

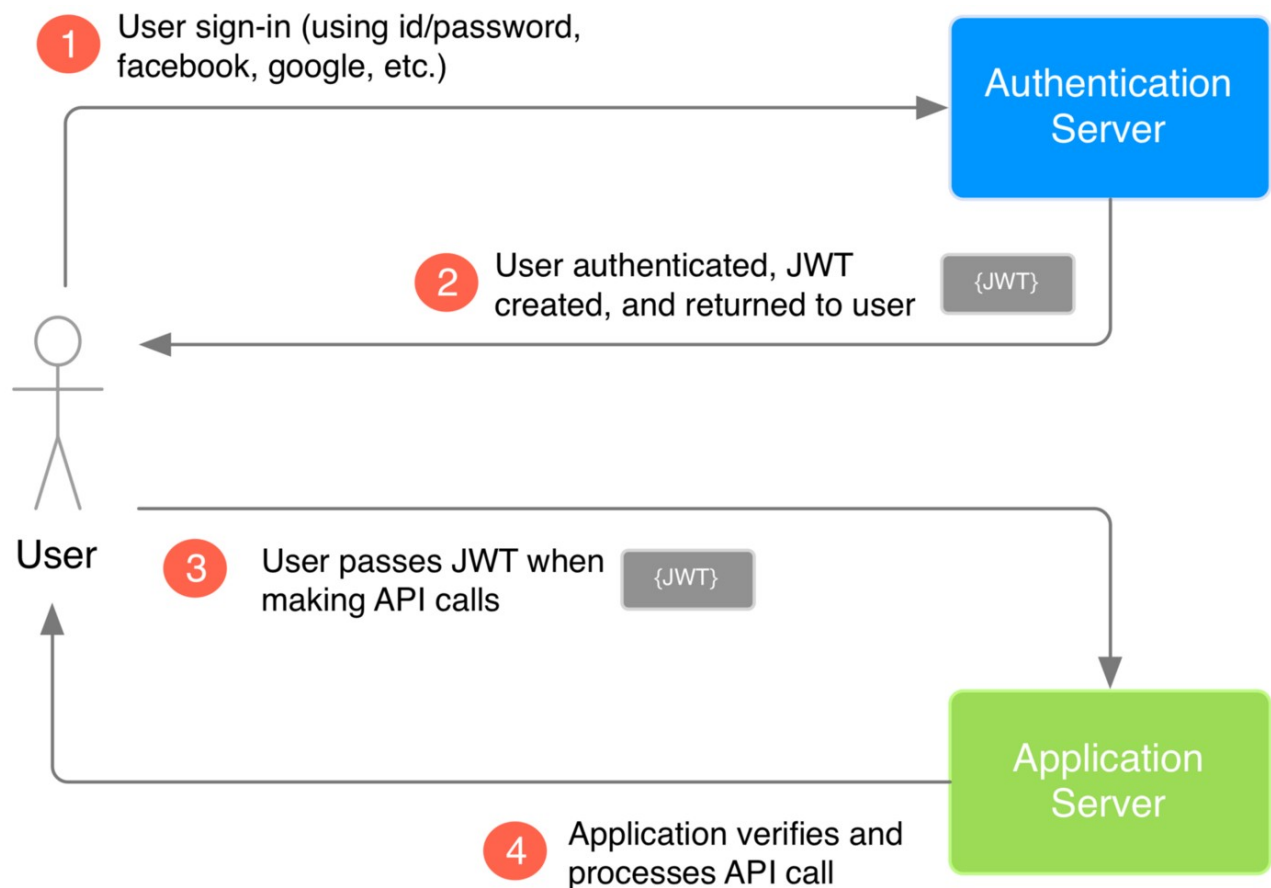
# TP: Authentification et autorisation basées sur des jetons

## Objectifs

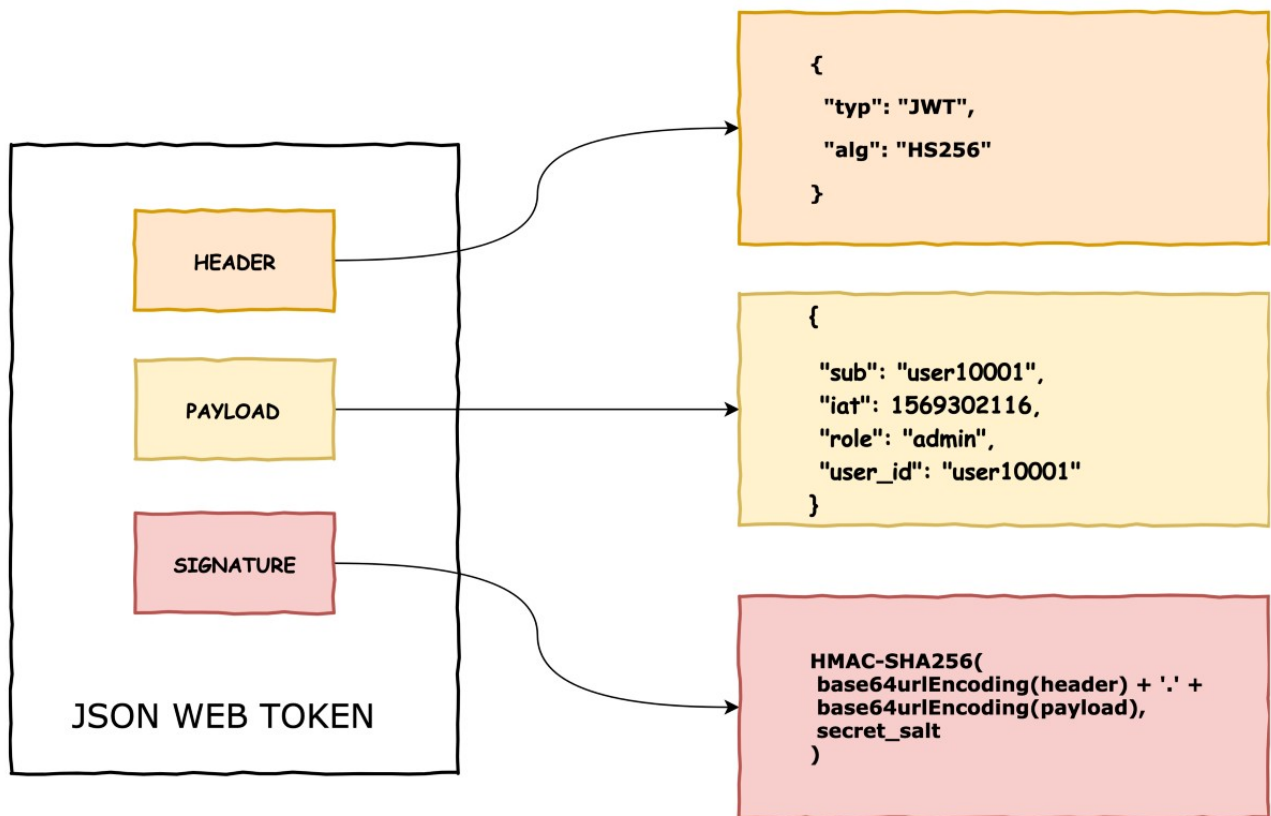
Dans ce TP, nous allons créer un exemple d'API Node.js Express REST qui prend en charge l'authentification basée sur des jetons avec JWT (JSONWebToken).

