NodeJS API - A3 - Authentification

Nous avons un utilisateur fonctionnel ! Ou presque... Il semblerait que notre mot de passe soit stocké en clair, ce qui vous le conviendrez, n'est pas fou !

Nous allons donc nous attaquer à un autre module de notre application, le module 'Auth'.

01 - Sécurité du mot de passe

L'avantage d'avoir découper notre application, il ne reste qu'à modifier notre service qui s'occupe de créer un utilisateur.

Pour s'occuper de la partie de hasher notre mot de passe, nous allons utiliser une librairie externe. Si votre serveur tourne, coupé le grâce au raccourcis `Ctrl+C`. Puis installez `bcrypt`

```
npm install bcrypt
```

Ensuite, il nous reste à modifier notre fonction qui crée les utilisateurs. Je vous donne un exemple de code et vous laisse l'analyser avec la doc pour comprendre :

```
let user ;
let userData = structuredClone(body); //Pour éviter de modifier le body
try {
    let hash = await bcrypt.hash(userData.password, 10)
    userData.password = hash;
    user = await userModel.create(userData);
} catch (error) {
    throw error
}
return user;
```

Vous pouvez tester votre route et normalement, en BD, votre utilisateur aura un champs password de ce type

`$2b$10$bsX7dAtPYCZs/QF9YFVswuzuvSdSGARxblDIwU4BZeWy4Z3XfcfQG`

Tous vos autres utilisateur n'auront plus d'intérêt quand nous implémenterons le login. Vous pourrez les supprimer.

Avant de continuer, mettez en place le module `Auth` avec un routeur/contrôleur qui auront pour endpoint `/auth`. Puis mettez en place une route POST `/register` qui reprends la création d'un utilisateur. Petit changement, nous ne retournerons pas l'userID, mais juste un message de validation (au format JSON évidemment).

02 - Connexion

Pour gérer la connexion de notre utilisateur, il faut qu'il ai accès à une route POST `/login`. Mettez en place le routeur et le contrôleur pour que cela fonctionne. Votre route devra retourner soit un message d'erreur que le combo email/password n'est pas correct, soit un JSON avec l'userID ainsi qu'un TOKEN. Pour le moment ce token ne serait qu'une chaine de caractère simple TOKEN. Nous verrons dans la prochaine partie pour générer une vrai token.

Première étape, récupérez l'utilisateur associé à l'email fourni. Si rien n'est trouvé, alors message d'erreur avec un code 400.

Puis, pour vérifier les informations de votre utilisateur, je vous invite à créer une fonction dans votre `UserService` qui s'appellerait par exemple `verifyPassword`. Celle-ci prendrait 2 paramètres, le password entré par l'utilisateur, et l'utilisateur (ou juste son hash) associé à l'email. Il utilisera la méthode `compare` de `bcrypt` pour retourner si oui ou non (donc un booléen), le mot de passe est correct.

Pour finir dans le contrôleur, utilisez votre service pour valider le password, puis retourner la réponse appropriée.

03 - Json Web Token

Dans le web, une multitude de possibilités existent pour gérer la connexion. Ici, nous allons utiliser JWT qui produit un token (une chaine de caractère). Je vous renvoi vers ce site si vous voulez plus d'informations. (En anglais).

De notre côté, il nous faudra installer une dépendance :

```
npm install jsonwebtoken
```

Puis créez un fichier keygen.js contenant

```
let c = require('crypto').randomBytes(64).toString('hex');
console.log(c);
```

Puis faite le exécuter par `node` dans le terminal. Cela devrait vous afficher une string de 64 caractère. Récupérez la puis ajoutez une nouvelle variable d'environnement `TOKEN_SECRET` à votre `props.env`. Nous pourrons utiliser cette variable dans notre code.

Retour à notre `AuthController`, importez le module `jwt`

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

Puis, à la place de la chaine `TOKEN` de notre réponse dans `/login`, utilisez la méthode `sign` de `jwt`.

Elle contient 3 paramètres :

- Le payload (contenu) : Ici, nous passeront le userId en tant que clé d'un objet
- La clé privé pour signer : Notre TOKEN_SECRET
- Les options : Nous préciseront que le token expire en 24h.

