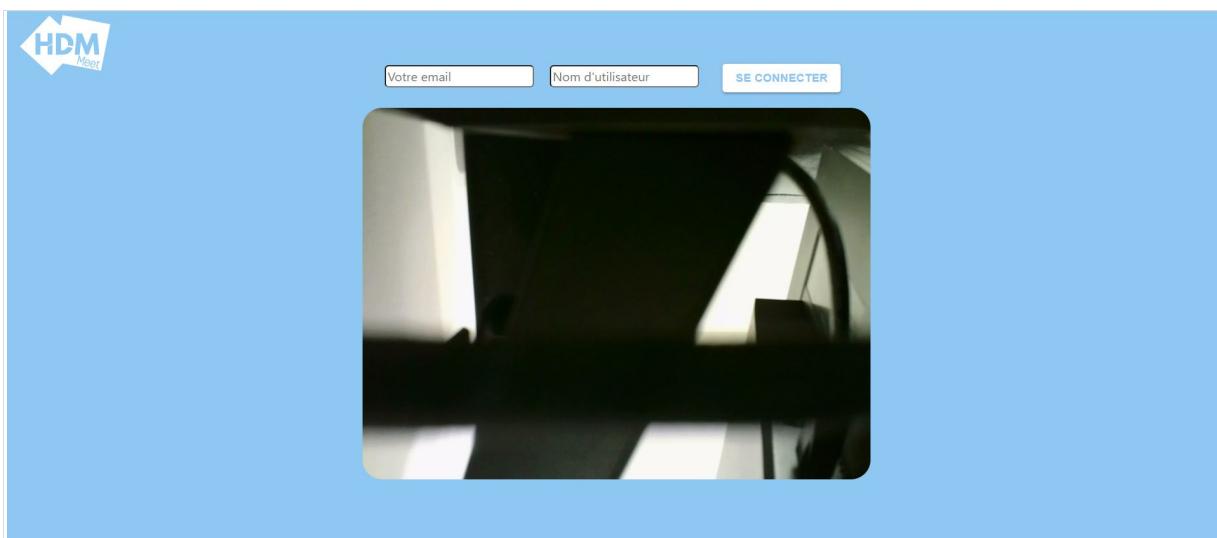
8. UTILISATION DE PROGRAMME

A quoi ressemble HDM MEET ?

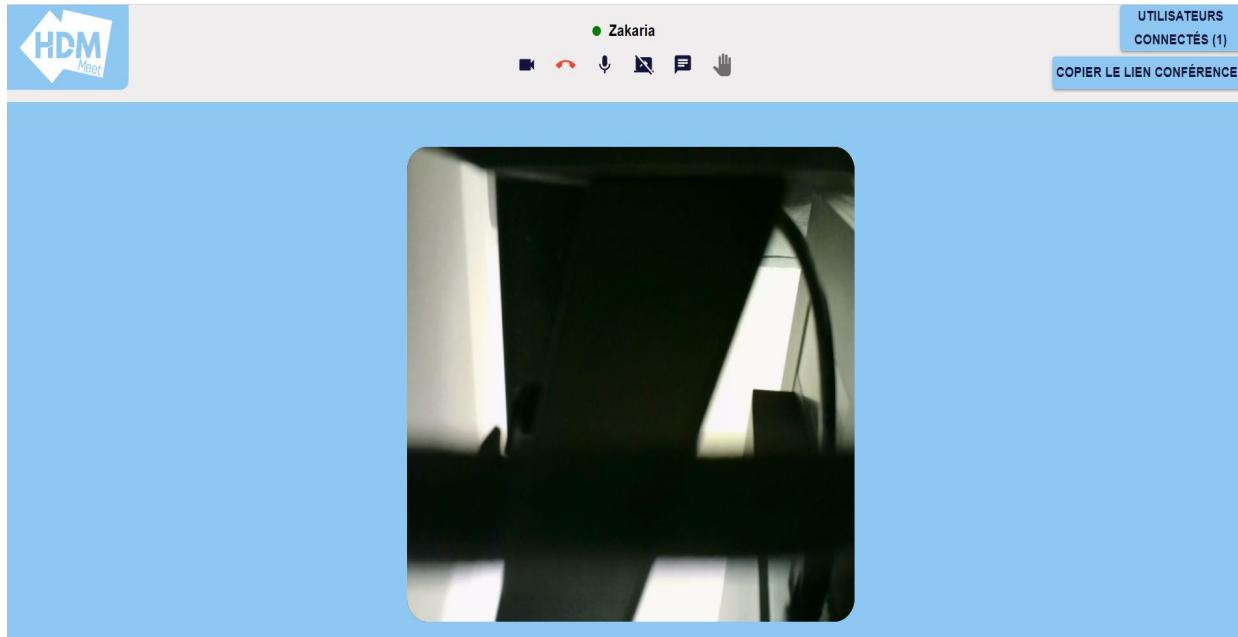
Page d'accueil :