## Authentification basée sur des jetons

Par rapport à l'authentification basée sur la session qui doit stocker la session sur le cookie, le grand avantage de l'authentification basée sur le jeton est que nous stockons le JSON WEB TOKEN (JWT) côté client: stockage local pour le navigateur, et préférences partagées pour Android...



Il y a trois parties importantes d'un JWT: **Header, Payload, Signature**. Ensemble, ils sont combinés en une structure standard: **header.payload.signature**.



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

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

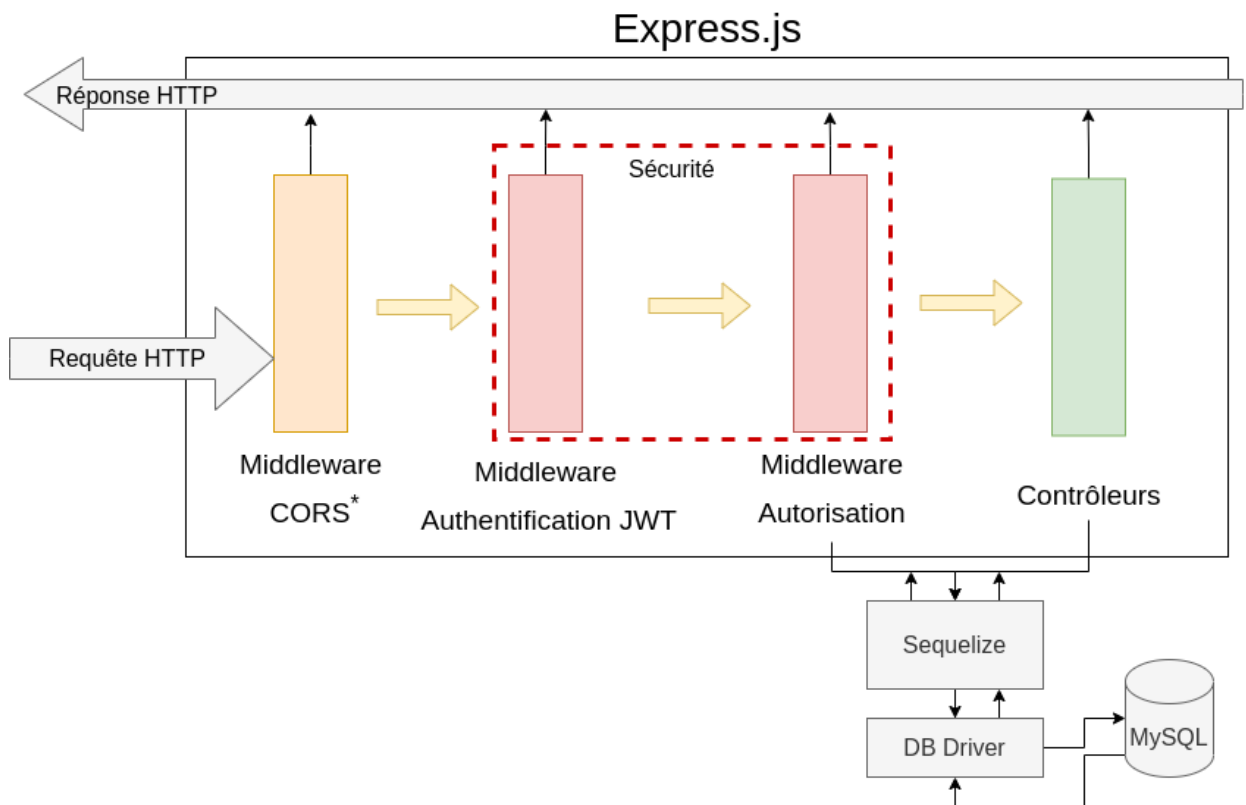
Ou uniquement dans l'en-tête **x-access-token**:

```
x-access-token: [header].[payload].[signature]
```

Voici les API que nous devons fournir:

Méthodes	URLs	Actions
POST	/auth/signup	créer un nouveau compte utilisateur
POST	/auth/signin	connecter un compte utilisateur
GET	/test/all	récupérer du contenu public
GET	/test/user	accéder au contenu de l'utilisateur enregistré

Vous pouvez avoir un aperçu de notre application Node.js Express JWT avec le schéma ci-dessous:



\*Cross-origin resource sharing (CORS) est un mécanisme qui permet d'accéder à des ressources restreintes sur une page Web à partir d'un autre domaine en dehors du domaine à partir duquel la première ressource a été servie

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:

- Middleware d'authentification JWT: vérifiez l'inscription, vérifiez le jeton
- Middleware d'autorisation: vérifiez les rôles de l'utilisateur avec un enregistrement dans la base de données (**Cette partie sera ignorée dans ce TP. Cependant, cela doit être pris en compte lors de la création d'applications où les utilisateurs sont divisés en catégories (utilisateurs normaux, administrateurs, etc.)**)

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

Les contrôleurs interagissent avec la base de données MySQL via Sequelize et envoient une réponse HTTP (token, informations utilisateur...) au client.

