AUTHENTIFICATION AVEC JWT DANS UNE APPLICATION EXPRESS NODEJS MONGOOSE

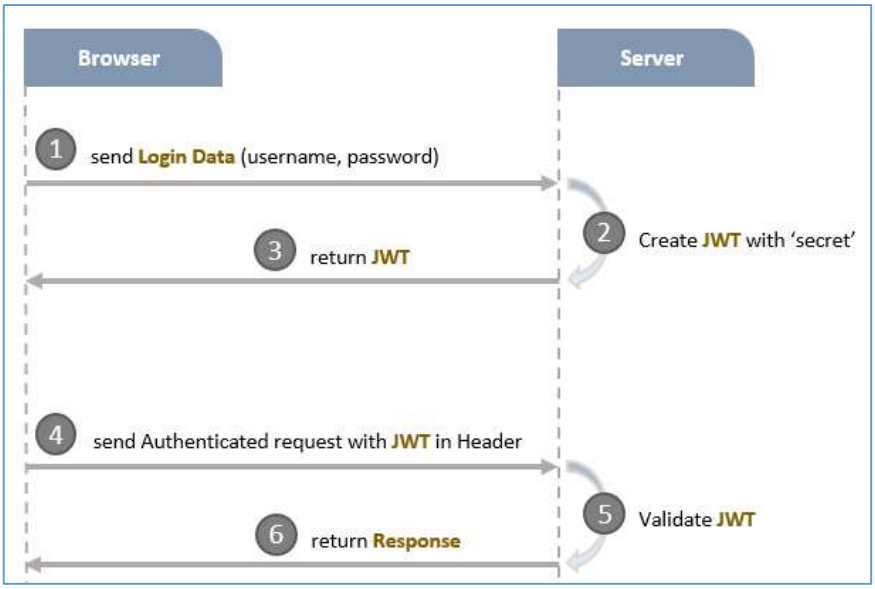
I. Introduction

Cet atelier a pour but de vous montrer comment utiliser JWT. Lors de votre développement il se trouve qu'il nécessite une authentification de la part de l'utilisateur. Pour ce faire soit vous passez par une API d'un service comme Google, Facebook, Twitter, etc. Ou bien vous la gérez vous-même. Pour ce faire, l'utilisateur doit pouvoir s'enregistrer puis se connecter. Il faut donc passer par un système sécurisé. Avec un jeton ou token en anglais. Le principe repose sur le fait de vérifier qu'un utilisateur existe. Si c'est le cas, un jeton lui est fourni. C'est via ce dernier que le serveur d'API autorise l'accès à certaines routes.

Dans cet atelier, nous allons créer un exemple d'API Node.js Express REST qui prend en charge l'authentification basée sur les jetons avec JWT (JSONWebToken). Et nous allons voir comment faire pour l'inscription et la connexion de l'utilisateur avec l'authentification JWT.

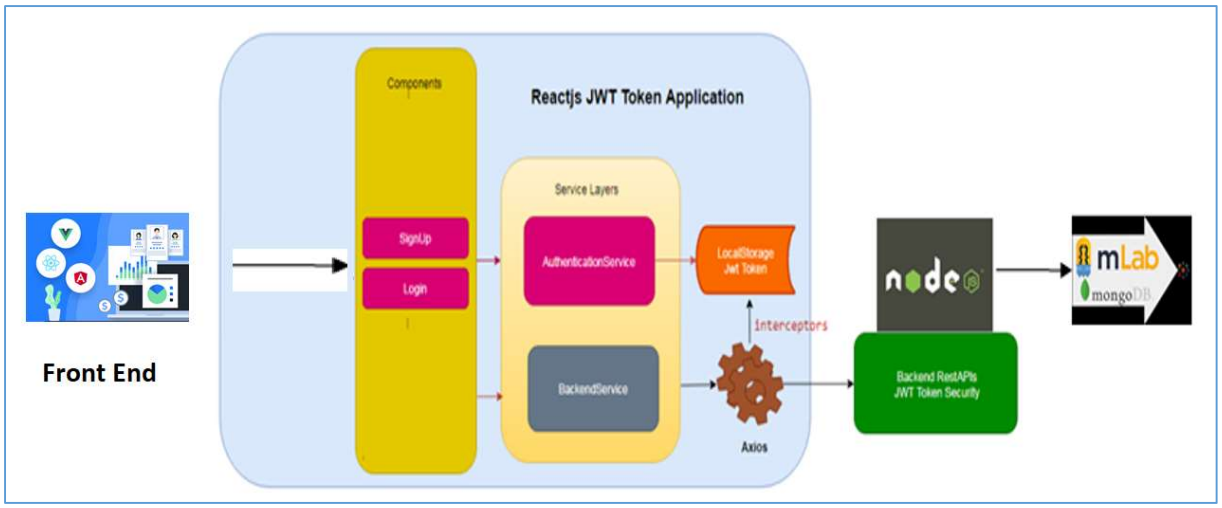
Au lieu d'enregistrer les identifiants de connexion de l'utilisateur (utilisateur et mot de passe) pendant un certain temps, enregistrons un "jeton" côté client qui est codé à partir d'une charge utile de données à l'aide d'un secret.

De cette façon, notre application sera plus sécurisée, avec chaque requête HTTP, nous envoyons ce jeton et c'est ainsi que nous vérifierons notre authentification auprès du serveur d'application distant.



Le grand avantage de l'authentification basée sur le jeton est que nous stockons le jeton Web JSON (JWT) côté client. Les jetons Web JSON sont des chaînes de texte qui peuvent être utilisées par le client et le serveur pour authentifier et partager facilement des informations.

L'architecture de l'application complète devient :



II. JWT

Un JSON Web Token est un access token (jeton d'accès) aux normes RFC 7519 qui permet un échange sécurisé de donnée entre deux parties. Il contient toutes les informations importantes sur une entité, ce qui rend la consultation d'une base de données superflue et la session n'a pas besoin d'être stockée sur le serveur (stateless session).

Les JSON Web Token sont particulièrement appréciés pour les opérations d'identification. Les messages courts peuvent être chiffrés et fournissent alors des informations sûres sur l'identité de l'expéditeur et si celui-ci dispose des droits d'accès requis. Les utilisateurs eux-mêmes ne sont qu'indirectement en contact avec les jetons, par exemple lorsqu'ils entrent un nom d'utilisateur et un mot de passe dans un masque. La véritable communication se fait entre les différentes applications du côté serveur et client.

Le token c'est une clé qui est générée lorsque vous accédez à la route de connexion de votre API. Dans le cas où la connexion est validée c'est à dire que l'utilisateur existe et que le mot de passe correspond, le serveur va envoyer un token. Ce token est hashé, c'est à dire crypté et va être divisé en 3 parties.

- Le **header**, c'est là que les informations sur le token sont stockés, en l'occurrence jsonwebtoken (jwt) et l'algorithme de hashage (HS256).

Exemple :

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

base64Header = base64Encode(header)

// eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9

- Ensuite on a le **payload**, c'est les données que l'on veut stocker dans ce token. Grâce à ça on peut par exemple contrôler que l'utilisateur est administrateur. Si c'est le cas on le laisse accéder aux données, et dans le cas contraire on le redirige.

Exemple :

```
{
        "sub": "1234567890",
        "name": "Jhon Doe",
        "admin": true
}
```

base64Payload = base64Encode(payload)

// eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9

- Enfin on a la **signature**, c'est l'aspect unique du token qui permet à votre serveur de vérifier qu'il provient bien de chez vous. Grâce à ça il n'y a aucun moyen que le token soit hacké.

Exemple :

signature = HS256(base64Header + '.' + base64Payload, 'secret')

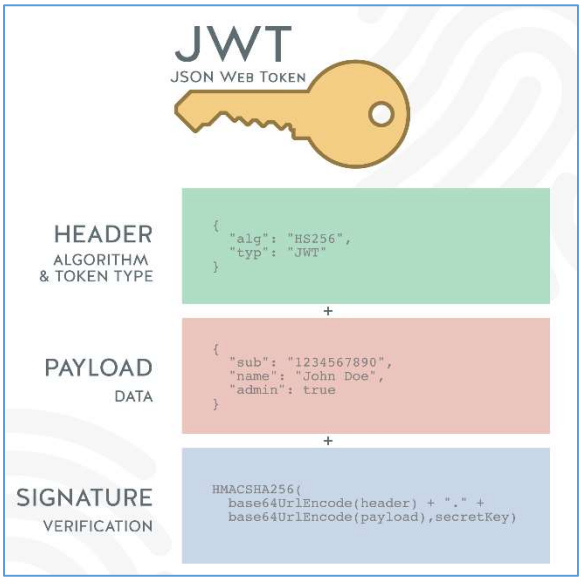
// dyt0CoTl4WoVjAHI9Q_CwSKhl6d_9rhM3NrXuJttkao

Pour finir, il faut rassembler ces trois composantes et les séparer par un point.