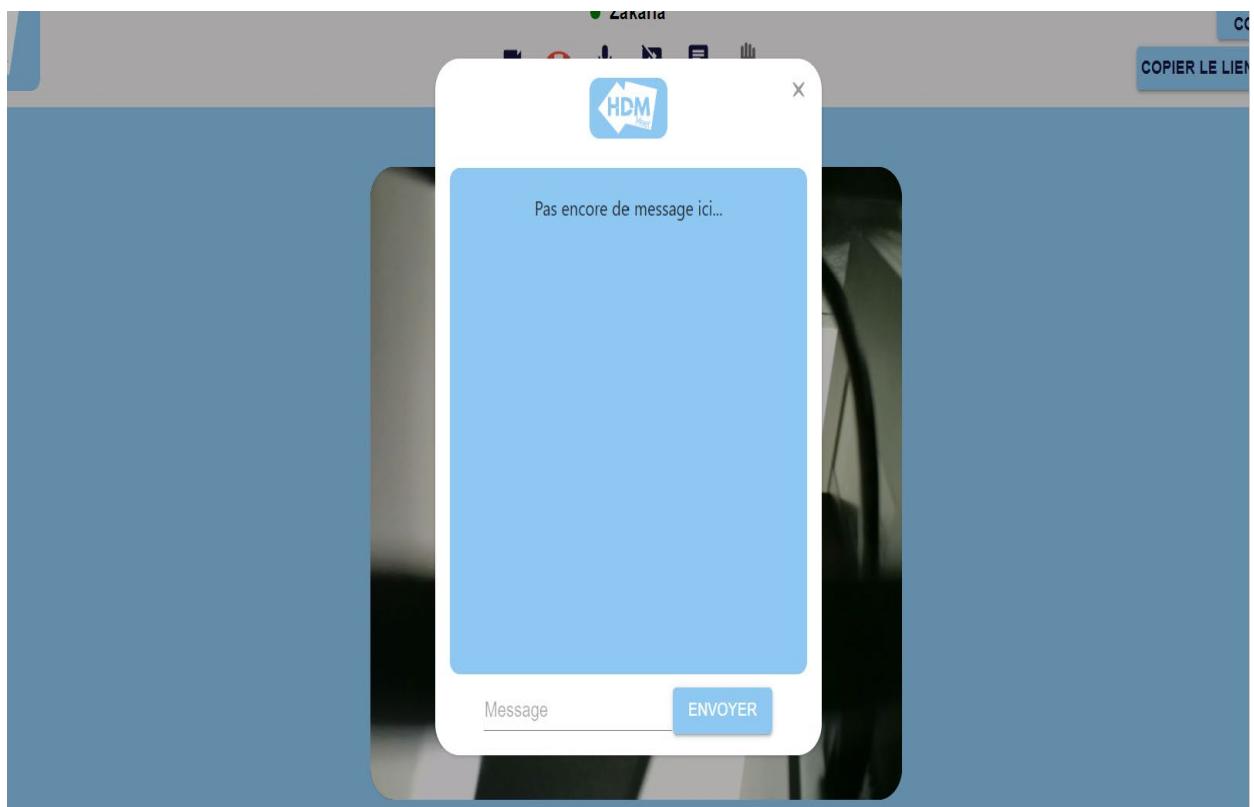
En cliquant sur « Commencer », on atterrit ici :



