self-service-machine-api - STEP 13

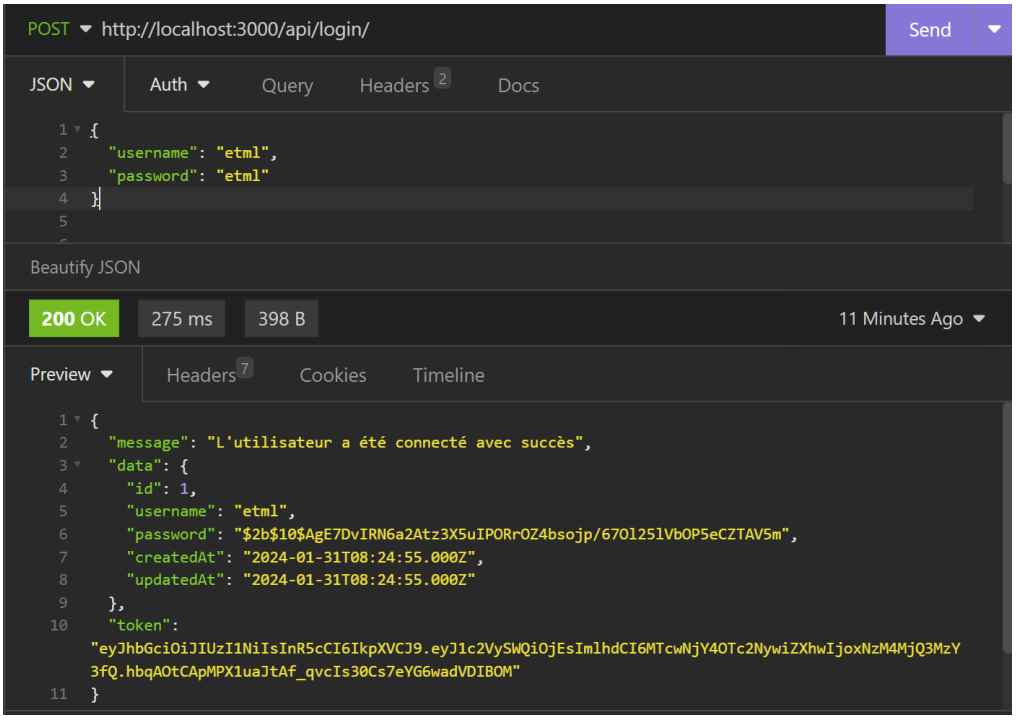
Nous allons mettre en place un système d'authentification.

L'authentification est un processus qui permet de restreindre l'accès aux points de terminaison (aux routes) de notre API REST.

Cette authentification comprend 2 étapes :

- Lors de la **1ère étape**, un utilisateur se connecte à l'aide son nom d'utilisateur et son mot de passe à l'API REST.

Dans notre cas, l'utilisateur utilise la route `POST /api/login/` en fournissant un json comprenant son `username` et son `password`.



A la suite de cette authentification, l'utilisateur obtient un jeton JWT.

En effet, l'authentification entre une application web et une API REST repose sur un standard reconnu, qui est l'authentification avec les « JSON Web Token ».

Chaque client doit obtenir un identifiant unique appelé un jeton JWT.

Un jeton JWT a une durée de validité limitée dans le temps, et se présente sous la forme d'une chaîne de caractères.

Le jeton JWT transite dans l'en-tête HTTP authorization, avec pour valeur "Bearer ".

- La **2ème étape** est l'utilisation de ce jeton JWT :

Une fois que l'utilisateur a obtenu le jeton JWT, il devra l'utiliser à chaque requête vers l'API REST. Ainsi les échanges entre le consommateur et l'API REST seront sécurisés.

2 exigences au niveau de la sécurité

La mise en place d'une authentification côté API REST nécessite de respecter deux exigences principales au niveau de la sécurité :

- chiffrer le mot de passe de l'utilisateur
- sécuriser l'échange des données entre le consommateur et l'API REST par l'utilisation de jeton JWT

Création d'un modèle UserModel

Pour commencer, nous avons besoin de stocker des utilisateurs dans la base de données. Nous allons donc créer un modèle `UserModel`.

```
const UserModel = (sequelize, DataTypes) => {
  return sequelize.define("User", {
    id: {
      type: DataTypes.INTEGER,
      primaryKey: true,
      autoIncrement: true,
    },
    username: {
      type: DataTypes.STRING,
      allowNull: false,
      unique: { msg: "Ce username est déjà pris." },
    },
    password: {
      type: DataTypes.STRING,
      allowNull: false,
    },
  });
};

export { UserModel };
```

Rien de nouveau pour ce modèle.

La table MySQL coorespondante à ce modèle est la table `User`.

A noter que l'on s'assure que le `username` soit unique.

Maintenant que la table `User` existe, nous devons importer des utilisateurs.

Importer un utilisateur en base de données

Le mot de passe de chaque utilisateur ne doit pas être stocké en clair pour des questions de sécurité.