## Créer une application Node.js

Créez un dossier pour le projet, puis initialisez l'application Node.js avec un fichier **package.json**:

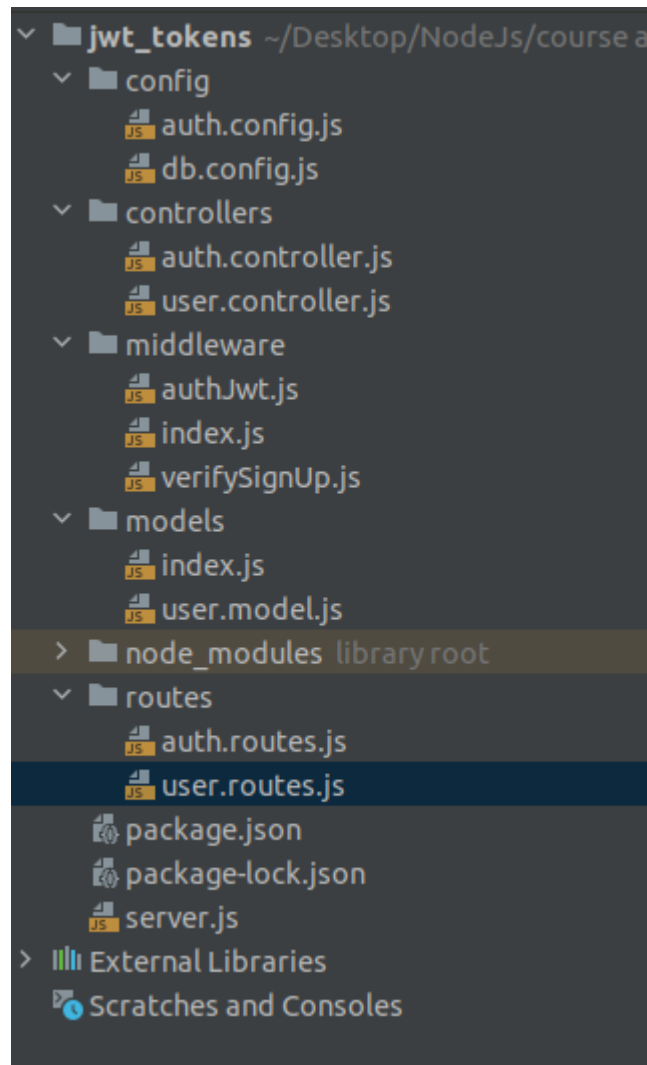
**\$ npm init -yes**

Nous devons installer les modules nécessaires : **express**, **cors**, **body-parser**, **sequelize**, **mysql2**, **jsonwebtoken** et **bcryptjs**.

Exécutez la commande:

```
$ npm install express sequelize mysql2 body-parser cors  
jsonwebtoken bcryptjs --save
```

Nous devons ensuite créer la structure des fichiers illustrée dans la figure ci-dessous:



```
$ mkdir config
```

```
$ mkdir controllers
```

```
$ mkdir middleware
```

```
$ mkdir models
```

```
$ mkdir routes
```

```
$ touch config/auth.config.js
```

```
$ touch config/db.config.js
```

```
$ touch controllers/auth.controller.js
$ touch controllers/user.controller.js
$ touch middleware/authJwt.js
$ touch middleware/index.js
$ touch middleware/verifySignUp.js
$ touch models/index.js
$ touch models/user.model.js
$ touch routes/auth.routes.js
$ touch routes/user.routes.js
$ touch server.js
```

- **config**
  - *db.config.js*: configurer la base de données MySQL et sequelize
  - *auth.config.js*: configurer la clé d'authentification
- **routes**
  - *auth.routes.js*: inscription et connexion (login) POST
  - *user.routes.js*: OBTENIR des ressources publiques et protégées GET
- **middlewares**
  - *verifySignUp.js*: vérifier si l'email existe
  - *authJwt.js*: vérifier le jeton (token)
- **controllers**
  - *auth.controller.js*: gérer les actions d'inscription et de connexion
  - *user.controller.js*: renvoie le contenu public et protégé
- **models**
  - *user.model.js*: modèle Sequelize

## Base de données

Nous allons utiliser la base de données utilisée dans TP4: **bdd\_node\_1**.

**IMPORTANT:** assurez-vous que le mot de passe de la colonne est "longtext" afin de stocker le texte chiffré. Sinon, téléchargez **db.sql** de la semaine 8 (cours-info) et remplacez l'ancienne base de données par celle du fichier **db.sql** (en refaisant "**source db.sql**").

```
mysql> describe users;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id     | int(11) | NO | PRI | NULL | auto_increment |
| firstName | varchar(45) | NO | | NULL |
| lastName | varchar(45) | NO | | NULL |
| emailId | varchar(45) | NO | UNI | NULL |
| password | longtext | NO | | NULL |
+-----+-----+-----+-----+-----+-----+
5 rows in set (0,01 sec)
```