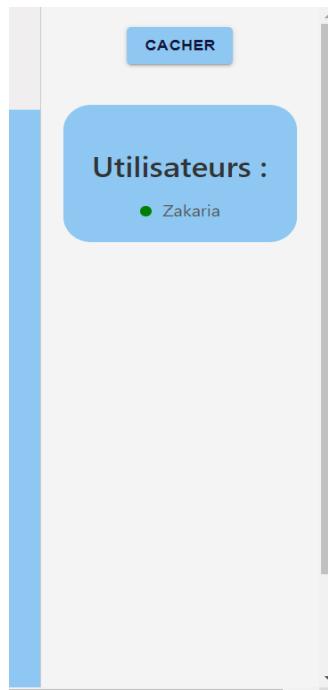
Après avoir entré ses identifiants, on atterrit dans la vidéo conférence :



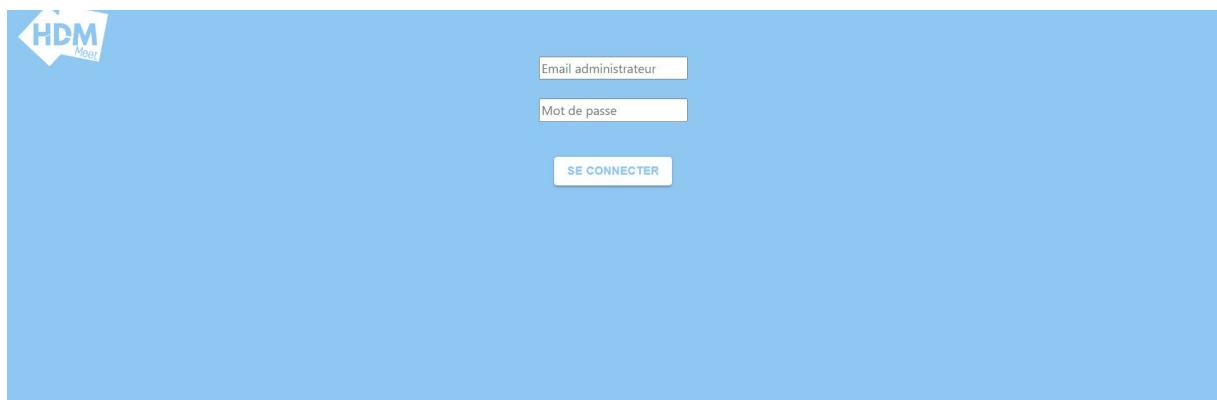
En cliquant sur on ouvre la fenêtre modale de chat :



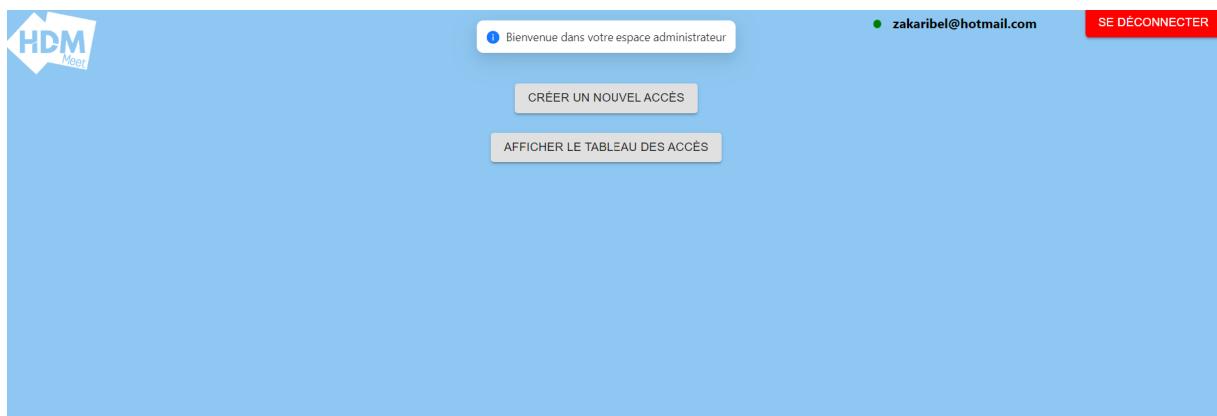
Lorsque l'on clique sur « Utilisateurs connectés » une sidebar apparaît :



Revenons à la page d'accueil et cette fois-ci, si on clique sur « Administration », on atterrit sur la page d'authentification administrateur :



