Sécuriser une application dorsale

La sécurité d'une application chargée de gérer des données sensibles comme les nom et prénom des agents d'une unité est fondamentale. La décision prise lors du premier TD sur l'authentification de ne pas assujettir les routes *GET* au contrôle d'accès est fortement discutable. Elle sera réévaluée au cours de ce TD pour offrir au final une application complètement sécurisée.

D'un point de vue pratique, vous allez mettre en place un mécanisme pour ajouter des utilisateurs en chiffrant leurs mots de passe à l'aide du module *Bcrypt* et pour leur permettre de s'authentifier à l'aide de leurs identifiants (email et mot de passe) afin de naviguer à travers les routes de l'application. Le processus d'authentification sera complexifié grâce au recours au module *JSONWebToken* chargé de générer un jeton (*token*) à partir d'une phrase secrète si les identifiants renseignés par l'utilisateur sont cohérents.

Étape 1 : lister les utilisateurs

Compléter le modèle des utilisateurs

Complétez le module *models/users.js* avec des méthodes pour :

- lister tous les utilisateurs ;
- obtenir le détail d'un utilisateur par son email.

```
/* GET all users */
async function getAllUsers() {

    const results = await db.query(
        `SELECT email, password
        FROM users
        ORDER BY email ASC;`
    );

    return results;
};

/* GET user by email */
async function getUserByEmail(email) {

    const results = await db.query(
        `SELECT id_user, password
        FROM users
        WHERE email = ?;`,
        [email]
    );

    return results[0];
};
```

Et n'oubliez pas de les exporter via `module.exports` !

Un contrôleur pour lister les utilisateurs

Créez un fichier *users.js* dans le répertoire *controllers* et définissez deux fonctions asynchrones pour d'une part lister tous les utilisateurs et d'autre part obtenir les informations d'un seul d'entre eux :

```
const users = require('../models/users');

const getUsers = async (req, res, next) => {
    try {
        res.send(await users.getAllUsers());
    } catch (err) {
        next(err);
    }
};

const getUser = async (req, res, next) => {
    try {
        res.send(await users.getUserById(req.params.id_user));
    } catch (err) {
        next(err);
    }
};

module.exports = {
    getUsers,
    getUser
};
```

Tracer les routes

Créez à présent un fichier *users.js* dans le répertoire *routes* et tracez les routes *ad hoc* pour lister les utilisateurs :

```
const express = require('express');
const router = express.Router();
const users = require('../controllers/users');

router.get('/', users.getUsers);
router.get('/:id_user', users.getUser);

module.exports = router;
```

Il ne vous reste plus qu'à importer ce routeur dans le fichier principal *app.js* et à l'utiliser :

```
var usersRouter = require('./routes/users');

app.use('/users', usersRouter);
```

Étape 2 : ajouter un utilisateur

Dansun premier temps, écrivez dans le modèle *users.js* une méthode asynchrone `addOneUser()` pour insérer un utilisateur dans la base de données qui accepte deux paramètres : un email et un mot de passe. Exportez cette méthode à la fin du module.

Dans un second temps, écrivez un contrôleur `addUser()` qui fasse appel à cette méthode et transmettez-lui les paramètres présents dans l'objet `req.body` en prenant soin de chiffrer préalablement le mot de passe avec la méthode `hash()` du module *Bcrypt* :

```
const password = await bcrypt.hash(req.body.password, 10);
```

Comme ce contrôleur utilise une librairie externe, vous devez d'abord l'installer avec *npm* :

```
npm install bcrypt --save
```

Et l'importer dans le fichier :

```
const bcrypt = require('bcrypt');
```

Enfin, définissez une route *POST* `/add` pour l'insertion d'un utilisateur.

Étape 3 : authentifier un utilisateur

Établir la route d'accès

L'application frontale met à disposition un formulaire d'authentification avec un champ pour renseigner l'email et un autre pour le mot de passe qui redirige, à la soumission, vers une route `/users/login`. Paramétrez le routeur de telle sorte à capturer les données transmises à cette adresse et à les traiter avec une méthode `users.login()` :

```
router.post('/login', users.login);
```

Comparer les mots de passe

Dans la base de données, le mot de passe de l'utilisateur est enregistré sous une forme chiffrée par la fonction de hachage *Bcrypt*. Le module Node homonyme met opportunément à disposition une méthode `compare()` pour comparer la saisie en clair de l'utilisateur avec la donnée chiffrée. Mais avant de réaliser cette opération, vous vérifierez si l'email saisi correspond bien à un enregistrement de la table `users`.