## Semaine 8 (Partiel devrait être envoyé cette semaine):

La date limite pour présenter le Partiel est le 12 novembre 18h

- TD: Authentification des comptes utilisateurs - partie 2
- db.sql (à télécharger avant de commencer avec le TD) ⇒ **bdd\_node\_1**
- TP 8: Ajout de l'authentification des utilisateurs au projet du Partiel **(IMPORTANT : CE TP EST NOTÉ)**
- Ressources:
  - <http://www.passportjs.org/packages/passport-local/> (Passport local)
  - <http://www.passportjs.org/packages/passport-google-oauth2/> (Passport Google OAuth2)
  - <http://www.passportjs.org/docs/facebook/> (Passport Facebook)
  - <https://www.youtube.com/watch?v=KIE9RAOI9KA> (HOW TO: Implement Facebook Login OAuth in Web App)
  - <https://www.youtube.com/watch?v=Q0a0594tOrc> (NodeJS & Express - Google OAuth2 using PassportJS)

Sujets abordés:

- Authentification des utilisateurs à l'aide de Passport

## Configurer le serveur Web Express

Ajoutez ce qui suit au fichier **server.js**:

```
const express = require("express");
const bodyParser = require("body-parser");
const cors = require("cors");
const app = express();
```

```
let corsOptions = {
  origin: "http://localhost:3000"
};
```

```
app.use(cors(corsOptions));
// analyser les requêtes de type de contenu - application/json
app.use(bodyParser.json());
// analyser les requêtes de type de contenu - application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: true }));
```

```
app.get("/", (req, res) => {
  res.json({ message: "Bienvenue dans l'application TP 10: Jetons JWT" });
});
```

```
// définir le port, écouter les requêtes
const PORT = process.env.PORT || 3000;
app.listen(PORT, () => {
  console.log(`Serveur écoute sur le port ${PORT}.`);
});
```

- **Express** est pour construire l'API Rest
- **body-parser** aide à analyser la requête et à créer l'objet req.body
- **cors** fournit un middleware Express pour activer CORS

Vous pouvez maintenant exécuter l'application avec la commande: **node server.js**

Ouvrez votre navigateur avec l'url **http://localhost:3000/**, vous verrez le message de bienvenue.

# Configurer la base de données MySQL

Ouvrez **db.config.js** dans le répertoire **config** et ajoutez ce qui suit:

```
module.exports = {
  hostname: "localhost",
  user: "user",
  password: "password",
  database: "bdd_node_1",
  dialect: "mysql",
  port: 3306,
  pool: {
    max: 10,
    min: 0,
    acquire: 30000,
    idle: 10000
  }
};
```

Assurez-vous de modifier la configuration.

pool est facultatif, il sera utilisé pour la configuration du pool de connexions Sequelize. Pour plus de détails, veuillez visiter la référence API pour le constructeur Sequelize:

<https://sequelize.org/master/class/lib/sequelize.js~Sequelize.html#instance-constructor-constructor>

## Définir le modèle Sequelize

Ouvrez **models/user.model.js** et ajoutez ce qui suit:

```
module.exports = function(sequelize, Sequelize) {
  const User = sequelize.define('user', {
    id: { autoIncrement: true, primaryKey: true, type: Sequelize.INTEGER, allowNull: false },
    firstName: { type: Sequelize.STRING, notEmpty: true },
    lastName: { type: Sequelize.STRING, notEmpty: true },
    emailId: { type: Sequelize.STRING, validate: { isEmail: true } },
    password: { type: Sequelize.STRING, allowNull: false },
  }, {
    tableName: 'users'
  });
  return User;
};
```

Le modèle **User** Sequelize représente la table des utilisateurs dans la base de données MySQL.

Après avoir initialisé Sequelize, nous n'avons pas besoin d'écrire les fonctions CRUD, Sequelize les prend toutes en charge:

- créer un nouvel utilisateur: **create(object)**
- trouver un utilisateur par identifiant: **findByPk(id)**
- trouver un utilisateur par e-mail: **findOne({ where: { emailId: ... } })**
- obtenir tous les utilisateurs: **findAll()**
- rechercher tous les utilisateurs par nom d'utilisateur: **findAll({ where: { username: ... } })**

# Initialiser Sequelize

Ouvrez **models/index.js** et ajoutez ce qui suit:

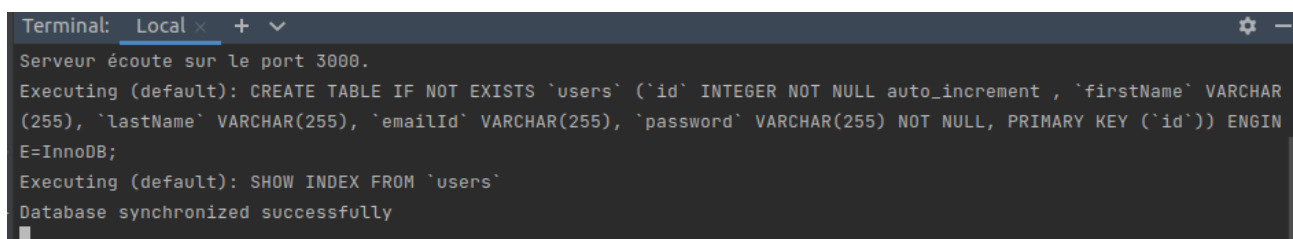
```
const config = require("../config/db.config.js");
const Sequelize = require("sequelize");
const sequelize = new Sequelize(
  config.database,
  config.user,
  config.password,
  {
    host: config.hostname,
    port: config.port,
    dialect: config.dialect,
    pool: {
      max: config.pool.max,
      min: config.pool.min,
      acquire: config.pool.acquire,
      idle: config.pool.idle
    },
    define: {
      timestamps: false
    }
  }
);
const db = {};
db.Sequelize = Sequelize;
db.sequelize = sequelize;
db.user = require("../models/user.model.js")(sequelize, Sequelize);
module.exports = db;
```

Nous voulons maintenant ajouter la méthode **sync()** dans **server.js**:

```
const db = require("./models");
db.sequelize.sync().then(() => {
  console.log('Database synchronized successfully');
});
```

**Remarque:** lorsque vous transmettez un dossier à `require()` de Node, il recherchera dans `package.json` pour un point de terminaison. Si cela n'est pas défini, il recherche `index.js` et enfin `index.node` (un format d'extension c++). Ainsi, `index.js` est très probablement le point d'entrée pour importer un module.