Une fois authentifié on accède au panel admin :



Si on clique sur "créer un nouvel accès" le formulaire correspondant apparaît :

CACHER LE FORMULAIRE

Email du nouvel utilisateur:

Rôle du nouvel utilisateur:

Créer l'utilisateur

AFFICHER LE TABLEAU DES ACCÈS

Si on clique sur "afficher le tableau des accès", voici ce qui apparaît :

CACHER LE TABLEAU

Email	Rôle	Date de création	Action
richard@gmail.com	ADMIN	19/01/2024 11:06	Supprimer Modifier le rôle
carmela@gmail.com	ADMIN	18/01/2024 21:57	Supprimer Modifier le rôle
quentin@gmail.com	ADMIN	12/01/2024 11:30	Supprimer Modifier le rôle
jennifer@gmail.com	ADMIN	12/01/2024 11:19	Supprimer Modifier le rôle
zakaribel@hotmail.com	ADMIN	07/01/2024 12:05	Supprimer Modifier le rôle

Si on clique sur modifier le rôle :

Email
richard@gmail.com
carmela@gmail.com
quentin@gmail.com
jennifer@gmail.com
zakaribel@hotmail.com

Modifier le rôle de l'utilisateur

richard@gmail.com

Nouveau rôle :

Modifier le rôle
Supprimer

Fermer

Action

Supprimer	Modifier le rôle
Supprimer	Modifier le rôle
Supprimer	Modifier le rôle
Supprimer	Modifier le rôle
Supprimer	Modifier le rôle

9. TESTS

TEST	Affichage	Création	Modification	Suppression
Gestion des utilisateurs (panel administration)	OK	OK	OK	OK

TEST	Affichage flux vidéo	Envoi signal SDP/IceCandidates côté client	Reception signal SDP/IceCandidates côté client
Gestion de communication webRTC via socket-io	OK	OK	OK

TEST	Envoi msg via émission socket	Réception msg (socket.on)	Affichage messages dans le DOM
Gestion du chat	Ok	OK	Ok

10. LA VEILLE TECHNIQUE

Sites Web et Blogs Technologiques :

StackOverflow, Github Community, Documentation React.

StackOverflow :

<https://stackoverflow.com/questions/70469717/cant-load-a-react-app-after-starting-server>

In file: node_modules/react-scripts/config/webpackDevServer.config.js

```

like this

onBeforeSetupMiddleware(devServer) {
  // Keep `evalSourceMapMiddleware` 
  // middlewares before `redirectServedPath` otherwise will not have any effect
  // This lets us fetch source contents from webpack for the error overlay
  devServer.app.use(evalSourceMapMiddleware(devServer));

  if (fs.existsSync(paths.proxySetup)) {
    // This registers user provided middleware for proxy reasons
    require(paths.proxySetup)(devServer.app);
  }
},
onAfterSetupMiddleware(devServer) {
  // Redirect to 'PUBLIC_URL' or 'homepage' from 'package.json' if url not match
  devServer.app.use(redirectServedPath(paths.publicUrlOrPath));

  // This service worker file is effectively a 'no-op' that will reset any
  // previous service worker registered for the same host:port combination.
  // We do this in development to avoid hitting the production cache if
  // it used the same host and port.
  // https://github.com/facebook/create-react-app/issues/2272#issuecomment-3028324
  devServer.app.use(noopServiceWorkerMiddleware(paths.publicUrlOrPath));
}

change to

setupMiddlewares: (middlewares, devServer) => {
  if (!devServer) {
    throw new Error('webpack-dev-server is not defined')
  }

  if (fs.existsSync(paths.proxySetup)) {
    require(paths.proxySetup)(devServer.app)
  }

  middlewares.push(
    evalSourceMapMiddleware(devServer),
    redirectServedPath(paths.publicUrlOrPath),
    noopServiceWorkerMiddleware(paths.publicUrlOrPath)
  )
  return middlewares;
}

```

merci !