Token = base64Header + '.' + base64Payload + '.' + signature

Exemple :

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJzdWIiOiIxMjM0NTY3ODkwIiwibmFtZSI6IkpvaG4gRG9lIiwiYWRtaW4iOnRydWV9.dyt0CoTl4WoVjAHI9Q_CwSKhl6d_9rhM3NrXuJttkao



Dans JWT, un jeton est codé à partir d'une charge utile de données (payload) à l'aide d'un secret. Ce jeton est transmis au client. Chaque fois que le client envoie ce jeton avec une requête, le serveur le valide et renvoie la réponse.

Une clé secrète est déterminée avant l'utilisation du JWT. Dès qu'un utilisateur a entré avec succès ses données de connexion, le JWT est renvoyé avec la clé et stocké localement. Le transfert se fait par HTTPS afin de mieux protéger les données.

Lorsque l'utilisateur veut accéder à des ressources protégées comme une API ou un chemin d'accès protégé, le JWT sera envoyé par l'agent utilisateur comme paramètre (par exemple « jwt » pour les GET-Requests) ou comme en-tête d'autorisation (pour POST, PUT, OPTIONS, DELETE). L'interlocuteur peut déchiffrer le JSON Web Token et si le contrôle est réussi, exécuter la demande.

Le JSON Web Token étant un identifiant de connexion, vous n'avez pas besoin de conserver le token plus longtemps que nécessaire et vous n'avez pas besoin de stocker de données sensibles dans la mémoire du navigateur.

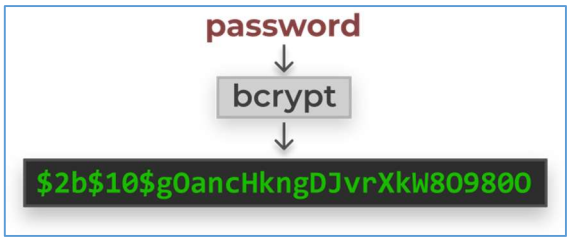
Les JSON Web Token offrent un certain nombre d'avantages comparés aux méthodes traditionnelles d'authentification et d'autorisation avec des cookies et sont pour cela utilisé dans les scénarios suivants :

- Applications REST : dans les applications REST, le JWT sécurise le protocole sans état en envoyant les informations pour l'authentification directement lors de la requête.
- Cross origin resource sharing : le JSON Web Token envoie les informations lors du Cross Origin Resource Sharing. Cela présente un énorme avantage par rapport aux cookies, qui ne sont généralement pas envoyés dans cette procédure.
- Utilisation de plusieurs Frameworks : les JSON Web Token sont standardisés et donc polyvalents. Lors de l'utilisation de plusieurs Frameworks, ils permettent de partager facilement les données d'authentification.

III. bcrypt

Le stockage des mots de passe des utilisateurs dans une base de données nécessite une sécurité au cas où ils seraient divulgués ou une personne non autorisée a eu accès à notre base de données. Actuellement, la méthode la plus populaire et la plus sûre consiste à utiliser bcrypt.

bcrypt est un algorithme de hachage qui est évolutif en fonction du serveur qui le lance. Sa lenteur et sa robustesse garantit qu'un hacker voulant décoder un mot de passe doit investir dans un matériel couteux et très puissant pour décoder un mot de passe. Ajouter à cela que l'algorithme utilise toujours des grains de sels, et vous pouvez être certain qu'une attaque est aujourd'hui presque impossible. Il s'agit aujourd'hui de la méthode de hashage la plus sûre.



bcrypt utilise l'algorithme de hachage Eksblowfish. Celui-ci est similaire à l'algorithme Blowfish (un algorithme de chiffrement symétrique), excepté que la phase de planification de la clé de Eksblowfish garantie que tout état sous-jacent dépend à la fois du sel et de la clé (le mot de passe encodé). Ainsi aucun état ne peut être pré calculé sans connaître à la fois ces deux éléments. A cause de cette différence, bcrypt est un algorithme de hashage unidirectionnel : Vous ne pourrez jamais retrouver le mot de passe sans connaitre à la fois le grain de sel, la clé et les différentes passes que l'algorithme à utiliser.

Un grain de sel, ou "salt", en cryptographie, est appliqué durant le processus de hachage pour éliminer la possibilité d'attaques par dictionnaires (hachages enregistrés dans une grande liste et comparés).

En d'autres termes, un grain de sel est une petite donnée additionnelle qui renforce significativement la puissance du hachage pour le rendre beaucoup plus difficile à cracker.

Remarque :

L'authentification est le processus de vérification de l'identité d'un utilisateur. Cela signifie généralement vérifier que les informations d'identification fournies par l'utilisateur, telles qu'un nom d'utilisateur et un mot de passe, correspondent à des informations valides stockées dans le système. L'objectif principal de l'authentification est de permettre à un utilisateur de prouver son identité auprès du système. Une fois qu'un utilisateur s'est authentifié avec succès, il est considéré comme authentifié et peut accéder à certaines parties de l'application ou des fonctionnalités qui lui sont autorisées.

L'autorisation, d'autre part, concerne les privilèges accordés à un utilisateur authentifié. Une fois qu'un utilisateur est authentifié, l'autorisation détermine ce qu'il est autorisé à faire dans le système. Cela implique de définir des règles et des permissions qui spécifient quelles ressources ou fonctionnalités un utilisateur peut accéder, modifier ou supprimer. Par exemple, un administrateur pourrait avoir des autorisations étendues pour accéder à toutes les fonctionnalités d'un système, tandis qu'un utilisateur régulier pourrait avoir un accès limité.

IV. Etapes de réalisation

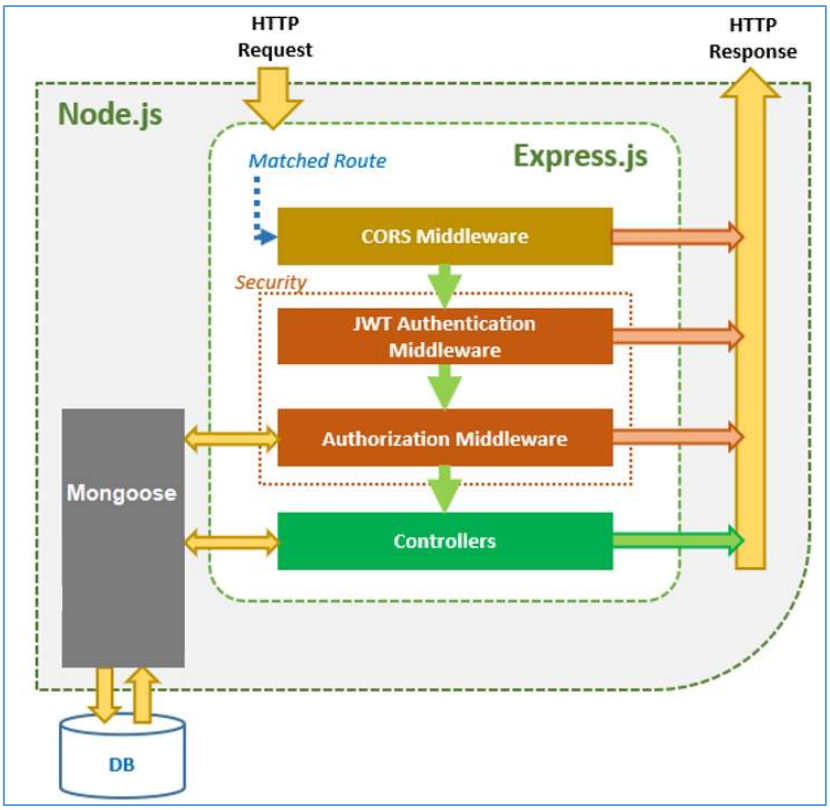
Via les routes express, la requête HTTP qui correspond à une route sera vérifiée par le middleware CORS avant d'arriver à la couche de sécurité.

