Autorisation de certaines actions de l'utilisateur

L'autorisation est le processus qui détermine si un utilisateur a les droits nécessaires pour effectuer une action. Par exemple, un utilisateur peut avoir le droit de modifier ses propres données, mais pas celles des autres utilisateurs. L'autorisation est souvent basée sur le rôle de l'utilisateur, qui est déterminé lors de la connexion.

Enregistrer le token dans le local storage dans le front-end

Lorsqu'un utilisateur se connecte, on fait une demande à l'api. Si la connexion est bonne, celle-ci renvoie le jeton dans la réponse. On peut enregistrer le token dans le local storage du navigateur. Cela permet de garder le token en mémoire et de l'envoyer avec chaque requête.

```
// Dans le front-end REACT
const handleLogin = async (e) => {
    e.preventDefault(); // Empêche le rechargement de la page

    const formulaire = e.target;

    const courriel = formulaire.courriel.value;
    const mdp = formulaire.mdp.value;

    try {
        const response = await
fetch("http://localhost:3000/api/utilisateurs/connexion", {
            method: "POST",
            headers: {
                "Content-Type": "application/json",
            },
            body: JSON.stringify({ courriel, mdp }),
        });

        const data = await response.json();

        if (response.status === 200) {
            //Mettre un nom significatif pour le token
            localStorage.setItem("api-token", data.token);
        } else {
            console.log(data.message);
        }
    } catch (error) {
        console.log(error);
    }
};
```

```
// Dans le back-end Express
const jwt = require("jsonwebtoken");

router.post(
    "/connexion",
    [
        //Validation des données avec express-validator
    ],
    async (req, res) => {
        try {
            const motDePasse = req.body.mdp;
            const courriel = req.body.courriel;

            const docRef = await db.collection("utilisateurs").where("courriel",
"==", courriel).get();
            const utilisateurs = [];

            docRef.forEach(async (doc) => {
                utilisateurs.push({ id: doc.id, ...doc.data() });
            });

            const utilisateur = utilisateurs[0];

            if (utilisateur === undefined) {
                res.statusCode = 400;
                res.json({ message: "Le courriel n'existe pas" });
            } else {
                const resultatConnexion = await bcrypt.compare(motDePasse,
utilisateur.mdp);

                if (resultatConnexion) {
                    token = jwt.sign({ id }, process.env.JWT_SECRET, { expiresIn:
"1d" });
                    res.json(token);
                } else {
                    res.statusCode = 400;
                    res.json({ message: "Mot de passe incorrect" });
                }
            }
        } catch (err) {
            console.log(err);
            res.status(500).send(err);
        }
    }
);
```

Décoder le token dans le front-end

Souvent, il est utile de décoder le token pour obtenir les informations qu'il contient. Cela permet de voir si l'utilisateur est connecté et d'afficher son nom dans le front-end.

Par exemple, on peut décoder le token pour obtenir l'ID de l'utilisateur et afficher son nom dans le front-end. Ne jamais mettre des informations sensibles dans le token, car il est toujours possible de le décoder mais il est impossible de le réencoder car il est signé avec une clé secrète qui est uniquement connue du serveur.

Le jeton contient toujours le paramètre `exp` qui est la date d'expiration du jeton. On peut comparer cette date avec la date actuelle pour voir si le jeton est toujours valide. La date d'expiration est enregistrée en secondes, donc on doit la multiplier par 1000 pour obtenir la date en millisecondes.

Pour décoder le jeton, on peut utiliser la méthode `decode` de jwt. Il faut installer la librairie `jwt-decode` pour pouvoir l'utiliser dans React.

```
npm install jwt-decode
```

```
import jwtDecode from "jwt-decode";

function jetonValide() {
    try {
        // On récupère le token du local storage
        const token = localStorage.getItem("api-film-token");
        const decoded = jwtDecode(token);

        // On vérifie si le token est toujours valide
        if (Date.now() < decoded.exp * 1000) {
            return true;
        } else {
            // Si le token est expiré, on le supprime du local storage
            localStorage.removeItem("api-film-token");
            return false;
        }
    } catch (error) {
        return false;
    }
}
```

Créer des routes privées dans React

Pour créer des routes privées dans React, on peut englober un composant `Route` d'un composant personnalisé `PrivateRoute`. On peut passer des props à `PrivateRoute` pour vérifier si l'utilisateur est connecté et s'il a les autorisations nécessaires pour accéder à la route.

```
```

```
<Route element={<PrivateRoute estConnecte={estConnecte} />}>
    <Route path="/admin/utilisateurs" element={<AdminUtilisateurs />} />
    <Route path="/admin/liste-films" element={<AdminListeFilms />} />
</Route>
```

Le composant `PrivateRoute` peut vérifier si l'utilisateur est connecté et s'il a les autorisations nécessaires pour accéder à la route. Si c'est le cas, il affiche le composant passé en paramètre. Sinon, il redirige l'utilisateur vers une autre page.

