Gestion des authentifications et autorisations des utilisateurs avec JWT

Authentification schéma des entetes d'autorisation

Lorsqu'un utilisateur se connecte, on peut générer un jeton pour lui permettre d'accéder à certaines routes. Le d'authentification qui est émis par le serveur permet de limiter à un client ou un type de client (admin vs utilisateur régulier) d'accéder à une ressource protégée. Le jeton d'accès est utilisé pour authentifier un utilisateur et il est transmis avec chaque requête.

Le jeton ou token est généré avec la méthode `sign` de la librairie `Json Web token` ou `jwt`. Celle-ci prend en paramètre un objet contenant les informations à stocker dans le token, une clé secrète et une durée de validité du token.

Il existe plusieurs schema d'autorisation, mais le plus courant est le schema Bearer. Le schema Bearer est utilisé pour les jetons d'accès OAuth 2.0 et JWT.

Pour que l'autorisation fonctionne, le mot-clé Bearer doit être suivi d'un espace, puis du jeton d'accès.

Installation de jwt au niveau de l'API Express

```
npm install jsonwebtoken
```

Générer le jeton après la connexion

Lorsqu'un utilisateur se connecte, on peut générer un token pour lui permettre d'accéder à certaines routes. Le token est généré avec la méthode `sign` de jwt.

Celle-ci prend en paramètre un objet contenant les informations à stocker dans le token, une clé secrète et une durée de validité du token. On peut stocker l'ID de l'utilisateur dans le token pour pouvoir l'identifier plus tard, son nom, son rôle (si c'est un super admin), etc. On ne met pas de données sensibles dans le token, car il est toujours possible de le décoder.

La durée peut être spécifiée en secondes, minutes, heures, jours, mois, années. Par exemple, pour spécifier une durée de 1 heure, on peut utiliser `1h`, pour 1 jour, on peut utiliser `1d`, pour 1 mois, on peut utiliser `1M`, pour 1 an, on peut utiliser `1y`.

Règle générale, on utilise une durée de validité courte pour les tokens, par exemple 1 jour. Cela permet de limiter les risques en cas de vol du token. Si le token est volé, il ne sera valide que pour une courte durée.

```
const jwt = require("jsonwebtoken");
```

```
const token = jwt.sign({ id: user._id, role: "admin" }, process.env.JWT_SECRET, {
    expiresIn: "1d",
});

fetch("http://localhost:3000/api/utilisateurs", {
    method: "GET",
    headers: {
        "Content-Type": "application/json",
        Authorization: `Bearer ${token}`,
    },
});
```

Vérifier le jeton

Pour vérifier le token, on peut utiliser la méthode `verify` de jwt. Celle-ci prend en paramètre le token et la clé secrète. Si le token est valide, elle retourne l'objet stocké dans le token.

Ceci est sécuritaire car le token est signé avec la clé secrète qui est uniquement connue du serveur. Si le token a été modifié, la signature ne correspondra plus et la méthode `verify` lèvera une erreur.

```
try {
    const decodedToken = jwt.verify(token, process.env.JWT_SECRET);
} catch (error) {
    console.log(error);
}
```

Créer une fonction middleware pour vérifier le token

On peut créer une fonction middleware réutilisable pour vérifier le token et autoriser l'accès à certaines routes. Le middleware se place entre la route et la fonction de callback. Il prend en paramètre `req`, `res` et `next`.

La fonction sert à vérifier si le token est présent dans le header de la requête, s'il est valide et s'il correspond à un utilisateur dans la base de données. Si tout est bon, il appelle la fonction `next` pour passer à la suite. En appelant `next()`, Express passe à la route qui était demandée.

La stratégie de vérification du token est la suivante :

- On vérifie si le token est présent dans le header de la requête et s'il commence par "Bearer ".
- On extrait le token de l'entête de la requête.
- On vérifie si le token est valide avec la méthode `verify` de jwt.
- On vérifie si l'utilisateur correspondant au token existe dans la base de données.

Si tout est bon, on appelle la fonction `next` pour passer à la suite. Sinon, on retourne une erreur.

```
```

```
const jwt = require("jsonwebtoken");
const db = require("../config/db.js");

const auth = async (req, res, next) => {
    try {
        if (req.headers.authorization &&
req.headers.authorization.startsWith("Bearer ")) {
            const token = req.headers.authorization.split(" ")[1];
            const decodedToken = jwt.verify(token, process.env.JWT_SECRET);

            const docRef = await
db.collection("utilisateurs").doc(decodedToken.id).get();

            if (!docRef.exists) {
                throw "Non autorisé";
            } else {
                req.user = docRef.data();
                next();
            }
        } else {
            throw "Non autorisé";
        }
    } catch (error) {
        res.statusCode = 401;
        res.json({ message: "Non autorisé" });
    }
};
```

Protéger les routes avec le middleware

Pour protéger une route, on peut utiliser le middleware créé précédemment. On l'ajoute en deuxième paramètre de la route.

Par exemple, pour s'assurer que seulement un utilisateur connecté puisse accéder à la liste des utilisateurs, on peut utiliser le middleware `auth` et vérifier si l'utilisateur est connecté.

Nous n'avons seulement besoin que de placer le middleware `auth` en deuxième paramètre de la route pour la protéger. Le middleware `auth` vérifie si l'utilisateur est connecté et autorise l'accès à la route si c'est le cas. Sinon, le reste de la route ne sera pas exécuté.

De manière générale, on authentifie toutes les routes qui nmodifient une ressource ou accédent à des informations sensibles, comme un compte utilisateur ou la liste de tous les utilisateurs.

```
const auth = require("../middlewares/auth.js");

router.post("/", auth, async (req, res) => {
```

```
    // ... Le reste du code de la route,
});
```