La couche de sécurité comprend :

- JWT Authentication Middleware : vérifier l'inscription, vérifier le jeton.
- Authorization Middleware : vérifier les rôles de l'utilisateur avec un enregistrement dans la base de données.

Si ces middlewares génèrent une erreur, un message sera envoyé en tant que réponse HTTP.

Les contrôleurs interagissent avec la base de données MongoDB via Mongoose et envoient une réponse HTTP (token, informations utilisateur, données basées sur les rôles,etc.) au client.



Remarque :

Nous supposons que l'application est déjà créée et que la configuration requise est déjà faite

1. Pour pouvoir créer et vérifier les tokens d'authentification, il nous faudra un nouveau package. On commence par installer le module **jsonwebtoken** et **bcrypt**

```
npm install jsonwebtoken bcrypt
```

2. En supposant que dotenv est déjà installé ouvrez le fichier **.env**

Puis ajoutez la ligne :

```
TOKEN = 328393161742435634
```

3. Créez le Model de User appelé « user.js » dans le dossier « models ».


```
const  mongoose =require("mongoose")
var userSchema = mongoose.Schema({
    name:{
        type:String,
```

```
        required:"Name is required"
    } ,
   email:{
        type:String,
        required:"Email is required",
        unique:true
    } ,
    password:{
        type:String,
        required:"password is required"
    } ,
    role:{
        type: String,
        enum: ["user", "admin"],
        default:"user"
    } ,
   avatar :{
        type: String,
        required: false
        }
},
{
    timestamps: true,
},
);

module.exports=mongoose.model('User',userSchema)
```

4. Créez les routes dans routes/user.route.js, puis créer la route pour l'enregistrement.

routes > JS user.route.js >

```
const express = require('express');
const router = express.Router();
const User = require('../models/user.js');
const  jwt  = require('jsonwebtoken');
const bcrypt = require('bcrypt');

//Register

router.post('/register', async (req, res, )=> {

    const{name,email,password,role,avatar}=req.body;

    const user = await User.findOne({ email })
    if (user) return res.status(404).send({ success: false, message: "Account
already exists" })
```

```
    const salt=await bcrypt.genSalt(10);
    const hash=await bcrypt.hash(password,salt);

    const newUser=new User({
      name:name,
      email:email,
      password:hash,
      role:role,
      avatar:avatar
        });
    try {
            await newUser.save();

        return res.status(201).send({ success: true, message: "Account created
successfully", user: newUser })
       } catch (error) {
            res.status(409).json({ message: error.message });
       }

});
  module.exports = router;
```

La méthode de hachage nécessite des tours de sel (genSalt), c'est-à-dire le facteur de coût - en termes simples, il s'agit d'une fonction de coût (plus le mot de passe est grand, plus le mot de passe est crypté) - la valeur recommandée est 10.

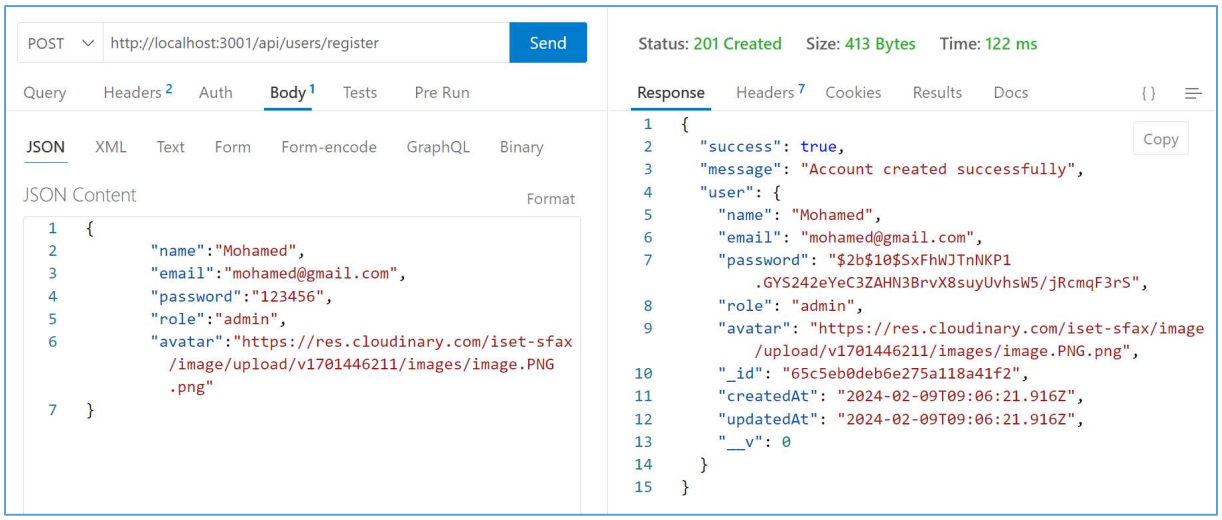
5. Dans le fichier **app.js**, on ajoute la nouvelle route pour user.


```
const userRouter =require("./routes/user.route")
app.use('/api/users', userRouter);
```

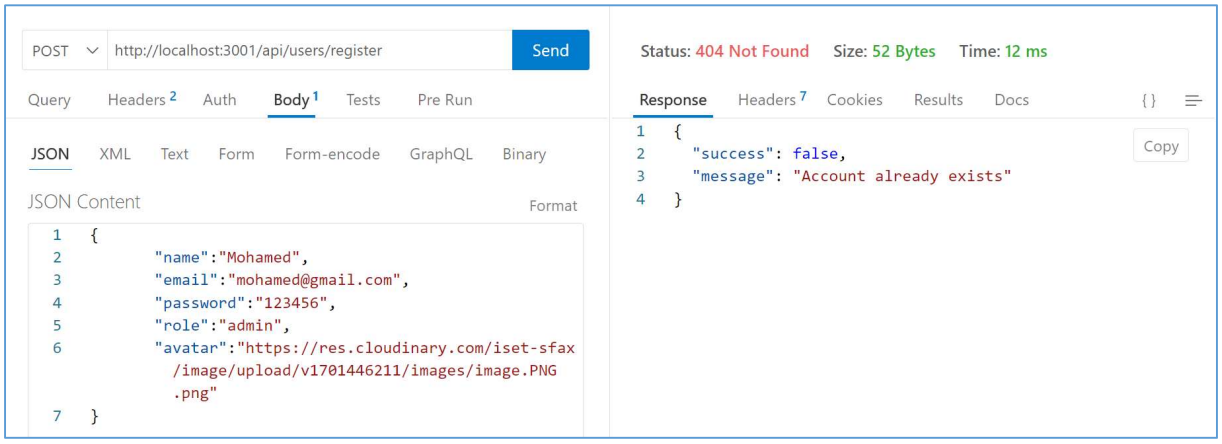
On va tester le code :

http://localhost:3001/api/users/register

Requête POST



Si on veut enregistrer le même utilisateur :



6. On va ajouter le code de connexion en créant la route login

routes > user.route.js >

```
const express = require('express');
const router = express.Router();
const User = require('../models/user.js');
const  jwt  = require('jsonwebtoken');
const bcrypt = require('bcrypt');

//Register

router.post('/register', async (req, res, )=> {

    const{name,email,password,role,avatar}=req.body;

    const user = await User.findOne({ email })
    if (user) return res.status(404).send({ success: false, message: "Account already exists" })
```

```
    const salt=await bcrypt.genSalt(10);
    const hash=await bcrypt.hash(password,salt);

    const newUser=new User({
      name:name,
      email:email,
      password:hash,
      role:role,
      avatar:avatar
        });
    try {
            await newUser.save();

        return res.status(201).send({ success: true, message: "Account created
successfully", user: newUser })
       } catch (error) {
            res.status(409).json({ message: error.message });
       }

});
```

```
//Generate Token
const generateToken=(user) =>{
    return jwt.sign({user}, process.env.TOKEN, { expiresIn: '1h' });
  }
```

```
//login
router.post('/login', async (req, res) =>  {
    try {
        let { email, password } = req.body

        if (!email || !password) {
            return res.status(404).send({ success: false, message: "All fields
are required" })
        }

        let user = await User.findOne({ email })

        if (!user) {

            return res.status(404).send({ success: false, message: "Account
doesn't exists" })

        } else {

      let isMatch = await bcrypt.compare(password, user.password)
      if(!isMatch) {res.status(400).json({success: false, message:'Please
verify your credentials'}); return;}
```

```
      const token = generateToken(user);
      res.status(200).json({
       success: true,
       token,
       user
    })
   }
 } catch (error) {
    res.status(404).json({ message: error.message });
}
});

module.exports = router;
```

Nous commençons par importer jwt de « jsonwebtoken » dans notre contrôleur utilisateur. Puis, nous l'utiliserons dans notre fonction « generateToken ».