Nous aurons besoin de la dépendance `bcrypt` pour hasher le mot de passe de l'utilisateur.

Comme nous en avons maintenant l'habitude, pour installer la dépendance, nous pouvons utiliser la commande suivante :

`npm install bcrypt --save`

Voila le code permettant d'ajouter un utilisateur `etml` avec un mot de passe hashé.

Le mot de passe en clair est `etml`.

```
const importUsers = () => {
  bcrypt
    .hash("etml", 10) // temps pour hasher = du sel
```

```
    .then((hash) =>
      User.create({
        username: "etml",
        password: hash,
      })
    )
    .then((user) => console.log(user.toJSON()));
};
```

Dans ce code, le seul élément nouveau est l'utilisation de la fonction `hash` permettant de hasher le mot de passe `etml`.

Nous pouvons appeler cette méthode `importUsers` juste après l'importation des produits.

```
let initDb = () => {
  return sequelize
    .sync({ force: true }) // Force la synchro => donc supprime les données également
    .then((_) => {
      importProducts();
      importUsers();
      console.log("La base de données db_products a bien été synchronisée");
    });
};
```

On peut vérifier en base de données que l'utilisateur est bien créé :

```
1 • SELECT * FROM users;
```

Result Grid | Filter Rows: | Edit: | Export/Import: | Wrap Cell Content:

id	username	password	createdAt	updatedAt
1	etml	$2b$10$14AxURQ6JwW...	2023-12-30 09:57:47	2023-12-30 09:57:47
NULL	NULL	NULL	NULL	NULL

Authentification de l'utilisateur

Nous devons mettre en place la nouvelle route permettant à un utilisateur de se connecter.

Nous allons isoler cette route dans un fichier js `src/routes/login.mjs`.

Et dans le fichier `src/app.mjs` nous pouvons importer le router du login.

```
...
import { productsRouter } from "./routes/products.mjs";
app.use("/api/products", productsRouter);

import { loginRouter } from "./routes/login.mjs";
app.use("/api/login", loginRouter);
...
```

Nous devons maintenant construire la route permettant à un utilisateur de s'authentifier à l'aide de son nom d'utilisateur et de son mot de passe et ainsi pouvoir obtenir le jeton JWT.

Pour cela nous avons besoin de la librairie `jsonwebtoken` pour générer un jeton JWT.

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

Voila le code complet de la nouvelle route `POST /api/login/`.

```
import express from "express";
import bcrypt from "bcrypt";
import jwt from "jsonwebtoken";

import { User } from "../db/sequelize.mjs";
import { privateKey } from "../auth/private_key.mjs";

const loginRouter = express();

loginRouter.post("/", (req, res) => {
  User.findOne({ where: { username: req.body.username } })
    .then((user) => {
      if (!user) {
        const message = `L'utilisateur demandé n'existe pas`;
        return res.status(404).json({ message });
      }
      bcrypt
        .compare(req.body.password, user.password)
        .then((isPasswordValid) => {
          if (!isPasswordValid) {
            const message = `Le mot de passe est incorrecte.`;
            return res.status(401).json({ message });
          } else {
            // JWT
            const token = jwt.sign({ userId: user.id }, privateKey, {
              expiresIn: "1y",
            });
            const message = `L'utilisateur a été connecté avec succès`;
            return res.json({ message, data: user, token });
          }
        });
    })
    .catch((error) => {
      const message = `L'utilisateur n'a pas pu être connecté. Réessayez dans quelques instants`;
      return res.json({ message, data: error });
    });
});

export { loginRouter };
```

Nous allons détailler 2 portions de code contenant des éléments nouveaux.

Comparaison des mots de passe

```
bcrypt.compare(req.body.password, user.password);
```

Ce code permet de comparer le mot de passe hashé de l'utilisateur stocké dans la base de données avec le mot en clair fourni dans le json pour notre consommateur.

On utilise la méthode `compare()` de la librairie `bcrypt`.

Générer un jeton JWT

Pour générer un jeton JWT, on utilise la méthode `sign` de la librairie `jsonwebtoken`.

```
const token = jwt.sign({ userId: user.id }, privateKey, {
  expiresIn: "1y",
```

```
});
```

Cette méthode prend :

- En 1er paramètre le payload. Ici un objet js contenant l'id de l'utilisateur.
- En 2ème paramètre une clé privée. Cette clé privée est simplement ici une chaîne de caractères. Il ne faut jamais la partager. Attention ! Même pas dans github !

```
const privateKey = "CUSTOM_PRIVATE_KEY";

export { privateKey };
```

- En 3ème paramètre un objet js pour définir des options. Dans notre cas, on précise dans combien de temps va expirer ce jeton JWT. "1y" signifie un an (1 year).

Définition du payload :

En développement web, le terme "payload" fait généralement référence aux données transmises entre le serveur web et le navigateur client dans le cadre des requêtes HTTP (Hypertext Transfer Protocol) et des réponses correspondantes. Plus précisément, il s'agit de la partie des données de la réponse HTTP qui contient le contenu réel que le navigateur affiche à l'utilisateur.