Après avoir exécuté votre application, vous devriez recevoir le message que la base de données est synchronisée avec succès.



```
Terminal: Local x + v
Serveur écoute sur le port 3000.
Executing (default): CREATE TABLE IF NOT EXISTS `users` (`id` INTEGER NOT NULL auto_increment , `firstName` VARCHAR (255), `lastName` VARCHAR(255), `emailId` VARCHAR(255), `password` VARCHAR(255) NOT NULL, PRIMARY KEY (`id`)) ENGINE=InnoDB;
Executing (default): SHOW INDEX FROM `users`
Database synchronized successfully
```



## Configurer la clé d'authentification

Les fonctions **jsonwebtoken** telles que **verify()** ou **sign()** utilisent un algorithme qui a besoin d'une clé secrète (sous forme de chaîne) pour coder et décoder le jeton.

Dans le dossier **config**, ouvrez **auth.config.js** et ajoutez ce qui suit:

```
module.exports = {  
  secret: "darth-vader" // vous pouvez modifier cette valeur  
};
```

## Créer des fonctions middleware

Ouvrez **middleware/verifySignUp.js** afin de créer une fonction qui vérifie si un "email" existe dans la base de données avant de créer un nouvel utilisateur. Ajoutez ce qui suit:

```
const db = require("../models");  
const User = db.user;  
checkDuplicateUsername = (req, res, next) => {  
  // Username  
  User.findOne({  
    where: {  
      emailId: req.body.emailId  
    }  
  }).then(user => {  
    if (user) {  
      res.status(400).send({  
        message: "Échoué! Nom d'utilisateur est déjà utilisé!"  
      });  
      return;  
    }  
    next();  
  });  
};  
const verifySignUp = {  
  checkDuplicateUsername: checkDuplicateUsername  
};  
module.exports = verifySignUp;
```

Ensuite, nous devons vérifier si un jeton est fourni, ou s'il est légal ou non. Nous obtenons le jeton de "x-access-token" ou "authorization" des en-têtes HTTP, puis nous utilisons la fonction **verify()** de **jsonwebtoken**. Ouvrez **middleware/authJwt.js** et ajoutez ce qui suit:

```
const jwt = require("jsonwebtoken");  
const config = require("../config/auth.config.js");  
  
verifyToken = (req, res, next) => {  
  // if you are using postman or creating a client app,  
  // you can set the token in the x-access-token inside the header  
  let token = req.headers["x-access-token"];  
  if (!token) {  
    // if using swagger, the token could be inside "authorization: Bearer <token>"  
    token = req.headers["authorization"];  
    if (!token) {  
      return res.status(403).send({  
        message: "Aucun jeton fourni!"  
      });  
    }  
  }  
}
```

```

        token = token.split(" ")[1]; // remove "Bearer " from the value
    }
    jwt.verify(token, config.secret, (err, decoded) => {
        if (err) {
            return res.status(401).send({
                message: "Non autorisé!"
            });
        }
        req.userId = decoded.id;
        next();
    });
};
const authJwt = {
    verifyToken: verifyToken
};
module.exports = authJwt;

```

Ouvrez **middleware /index.js** et ajoutez ce qui suit:

```

const authJwt = require("../authJwt");
const verifySignUp = require("../verifySignUp");
module.exports = {
    authJwt,
    verifySignUp
};

```

## Créer les contrôleurs

### Contrôleur d'authentification

Il y a 2 fonctions principales pour l'authentification:

- **signup**: créer un nouvel utilisateur dans la base de données
- **signin**: trouver le nom d'utilisateur de la requête dans la base de données, s'il existe

comparer le mot de passe avec le mot de passe dans la base de données en utilisant **bcrypt**, s'il est correct générer un jeton en utilisant **jsonwebtoken**.

Dans le dossier **controllers**, ouvrez **auth.controller.js** et ajoutez ce qui suit:

```

const db = require("../models");
const config = require("../config/auth.config");
const User = db.user;
const jwt = require("jsonwebtoken");
const bcrypt = require("bcryptjs");
exports.signup = (req, res) => {
    // Enregistrer l'utilisateur dans la base de données
    User.create({
        firstName: req.body.firstName,
        lastName: req.body.lastName,
        emailId: req.body.emailId,
        password: bcrypt.hashSync(req.body.password, 8)
    })
    .then(user => {
        res.send({ message: "L'utilisateur a été enregistré avec succès!" });
    })
};

```

```

        .catch(err => {
            res.status(500).send({ message: err.message });
        });
};
exports.signin = (req, res) => {
    console.log(req.body);
    User.findOne({
        where: {
            emailId: req.body.emailId
        }
    })
    .then(user => {
        if (!user) {
            return res.status(404).send({ message: "Utilisateur non trouvé." });
        }
        let passwordIsValid = bcrypt.compareSync(
            req.body.password,
            user.password
        );
        if (!passwordIsValid) {
            return res.status(401).send({
                accessToken: null,
                message: "Mot de passe incorrect!"
            });
        }
        let token = jwt.sign({ id: user.id }, config.secret, {
            expiresIn: 86400 // 24 heures
        });
        res.status(200).send({
            id: user.id,
            username: user.emailId,
            accessToken: token
        });
    })
    .catch(err => {
        res.status(500).send({ message: err.message });
    });
};

```

On considère que le nom d'utilisateur (username) dans cet exemple est l'adresse mail de l'utilisateur.

Ouvrez maintenant **user.controller.js** pour ajouter deux fonctions qui permettent d'accéder à un contenu public et à un contenu privé:

```

exports.allAccess = (req, res) => {
    res.status(200).send("Contenu public.");
};
exports.userAccess = (req, res) => {
    res.status(200).send("Contenu utilisateur.");
};

```

## Définir les routes

Lorsqu'un client envoie une requête pour un point de terminaison à l'aide d'une requête HTTP (GET, POST, PUT, DELETE), nous devons déterminer comment le serveur répondra en configurant les routes.