Nous utilisons la fonction sign de jsonwebtoken pour encoder un nouveau token.

Nous utilisons une chaîne secrète de développement temporaire process.env.TOKEN pour encoder notre token (à remplacer par une chaîne aléatoire beaucoup plus longue pour la production).

Nous définissons la durée de validité du token à 1 min. L'utilisateur devra donc se reconnecter au bout de 1 heure.

La fonction permet de vérifier si un utilisateur existe ou non dans la base de données. Dans ce cas nous renvoyons le token au front-end avec notre réponse au format json.

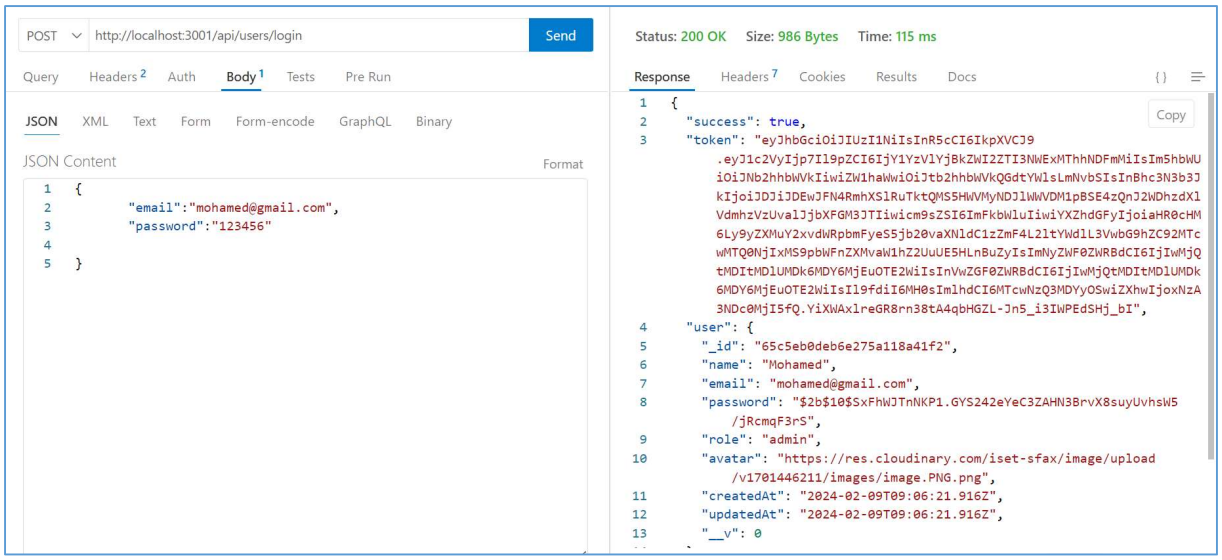
Il s'agit de comparer le mot de passe donné avec le mot de passe de la base de données. bcrypt utilise un algorithme qui crée différents mots de passe hachés à chaque fois, mais la comparaison des mots de passe pour chaque mot de passe donnera true.

La fonction findOne() est utilisée pour trouver un document selon la condition (ici c'est l'email). Si plusieurs documents correspondent à la condition, il renvoie le premier document satisfaisant la condition.

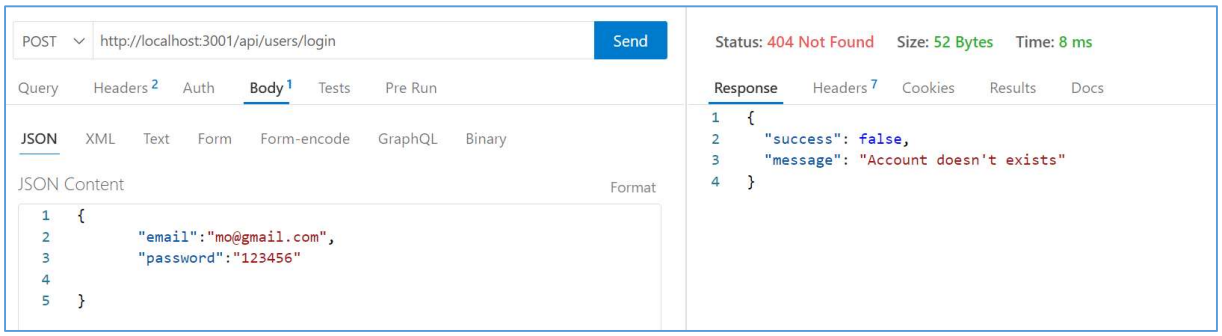
On va tester le code :

http://localhost:3001/api/users/login

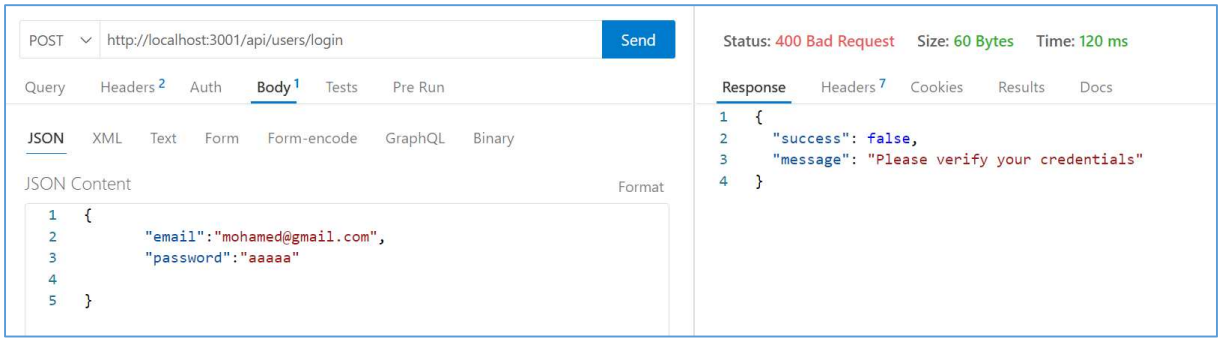
Requête POST



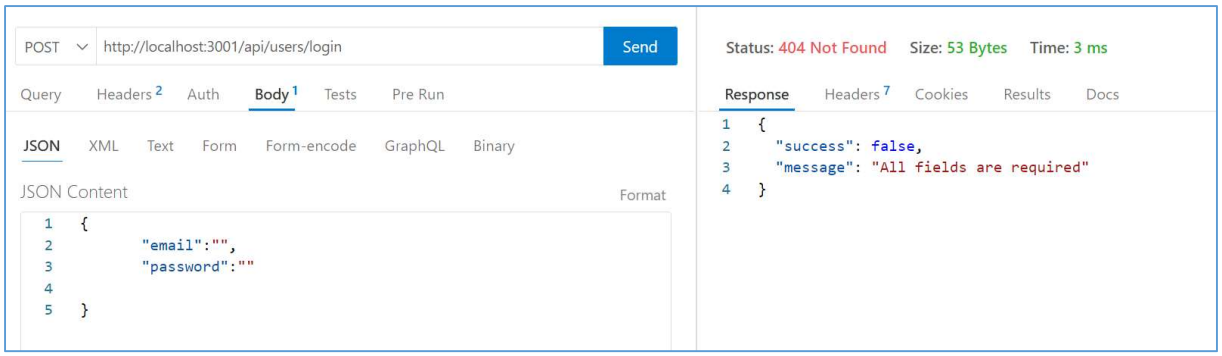
Si email inexistent



Si mot de passe incorrect



Si les champs sont vides



7. Nous allons à présent créer le **middleware** qui protégera les routes sélectionnées et vérifiera que l'utilisateur est authentifié avant d'autoriser l'envoi de ses requêtes.