Utilisation du jeton JWT

Maintenant que le consommateur de notre API REST a obtenu son jeton JWT, il doit l'utiser pour chaque requête.

Nous devons donc récupérer ce jeton JWT et vérifier sa validité.

Dans chacune des routes des produits, nous allons ajouter un nouveau paramètre en 2ème position.

Dans le fichier `src/routes/products.mjs` :

```
...
import { auth } from "../auth/auth.mjs";

const productsRouter = express();

productsRouter.get("/", auth, (req, res) => {
...
```

TODO : A vous de faire cela pour toutes les routes !

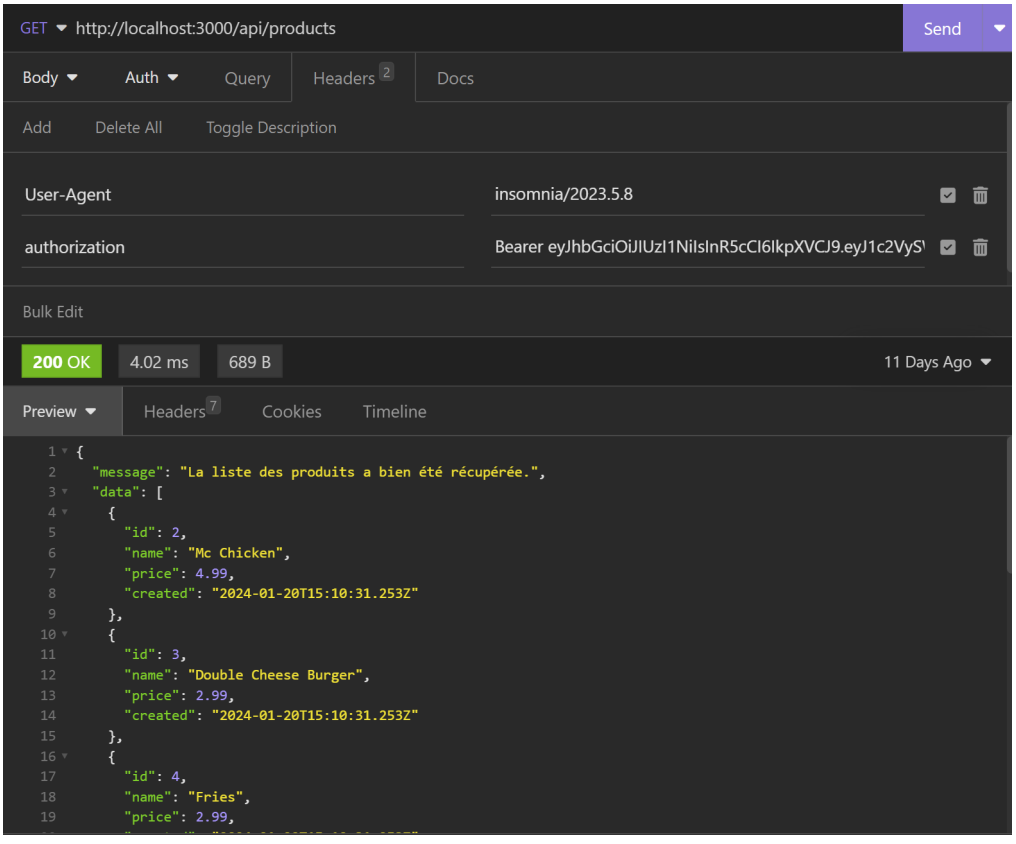
C'est maintenant à nous d'implémenter le fichier `src/auth/auth.mjs` afin de vérifier le jeton JWT fourni par le consommateur.

```
import jwt from "jsonwebtoken";
import { privateKey } from "./private_key.mjs";

const auth = (req, res, next) => {
  const authorizationHeader = req.headers.authorization;

  if (!authorizationHeader) {
    const message = `Vous n'avez pas fourni de jeton d'authentification. Ajoutez-en un dans l'en-tête de la requête.`;
    return res.status(401).json({ message });
  } else {
```

```
    const token = authorizationHeader.split(" ")[1];
    const decodedToken = jwt.verify(
      token,
      privateKey,
      (error, decodedToken) => {
        if (error) {
          const message = `L'utilisateur n'est pas autorisé à accéder à cette ressource.`;
          return res.status(401).json({ message, data: error });
        }
        const userId = decodedToken.userId;
        if (req.body.userId && req.body.userId !== userId) {
          const message = `L'identifiant de l'utisateur est invalide`;
          return res.status(401).json({ message });
        } else {
          next();
        }
      }
    );
  }
};

export { auth };
```

TODO : A vous de commenter en détail le code ci-dessus afin de le comprendre parfaitement

Maintenant que le code est en place, vous devez tester toutes vos routes mais cette fois en utilisant un jeton JWT.



Cette étape était fondamentale afin de sécuriser notre API REST !

Dans la prochaine étape, nous allons nous intéresser à la documentation de notre API REST.

Passons à l'étape n°14