Nous pouvons séparer nos routes en 2 parties: pour l'authentification et pour accéder aux ressources publiques et privées.

Authentification:

- POST /auth/signup
- POST /auth/signin

Dans le dossier **routes**, ouvrez **auth.routes.js** et ajoutez ce qui suit:

```
const { verifySignUp } = require("../middleware");  
const controller = require("../controllers/auth.controller");  
module.exports = function(app) {  
  app.use(function(req, res, next) {  
    res.header(  
      "Access-Control-Allow-Headers",  
      "x-access-token, Origin, Content-Type, Accept"  
    );  
    next();  
  });  
};
```

```
app.post(  
  "/auth/signup",  
  [  
    verifySignUp.checkDuplicateUsername,  
  ],  
  controller.signup  
);
```

```
app.post("/auth/signin", controller.signin);  
};
```

Vous pouvez remarquer que nous utilisons le middleware **verifySignUp** avant d'inscrire un nouvel utilisateur.

Dans le dossier **routes**, ouvrez **user.routes.js** et ajoutez ce qui suit:

```
const { authJwt } = require("../middleware");  
const controller = require("../controllers/user.controller");  
module.exports = function(app) {  
  app.use(function(req, res, next) {  
    res.header(  
      "Access-Control-Allow-Headers",  
      "x-access-token, Origin, Content-Type, Accept"  
    );  
    next();  
  });  
  app.get("/test/all", controller.allAccess);  
};
```

```
app.get(  
  "/test/user",  
  [authJwt.verifyToken],  
  controller.userAccess  
);
```

```
};
```

Ouvrez **server.js** et ajoutez les routes:

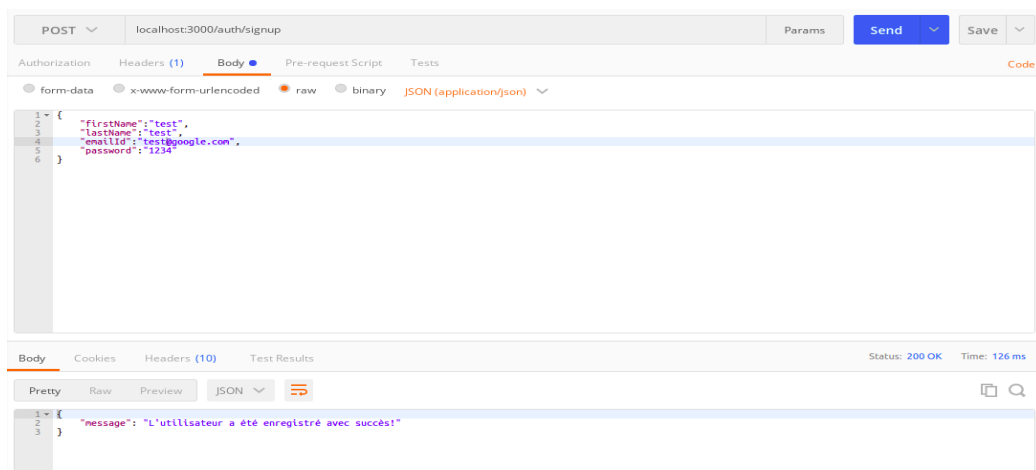
```
// routes
```

```
require('./routes/auth.routes')(app);  
require('./routes/user.routes')(app);
```

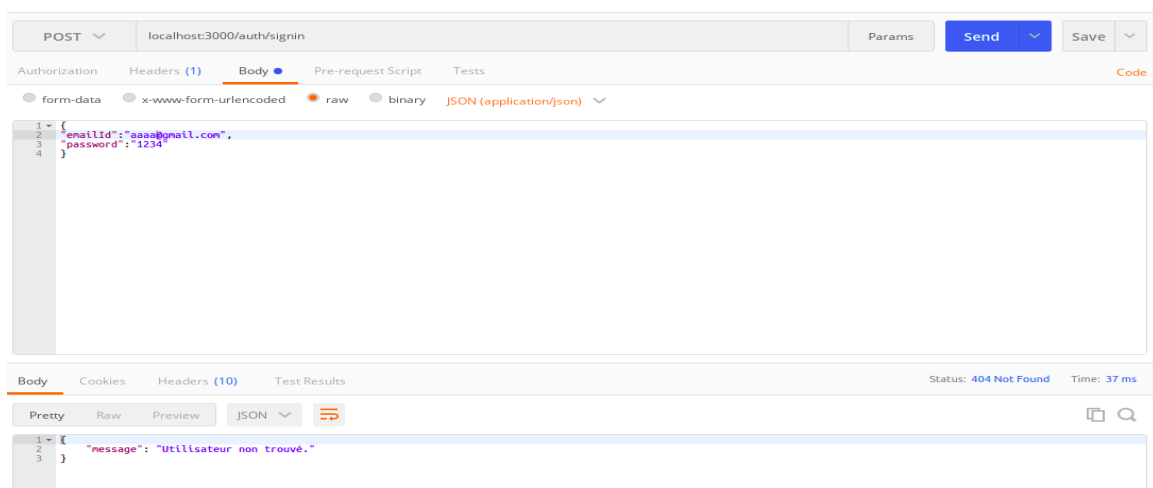
```
// route route  
app.get("/", (req : Request<P, ResBody, ReqBody, ReqQuery, Locals> , res : Response) => {  
  res.json( body: { message: "Bienvenue dans L'application TP 10: J" );  
});  
  
// routes  
require('./routes/auth.routes')(app);  
require('./routes/user.routes')(app);  
  
// définir le port, écouter les requêtes  
const PORT = process.env.PORT || 3000;  
app.listen(PORT, hostname: () => {  
  console.log(`Serveur écoute sur le port ${PORT}.`);  
});
```