Créez un dossier « **middleware** » et un fichier « **auth.js** » à l'intérieur.

middlewares/auth.js est le fichier qui vérifiera qu'on détient le jeton.

```
const  jwt  = require('jsonwebtoken');
const auth=async(req,res,next)=>{
const authHeader = req.headers['authorization'];
const token = authHeader && authHeader.split(' ')[1];
  if (!token) {
    return res.status(401).send({ success: false, message: 'No token provided'
});
  }
  jwt.verify(token, process.env.TOKEN, (err, user) => {
    if (err) {
        return res.status(401).send({ success: false, message: 'Invalid token'
});
    }
    req.user = user;
    next();
  });
 }

 module.exports = auth;
```

Dans ce middleware 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 récupérer tout après l'espace dans le header (d'où le split(' ')[1]).

Le client attache généralement JWT dans l'en-tête d'autorisation avec le préfixe Bearer :

```
Authorization: Bearer [header].[payload].[signature]
```

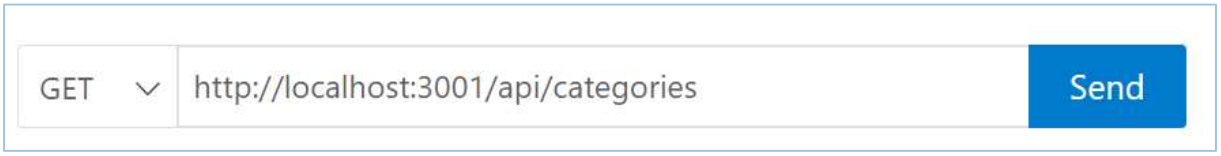
Nous utilisons ensuite la fonction **verify** pour décoder notre token. Si celui-ci n'est pas valide, une erreur sera générée avec le status 401. Dans le cas contraire, tout fonctionne, et notre utilisateur est authentifié. Nous passons l'exécution à l'aide de la fonction next() .

8. On va mettre en place un middleware pour chaque page protégée. Dans le fichier « categorie.route.js » ; si on veut afficher la liste des categories mais avec JWT ont doit importer le module « **auth** ».

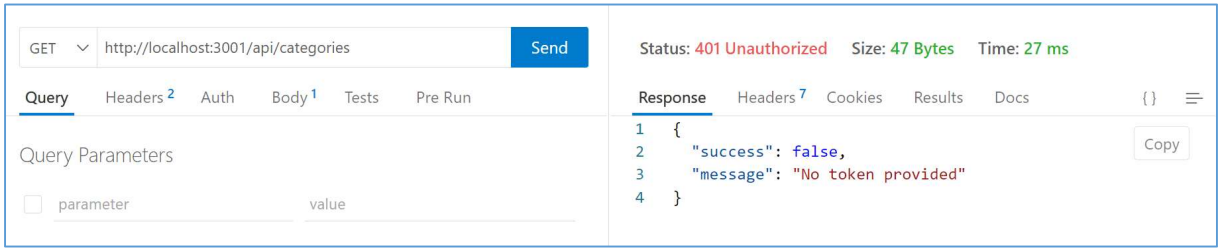
```
const auth = require( "../middleware/auth.js");

router.get('/', auth, async (req, res, )=> {
```

Tester l'API :

a- Sans token

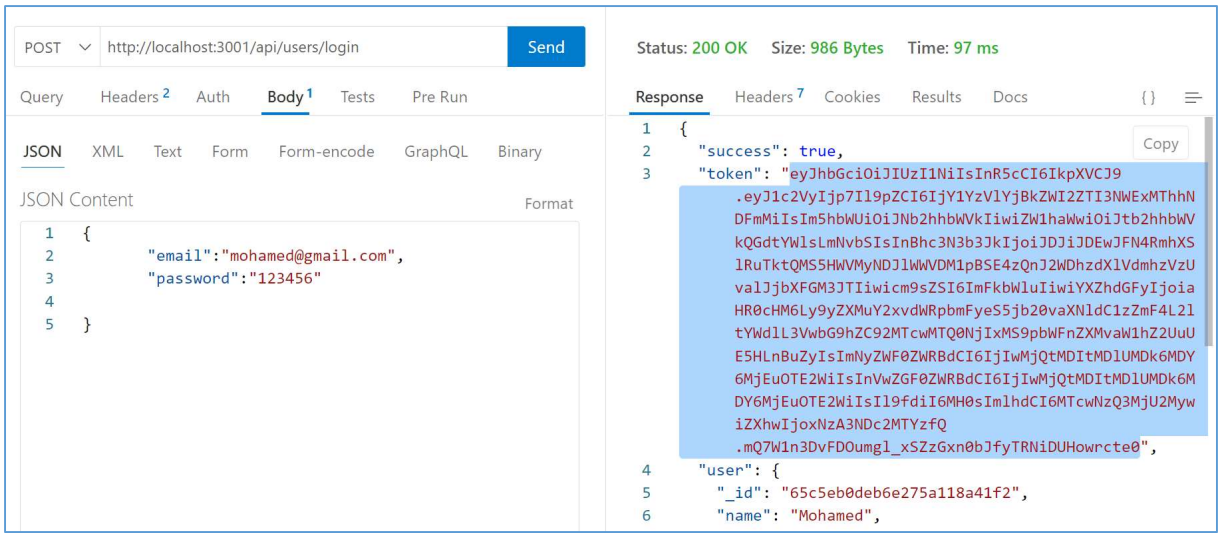


Donne :



La réponse est « Unauthorized » car maintenant la requête passe par le middleware « auth » qui vérifie s'il existe un token valide pour pouvoir donner le résultat.

b- Login : Copier le contenu du Token



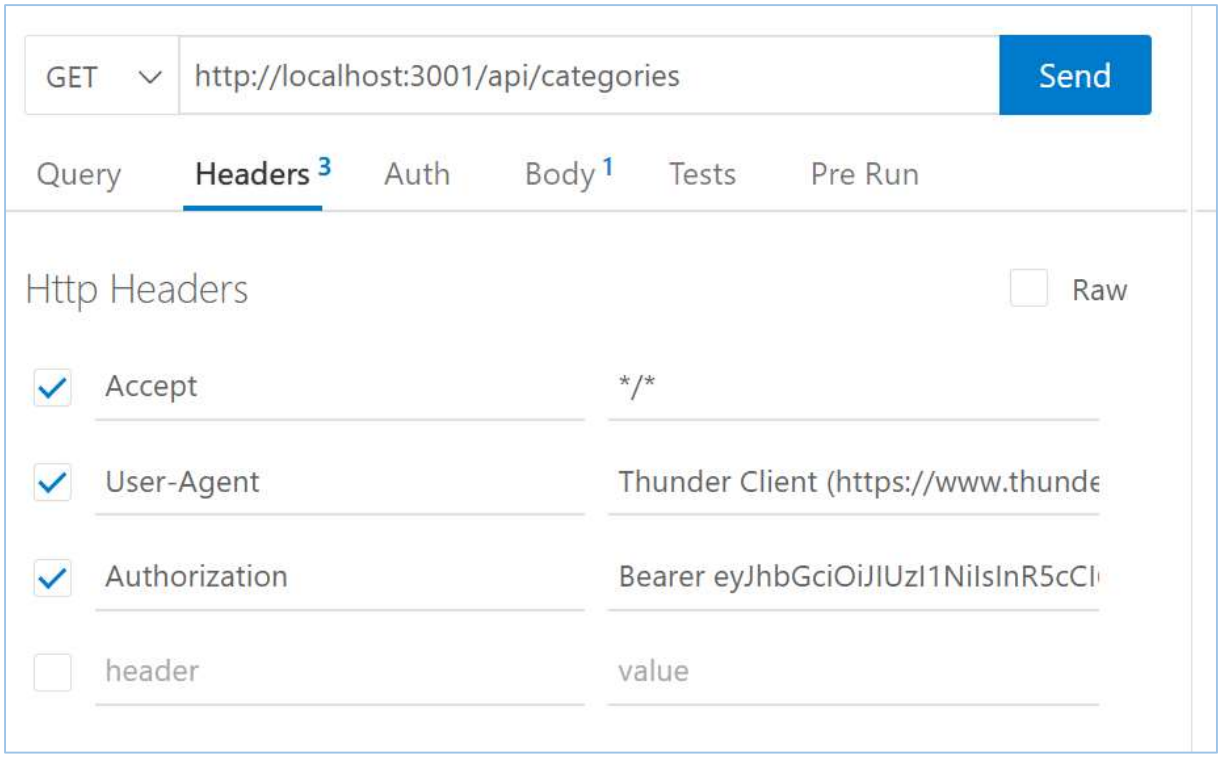
c- Puis dans la requête http://localhost:3001/api/categories