Github Community :

<https://github.com/tinode/chat/issues/766>

ICE, STUN & TURN included #766
rkgarci opened this issue on Jun 20, 2022 · 4 comments

pwFoo commented on Jan 17, 2023

```

For now I just added a public stun server...

turn-config.json
[
  {
    "urls": [
      "stun:stun.l.google.com"
    ]
  }
]

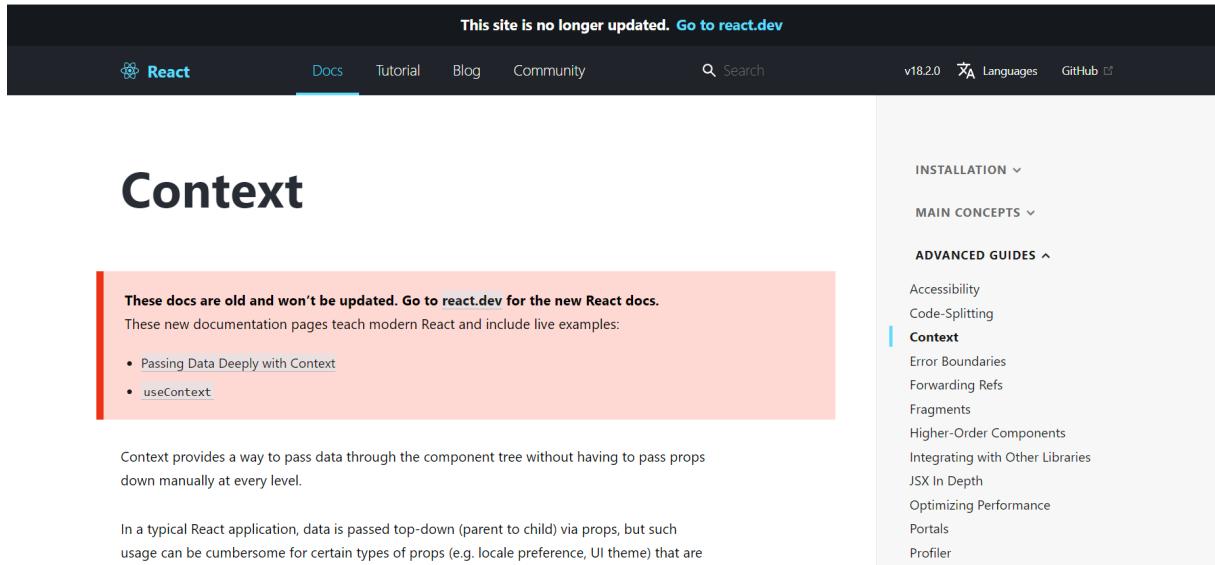
```

But I'll take a look into coturn or something else in the future with a little bit more time. Any suggestion about a good / simple self hosted stun / coturn service?

or-else commented on Jan 17, 2023

Documentation React :

<https://legacy.reactjs.org/docs/context.html>



This site is no longer updated. Go to react.dev

React Docs Tutorial Blog Community v18.2.0 Languages GitHub

Context

These docs are old and won't be updated. Go to react.dev for the new React docs.

These new documentation pages teach modern React and include live examples:

- Passing Data Deeply with Context
- useContext

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are

INSTALLATION ▾

MAIN CONCEPTS ▾

ADVANCED GUIDES ▾

- Accessibility
- Code-Splitting
- Context
- Error Boundaries
- Forwarding Refs
- Fragments
- Higher-Order Components
- Integrating with Other Libraries
- JSX In Depth
- Optimizing Performance
- Portals
- Profiler

11.

BILAN

Le projet *HDM MEET* que j'ai mené au sein d'HDM NETWORK consistait à développer une application de visioconférence utilisant les technologies Node.js/Express, React, WebRTC, et Socket.io. Cette expérience a été extrêmement enrichissante, m'incitant à sortir de ma zone de confort et à relever des défis techniques stimulants. Grâce à ce projet, j'ai acquis de nouvelles compétences, renforcé ma maîtrise des technologies webRTC et socket.io, et développé une expertise dans la création d'applications collaboratives en temps réel.

Ce parcours m'a non seulement permis de consolider mes connaissances en développement full-stack, mais il a également renforcé ma confiance dans la résolution de problèmes complexes. En fin de compte, je suis sorti de cette expérience non seulement plus fort techniquement, mais également plus à l'aise dans la gestion de projets d'envergure et fier des résultats que j'ai obtenus. Cette réussite atteste de ma capacité à mener à bien des projets ambitieux et stimulants.