## Testez votre application avec Postman

Création d'un nouvel utilisateur:



Lorsque vous essayez de vous connecter avec un e-mail ou un mot de passe erroné, vous devriez obtenir une erreur:



La première fois que vous vous connectez avec succès, vous obtiendrez le jeton:

POST localhost:3000/auth/signin

Body

```
1 {
2   "emailId": "jazar@gmail.com",
3   "password": "1234"
4 }
```

Status: 200 OK Time: 54 ms

Body

```
1 {
2   "id": 142,
3   "username": "jazar@gmail.com",
4   "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTQyLCJpYXQiOiJlZ2MzNDQ4MzEsInV4cCI6MTYzODIyODIzMX0.F-t5V7eYxP8XqG6yRQAvh-6FVRfdZdJr4pn6prM"
5 }
```

Si vous essayez d'accéder à une page privée sans fournir le jeton, vous devriez obtenir un message d'erreur:

GET localhost:3000/test/user

Type: No Auth

Status: 403 Forbidden Time: 25 ms

Body

```
1 {
2   "message": "Aucun jeton fourni!"
3 }
```

Vous devez ensuite ajouter le jeton à l'en-tête de la requête (x-access-token).

GET localhost:3000/test/user

Headers (2)

Key	Value	Description
<input checked="" type="checkbox"/> Content-Type	application/json	
<input checked="" type="checkbox"/> x-access-token	eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTQyLCJpYXQiOiJlZ2MzNDQ4MzEsInV4cCI6MTYzODIyODIzMX0.F-t5V7eYxP8XqG6yRQAvh-6FVRfdZdJr4pn6prM	

Status: 200 OK Time: 23 ms

Body

```
1 Contenu utilisateur.
```

## Ajouter la documentation Swagger

Installez swagger à l'aide de la commande suivante:

```
$ npm install --save swagger-jsdoc swagger-ui-express
```

Ouvrez **server.js** et importez swagger:

```
//ADD SWAGGER MODULES
const swaggerJsdoc = require("swagger-jsdoc");
const swaggerUi = require("swagger-ui-express");
```

Initialiser swagger:

```
app.use(bodyParser.json());
// analyser les requêtes de type de contenu - application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({ extended: true }));

/** Swagger Initialization - START */
const swaggerSpec = swaggerJsdoc({ options: {
  swaggerDefinition: {
    openapi: "3.0.2",
    info: {
      title: "TP EXPRESS JWT",
      version: "1.0.0",
      description: "API documentation",
      servers: ['http://localhost:${process.env.PORT || 3000}'],
    },
    components: {
      securitySchemes: {
        jwt: {
          type: "http",
          scheme: "bearer",
          in: "header",
          bearerFormat: "JWT"
        },
      },
    },
    security: [{
      jwt: []
    }],
  },
  apis: ["server.js", "./routes/*.js"],
});
app.use("/api-docs", swaggerUi.serve, swaggerUi.setup(swaggerSpec));
/** Swagger Initialization - END */

// route route
app.get("/", (req :Request<P, ResBody, ReqBody, ReqQuery, Locals>, res :Response<ResBody, Locals>) => {
  res.json( body: { message: "Bienvenue dans l'application TP 10: Jetons JWT" });
});
```

Vous pouvez trouver le code sur

[https://cours-info.iut-bm.univ-fcomte.fr/upload/supports/S3/web/cot%20serveur/tp10\\_swagger.txt](https://cours-info.iut-bm.univ-fcomte.fr/upload/supports/S3/web/cot%20serveur/tp10_swagger.txt)

## Semaine 10:

- [Presentation](#)
- [TD - Créer une application de microservices de base avec Node.JS](#)
- [TP 10 \(START\)](#)
- [tp10\\_swagger.txt](#)

Description de l'authentification du porteur (Bearer) dans Swagger:

Dans OpenAPI 3.0, l'authentification Bearer est un schéma de sécurité avec le **type: http** et le **scheme: bearer**. Vous devez d'abord définir le schéma de sécurité sous **components/securitySchemes**, puis utiliser le mot-clé **security** pour appliquer ce schéma à la portée souhaitée.

Maintenant, pour chaque route, ajoutez le YAML correspondant pour définir la documentation. Vous pouvez trouver le code en utilisant la même URL fournie ci-dessus.

```
app.post("/auth/signin", controller.signin);  
  
/**  
 * @swagger  
 * /auth/signin:  
 *   post:  
 *     description: Used to sign in user  
 *     tags:  
 *       - users  
 *     requestBody:  
 *       required: true  
 *       content:  
 *         application/json:  
 *           schema:  
 *             type: object  
 *             properties:  
 *               emailId:  
 *                 type: string  
 *                 minLength: 1  
 *                 maxLength: 50  
 *                 example: test@something.com  
 *               password:  
 *                 type: string  
 *                 minLength: 4  
 *                 maxLength: 50  
 *                 example: qBcd  
 *     security:  
 *       - jwt: []  
 *     responses:  
 *       '200':  
 *         description: Resource added successfully  
 *       '500':  
 *         description: Internal server error  
 *       '400':  
 *         description: Bad request  
 */
```