Voici un exemple non contractuel:

```
token: jwt.sign(
    { userId : userModel._id },
    'ORENOKEY',
    { expiresIn : '2h'}
)
```

Si tout fonctionne, la réponse de votre route si les infos sont correct devrait ressembler à ceci :

```
{
    "userId": "67a22e9ca059752290580b90",
    "token":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ1c2VySWQiOiI2N2EyMmU5Y2EwNTk3NT
IyOTA1ODBiOTAiLCJpYXQiOjE3Mzg2OTgwOTYsImV4cCI6MTczODc4NDQ5Nn0.umeLBn0-
```

```
eZQCl1gZ85lG73R5n2j3a--Zl93EIjp_wgg"
}
```

Voila, nous avons un moyen de garder l'utilisateur connecter. Il nous reste donc à protéger nos routes privés pour qu'elle se serve du token.

04 - Middleware

Nous allons à présent créer le `middleware` qui va vérifier que l'utilisateur est bien connecté et transmettre les informations de connexion aux différentes méthodes qui vont gérer les requêtes.

Créez un dossier `middleware` et un fichier `auth.middleware.js` à l'intérieur :

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

module.exports = (req, res, next) => {
   try {
       const token = req.headers.authorization.split(' ')[1];
       const decodedToken = jwt.verify(token, process.env.TOKEN_SECRET);
       const userId = decodedToken.userId;
       req.auth = {
           userId: userId
       };
    next();
   } catch(error) {
       res.status(401).json({ error });
   }
};
```

Dans ce middleware :

- Étant donné que de nombreux problèmes peuvent se produire, nous insérons tout à l'intérieur d'un bloc `try...catch`.
- Nous extrayons le token du header `Authorization` de la requête entrante. N'oubliez pas qu'il contiendra également le mot-clé `Bearer`. Nous utilisons donc la fonction `split` pour tout récupérer après l'espace dans le header. Les erreurs générées ici s'afficheront dans le bloc `catch`.
- Nous utilisons ensuite la fonction `verify` pour décoder notre token. Si celui-ci n'est pas valide, une erreur sera générée et `catch`.

- Nous extrayons l'ID utilisateur de notre token et le rajoutons à l'objet `Request` afin que nos différentes routes puissent l'exploiter.
- Tout fonctionne et notre utilisateur est authentifié. Nous continuons l'exécution à l'aide de la fonction `next()`.

```
Petite précision sur next. Celui-ci doit être un paramètre de votre fonction si cette dernière n'est pas la dernière à s'executer dans le processus. Hors, souvent, nos contrôleurs étaient dans ce cas là.
```

Pour vérifier que tout est OK, retournons dans notre `UserRouter`. Puis protégez la route GET avec notre nouveau `middleware`.

```
const authM = require('../middlewares/auth.middleware');
router.get('/:field', authM, userController.getUser);
```

Notez bien l'utilisation de `authM` avant le contrôleur.

Puis pour testez :

- loggez vous avec un compte
- Récupérez le token
- Dans la requête GET `/users`, dans l'onglet `Authorization`, sélectionnez `Bearer Token`
- Insérez votre token, puis lancez la requête
- Puis testez en enlevant un caractère du token.

Si dans le premier cas, vous avez la bonne réponse de l'API, puis une erreur dans le deuxième, alors félicitation, votre route est bien protégée.

Enfin presque ...

Actuellement, toute personne connecté peut avoir accès aux infos de tous les utilisateurs. Pour finir, je vous invite à essayer de retourner une erreur si l'utilisateur loggé n'est pas celui dont on demande les informations.

A vous de jouer !

05 - Vérifier l'utilisateur

Souvent, côté front, des pages devront être exclusif à des utilisateurs connecté, où bien à des utilisateurs spécifiques. Et puis, avec le temps, le token n'est plus forcement valide. Je vous propose donc de créer une route `/checkUser`. Son seul objectif, est de valider que le token est valide. Si oui, renvoyer un simple code 204 No content. Si le token n'est plus bon, renvoyez un code 401 avec une réponse JSON expliquant le soucis.

Cela ne devrait vous demander que de coder 2 lignes dans le routeur et une ligne dans le contrôleur (à mettre dans une fonction).