Aller dans la rubrique Header

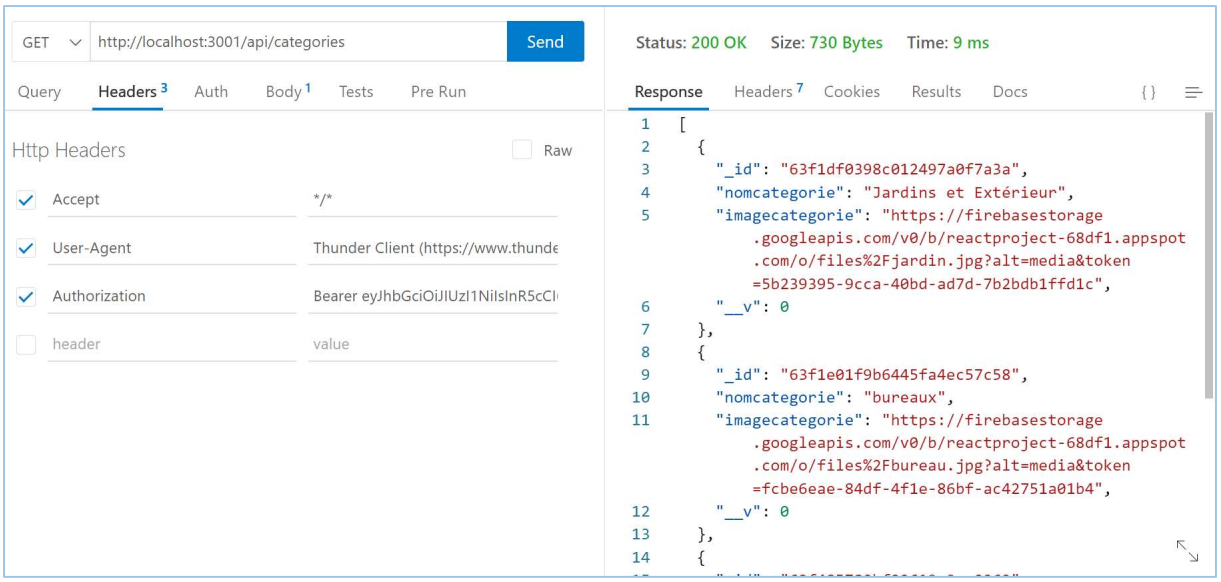
Ajouter la ligne

Authorization

Puis écrire ***Bearer*** et coller le contenu du token après un espace



On aura notre résultat :



V. Refresh

Refresh token est utilisé pour générer un nouveau jeton d'accès. En règle générale, si le jeton d'accès a une date d'expiration, une fois qu'il expire, l'utilisateur doit s'authentifier à nouveau pour obtenir un jeton d'accès. Avec le jeton d'actualisation, cette étape peut être ignorée et avec une demande à l'API, obtenez un nouveau jeton d'accès qui permet à l'utilisateur de continuer à accéder aux ressources de l'application.

9. Dans le fichier .env

Ajoutez :

```
REFRESH_TOKEN=qwerty
```

10. Dans la route « user.route.js » Ajoutez ces lignes qui permettent de faire le refresh.

```
const express = require('express');
const router = express.Router();
const User = require('../models/user.js');
const  jwt  = require('jsonwebtoken');
const bcrypt = require('bcrypt');

//Register

router.post('/register', async (req, res, )=> {

    const{name,email,password,role,avatar}=req.body;

    const user = await User.findOne({ email })
```

```
    if (user) return res.status(404).send({ success: false, message: "Account
already exists" })

    const salt=await bcrypt.genSalt(10);
    const hash=await bcrypt.hash(password,salt);

    const newUser=new User({
      name:name,
      email:email,
      password:hash,
      role:role,
      avatar:avatar
        });
    try {
           await newUser.save();

        return res.status(201).send({ success: true, message: "Account created
successfully", user: newUser })
       } catch (error) {
           res.status(409).json({ message: error.message });
       }

});

//Generate Token
const generateToken=(user) =>{
    return jwt.sign({user}, process.env.TOKEN, { expiresIn: '60s' });
  }

// login
router.post('/login', async (req, res) =>  {
    try {
        let { email, password } = req.body

        if (!email || !password) {
            return res.status(404).send({ success: false, message: "All fields
are required" })
        }

        let user = await User.findOne({ email })

        if (!user) {

            return res.status(404).send({ success: false, message: "Account
doesn't exists" })

        } else {

      let isMatch = await bcrypt.compare(password, user.password)
```

```
      if(!isMatch) {res.status(400).json({success: false, message:'Please
verify your credentials'}); return;}

       const token = generateToken(user);
       const refreshToken = generateRefreshToken(user);

       res.status(200).json({
        success: true,
        token,
        refreshToken,
        user
    })
   }
 } catch (error) {
    res.status(404).json({ message: error.message });
}
});

// Refresh
function generateRefreshToken(user) {
    return jwt.sign({user}, process.env.REFRESH_TOKEN, { expiresIn: '1y' });
  }

  //Refresh Route

  router.post('/refreshToken', async (req, res, )=> {
  const refreshtoken = req.body.refreshToken;
    if (!refreshtoken) {
     return res.status(404).json({ success: false,message: 'Token Not Found'
});
        }
    else {
        jwt.verify(refreshtoken, process.env.REFRESH_TOKEN, (err, user) => {
          if (err) {
            return res.status(406).json({success: false, message:
'Unauthorized Access' });
          }
          else {
           const token = generateToken(user);

           const refreshToken = generateRefreshToken(user);

          res.status(200).json({
           token,
           refreshToken
         })
            }
        });
       }
```

```
});

module.exports = router;
```

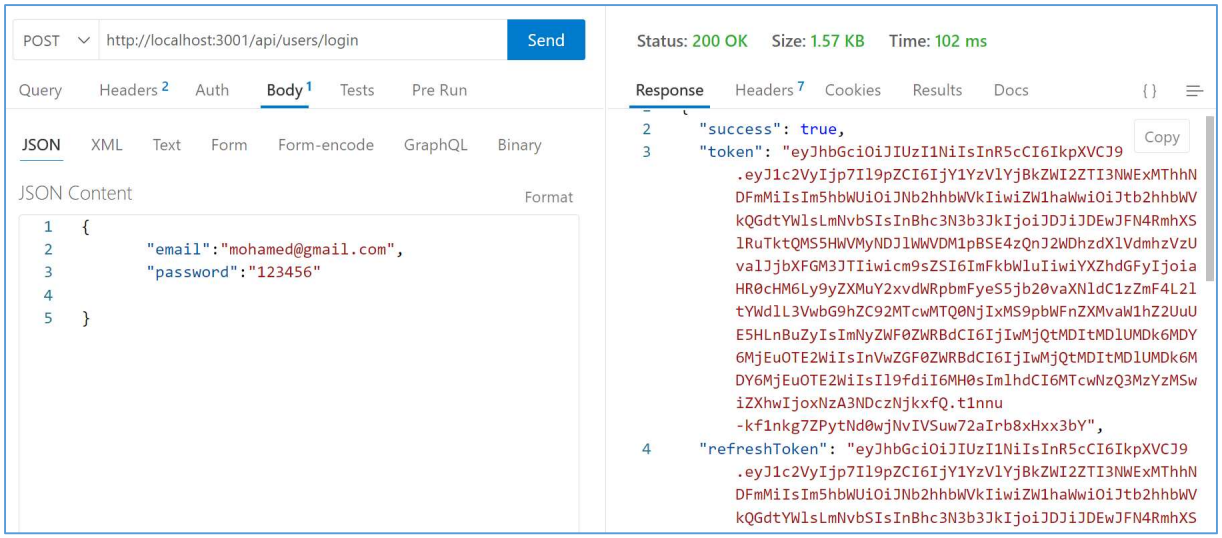
Nous avons créé la route RefreshToken qu'on va lui associer une route par la suite et qui permet de générer un nouveau token lors de l'appel de refresh.