Si plusieurs routes sont englobées par `PrivateRoute`, on peut utiliser un composant `Outlet` pour afficher les routes enfants. Le composant `Outlet` prend la forme de la route enfant choisie par l'utilisateur.

```
import React from "react";
import { Navigate, Outlet } from "react-router-dom";

const PrivateRoute = ({ estConnecte }) => {
    try {
        if (estConnecte) {
            return <Outlet />;
        } else {
            return <Navigate to="/" />;
        }
    } catch (error) {
        return <Navigate to="/" />;
    }
};

export default PrivateRoute;
```

Afficher conditionnellement des éléments dans le front-end si l'utilisateur est connecté

Pour afficher conditionnellement des éléments dans le front-end si l'utilisateur est connecté, on peut utiliser une condition ternaire. Si l'utilisateur est connecté, on affiche un bouton pour se déconnecter. Sinon, on affiche un bouton pour se connecter.

```
const [estConnecte, setConnexion] = useState(false);

useEffect(() => {
    if (localStorage.getItem("api-film-token")) {
        setConnexion(jetonValide());
    }
}, []);
```

On peut ensuite passer la variable `estConnecte` à la navigation pour afficher des éléments du menu conditionnellement.

```
<nav>
    {estConnecte ? (
        <NavLink to="/admin" className="underline">
            Admin
        </NavLink>
    ) : (
        ""
    )}
    <NavLink to="/liste-films" className="underline">
        Liste des films
    </NavLink>
</nav>
```

Se déconnecter et supprimer le token du local storage

Pour se déconnecter, on peut simplement supprimer le token du local storage. Cela permet de déconnecter l'utilisateur et de le rediriger vers la page de connexion.

```
const handleDeconnexion = () => {
    localStorage.removeItem("api-film-token");
    setConnexion(false);
};
```

Optimiser le passage des données avec UseContext

Pour optimiser le passage des données dans le front-end, on peut utiliser le hook `useContext` de React. Cela permet de passer des données à travers plusieurs composants sans avoir à les passer en props à chaque composant.

À la racine du projet, on crée un contexte qui contient les données à passer. On doit exporter ce contexte pour pouvoir l'utiliser dans d'autres composants.

```
import React, { createContext, useState } from "react";

export const AuthContext = createContext();
```

Pour l'utiliser, le contexte doit englober tous les composants qui ont besoin d'accéder aux données. Par exemple, autour du routeur principal, on ajoute un `Provider` qui contient les données à passer.

Le provider possède une propriété `value` qui contient les données à passer. On peut passer des fonctions, des objets ou des variables.

```
return (
    <AppContext.Provider value={estConnecte}>
        <Router>
            <Entete handleLogin={login} />
            <Routes>
                <Route element={<PrivateRoute />}>
                    <Route path="/admin" element={<FormFilms />} />
                </Route>

                <Route path="/" element={<Accueil />} />
                <Route path="/film/:id" element={<Film />} />
                <Route path="/liste-films" element={<ListeFilms />} />
                <Route path="/*" element={<Accueil />} />
            </Routes>
        </Router>
    </AppContext.Provider>
);
```

Utiliser le contexte dans un composant

Pour utiliser le contexte dans un composant, on peut utiliser le hook `useContext` de React. Cela permet de récupérer les données passées par le contexte. On doit également importer l'objet contenant le contexte pour pouvoir l'utiliser.

```
import React, { useContext } from "react";
import { AppContext } from "../App/App";

function Entete() {
    const estConnecte = useContext(AppContext);
    return (
        <header>
            <h1>Mon application</h1>
            <nav>
                {estConnecte ? (
                    <NavLink to="/admin" className="underline">
                        Admin
                    </NavLink>
                ) : (
                    ""
                )}
                <NavLink to="/liste-films" className="underline">
                    Liste des films
                </NavLink>
```

```
            </nav>
        </header>
    );
}
```