Écrivez tout d'abord la structure de votre contrôleur `login` :

```
const login = async (req, res, next) => {
    try {
        // verify if the given email is in the database
        // if so, compare the hash with the given password
        // and, then, emit a token
    } catch (err) {
        next(err);
    }
```

Appelez la méthode `getUserByEmail()` pour chercher un utilisateur en fonction d'un mot de passe saisi :

```
const user = await users.getUserByEmail(req.body.email);
```

S'il n'existe aucun utilisateur pour l'email, renvoyez une erreur 401 :

```
if (!user) return res.status(401).send('Utilisateur non trouvé');
```

Dans les autres cas, vous pouvez comparer les mots de passe et, si les empreintes numériques (*hash*) ne correspondent pas, renvoyez une autre erreur 401 :

```
bcrypt.compare(req.body.password, user.password)
.then( result => {
    if (!result) return res.status(401).send('Mot de passe incorrect !');
    else {
        // send a token
    }
} );
```

Émettre un jeton d'authentification

L'émission d'un jeton d'authentification recourt à une autre librairie : *JSONWebToken*. Installez-la avec *npm* et sauvegardez-la dans le fichier *package.json* :

```
npm install jsonwebtoken --save
```

Importez-le ensuite comme tous les autres :

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

À l'intérieur de votre contrôleur `login`, une fois le mot de passe vérifié, émettez un objet avec la méthode `res.json()` dans lequel se trouvent deux clés : une pour transmettre l'identifiant de l'utilisateur et l'autre pour le *token* :

```
res.status(200).json({
    userId: user.id_user,
    token: 'A SIGNED TOKEN'
)});
```

Il est temps de passer à la dernière phase : signer le jeton d'authentification avec la méthode `jwt.sign()` :

```
res.status(200).json({
    userId: user.id_user,
    token: jwt.sign(
        { userId: user.id_user },
        'SECRET_TOKEN',
        { expiresIn: '24h' }
    )
)});
```

Remarque : le second paramètre de la méthode `jwt.sign()` doit contenir une phrase secrète bien plus complexe que celle de l'exemple.

Et n'oubliez pas d'exporter votre contrôleur !

Étape 4 : appliquer le contrôle d’accès

L’identification de l’utilisateur devrait maintenant fonctionner. Il reste à paramétrer un *middleware* qui vérifie l’identité de tout utilisateur essayant d’atteindre une route avec accès restreint. Ce contrôleur existe déjà dans le module *auth.js* : `checkAuth()`.

Renforcer le contrôle d’accès

Importez le module *JSONWebToken* et revoyez son squelette :

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

const checkAuth = async (req, res, next) => {
    try {
        // get the token
        // decode the token
        // extract the userId
        // check that the user exists in database
        // if not, throw an exception
        // otherwise process!
    } catch (err) {
        next(err);
    }
};
```

Comment récupérer le jeton d’authentification ?

La création, à l’étape précédente, d’un jeton d’authentification par le contrôleur `login` a eu pour effet d’ajouter un objet `Authorization` en tête de chaque requête en provenance de l’application frontale et avec, comme valeur de la clé `Bearer`, une longue chaîne encodée. C’est cette chaîne qu’il faut analyser pour récupérer le jeton :

```
const token = req.headers.authorization.split(' ')[1];
```

Et pour ensuite le décoder à l’aide de la méthode `jwt.verify()` :

```
const decodedToken = jwt.verify(token, 'SECRET_TOKEN');
```

Une fois le jeton décodé, vous pouvez récupérer la clé `userId` et vous en servir pour tenter de retrouver un utilisateur dans la base de données :

```
const user = await users.getUserById(decodedToken.userId);
```

Si les informations ne concordent pas, levez une exception. Dans les autres cas, passez simplement au *middleware* suivant !

```
if (req.body.userId && req.body.userId !== decodedToken.userId) {
    throw 'Invalid user ID';
} else {
    next();
}
```

Étape 5 : appliquer le contrôle d’accès

Suivez le modèle ci-dessous pour assujettir toutes les routes de votre application (à part celle réservée à l’authentification) au contrôle d’accès :

```
// import auth controller
const auth = require('../controllers/auth');

router.get('/', auth.checkAuth, users.getUsers);
```

Vous trouverez le code final dans le dossier *fin* de ce second TD.