Nous avons aussi créé la méthode generateRefreshToken qui permet de générer un token qui expire dans 1 an.

On modifie la durée du token qui expire dans 60 seconde.

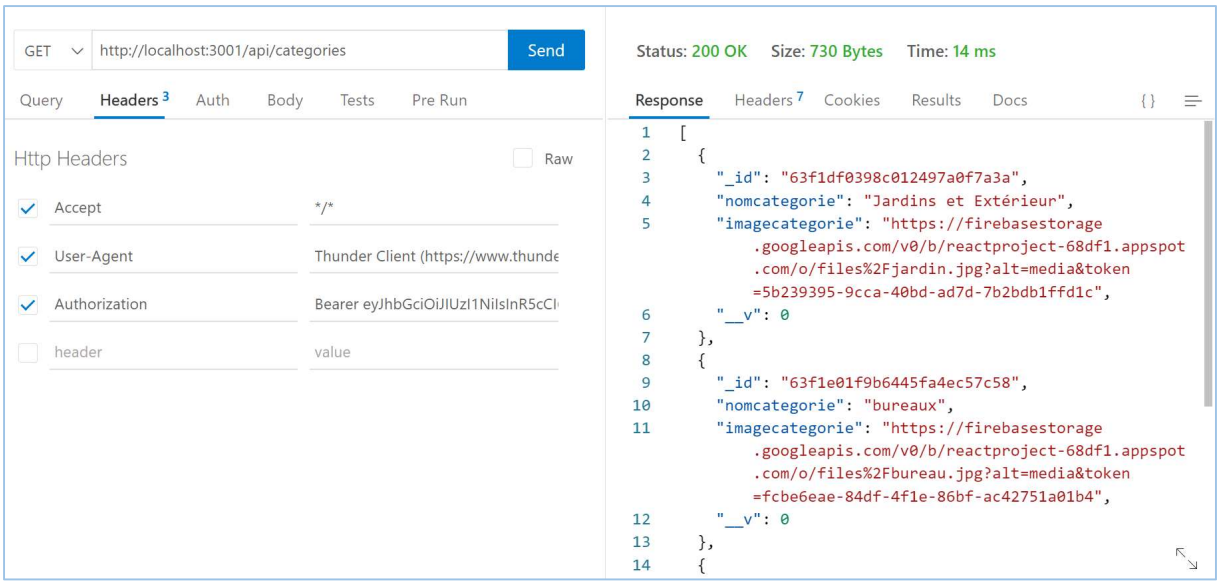
Tester l'API

a- D'abord le login d'un utilisateur déjà enregistré :



Deux token sont générés l'accessToken et le refreshToken.

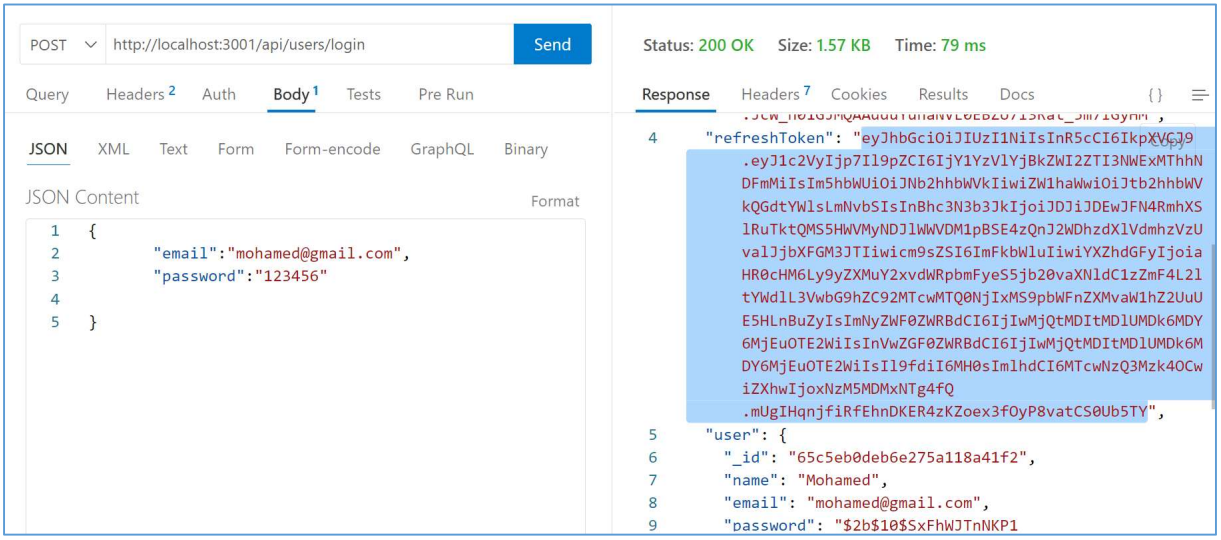
b- Avec la valeur du **token** dans le header, on va tester l'affichage des catégories :



Ce qui a été fait avec succès.

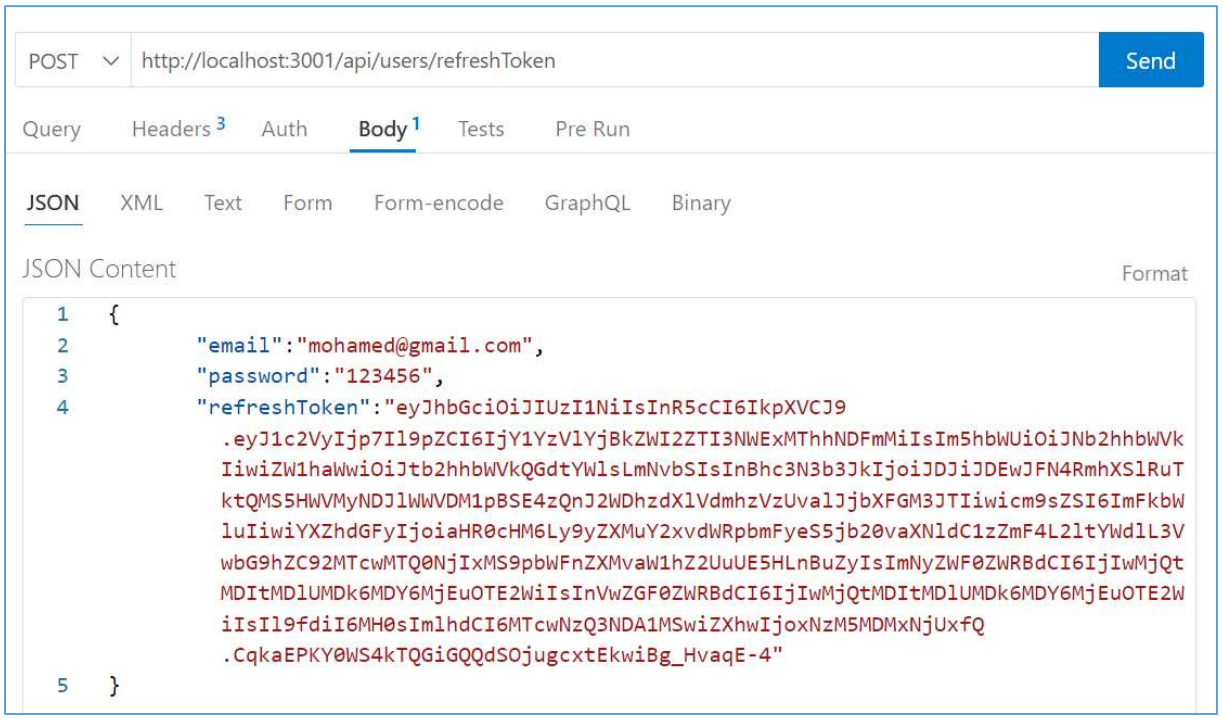
c- Mais **au bout de 60 secondes** lorsque la durée du jeton a expiré en renouvelant la requête. On a eu « Unauthorized » comme résultat. **Le jeton n'est plus valable**.

d- Dans la requête de ***refreshToken*** on doit envoyer la valeur de **refreshToken** obtenue à partir de la requête login (qu'on a copié)

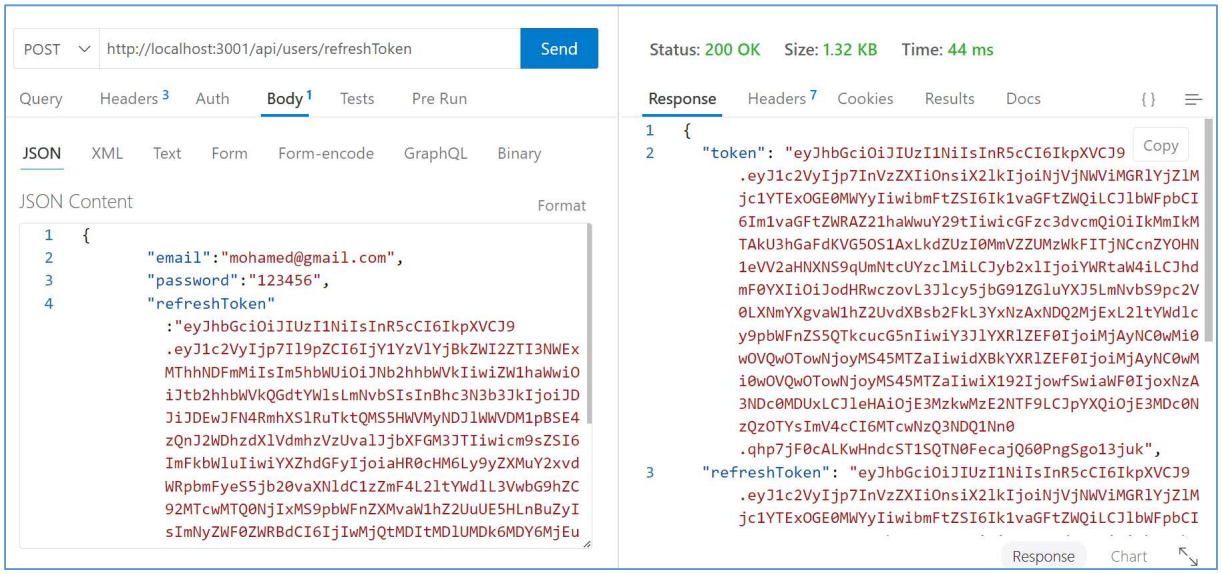


http://localhost:3001/api/users/refreshToken

Requête POST

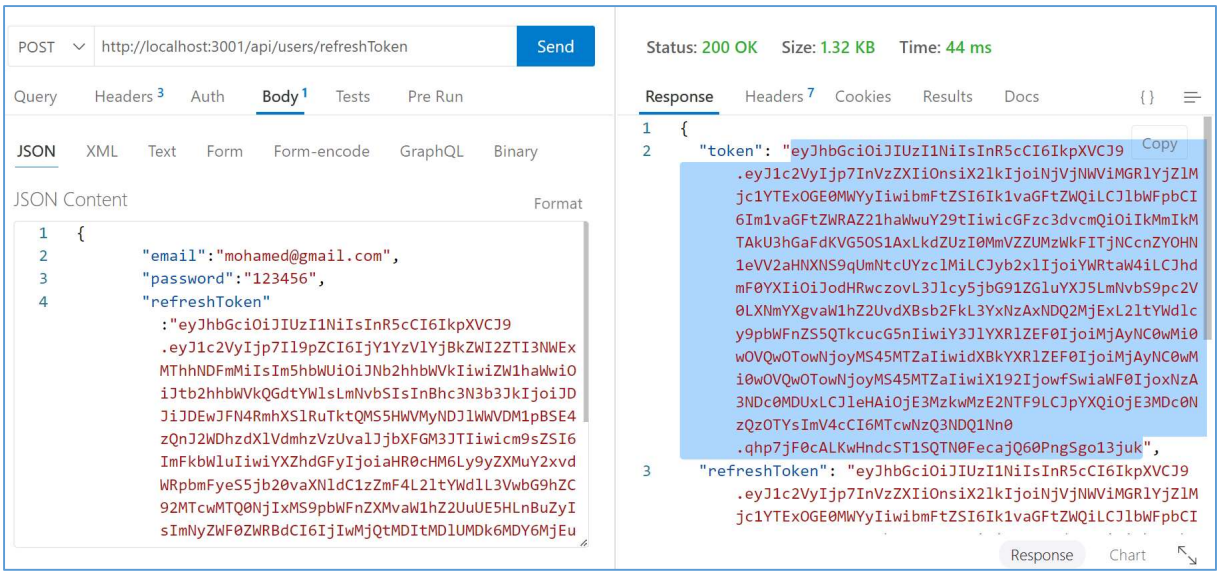


Puis exécuter la requête en cliquant sur Send.

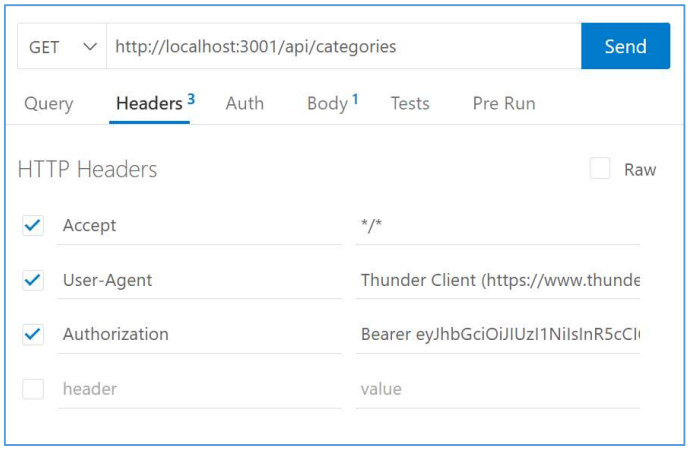


Un nouveau **token** est généré.

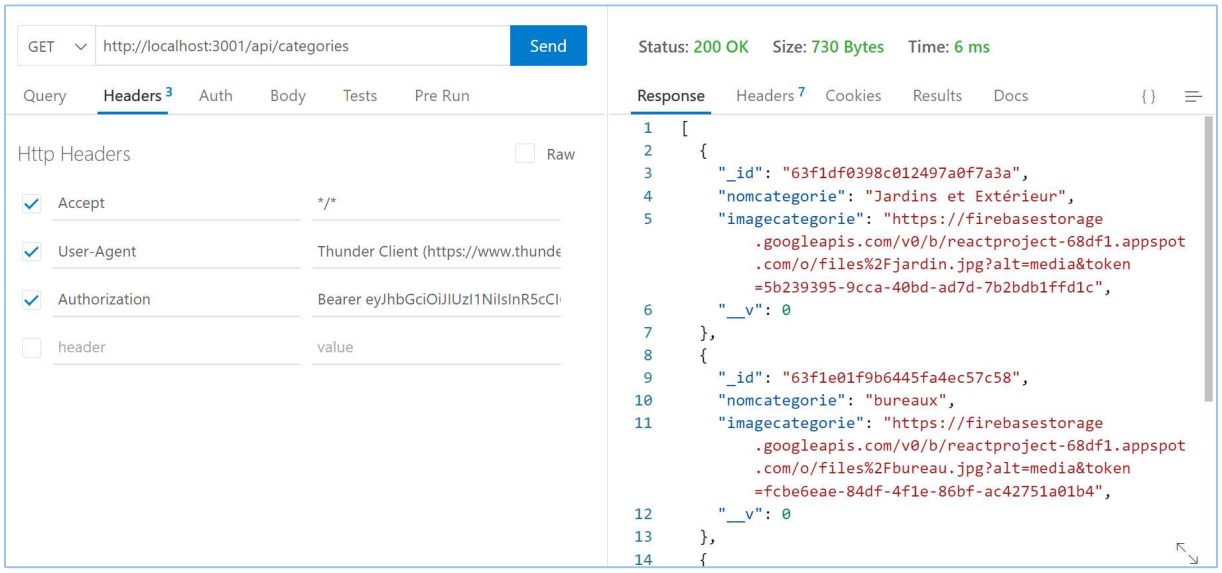
e- On copie la valeur du **token**



f- On le colle dans l'entête de la requête GET de categories.



g- Et on exécute Send.



On pourra avoir le résultat avec succès de nouveau.