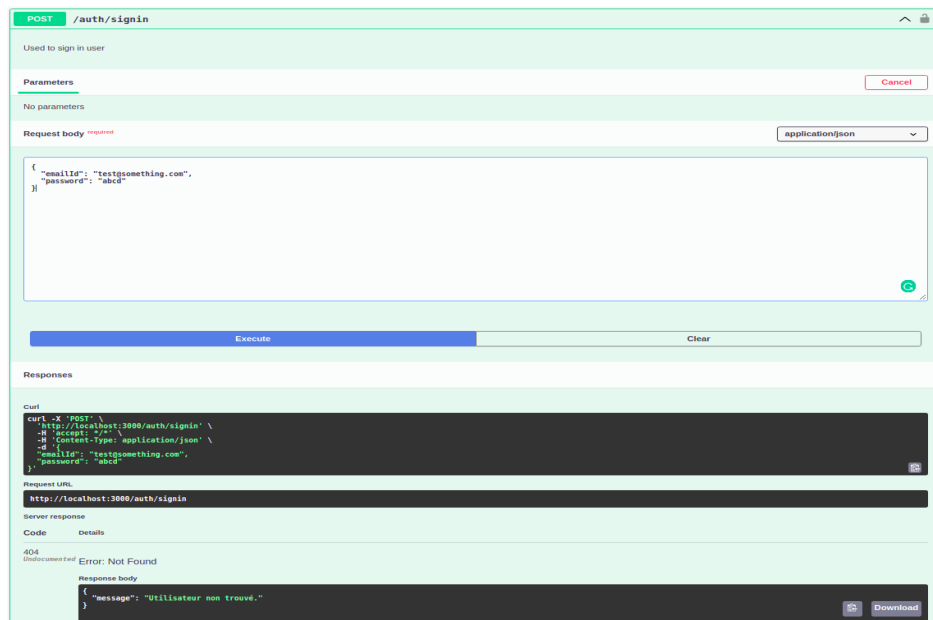
## Testez votre application avec Swagger

Création d'un nouvel utilisateur:

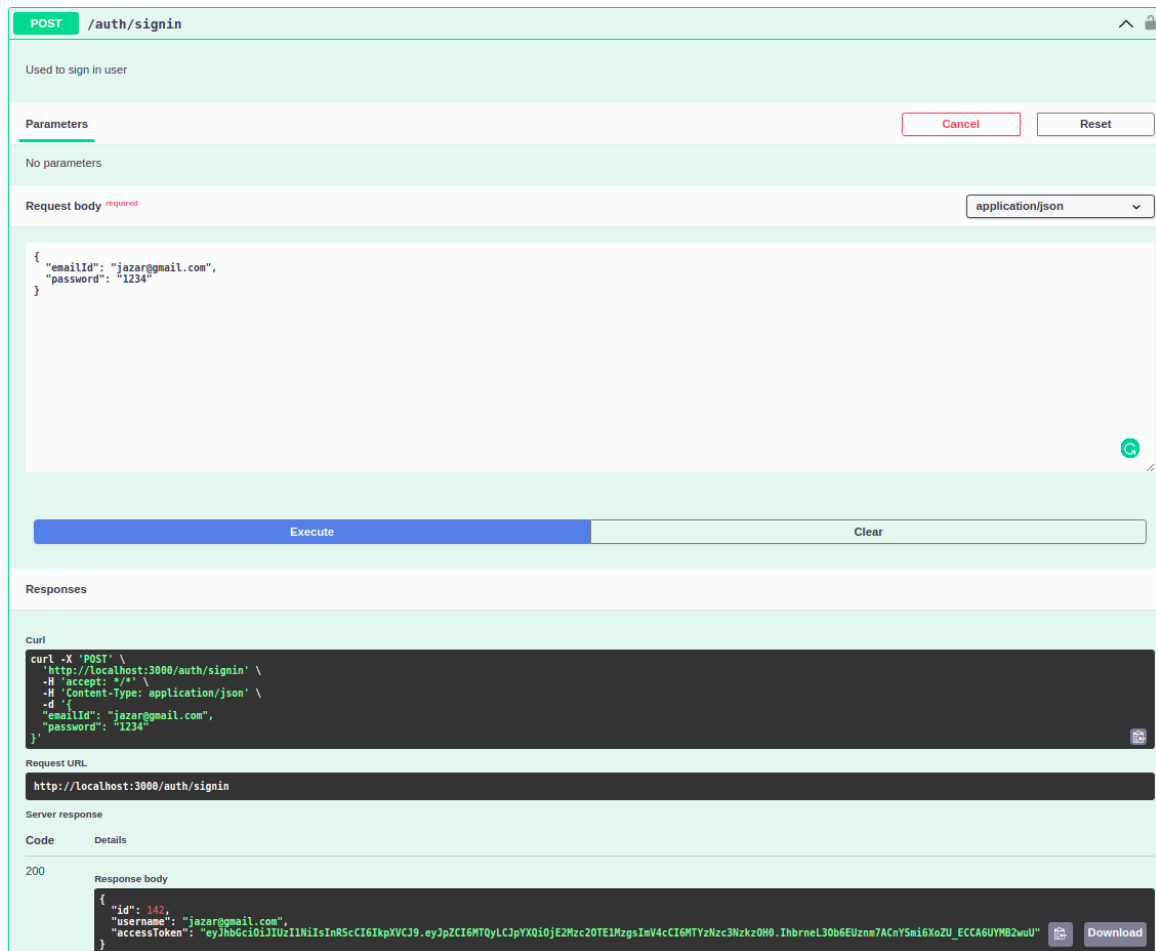
The screenshot displays the Swagger UI interface for a user registration endpoint. The top section shows the endpoint `POST /auth/signup` with a description "Used to signup user". Below this, the "Parameters" section is empty. The "Request body" section is set to `application/json` and contains a JSON object: `{ "firstName": "Joseph", "lastName": "Azar", "emailId": "azar@gmail.com", "password": "1234" }`. An "Execute" button is visible. The "Responses" section shows a `200` status code with a response body: `{ "message": "L'utilisateur a été enregistré avec succès!" }`. A "Curl" section at the bottom provides the command: `curl -X POST \n http://localhost:3000/auth/signup \n -H 'accept: */*\n' \n -H 'Content-Type: application/json' \n -d '{\n \"firstName\": \"Joseph\", \n \"lastName\": \"azar\", \n \"emailId\": \"azar@gmail.com\", \n \"password\": \"1234\" \n}'`. The "Request URL" is `http://localhost:3000/auth/signup`.



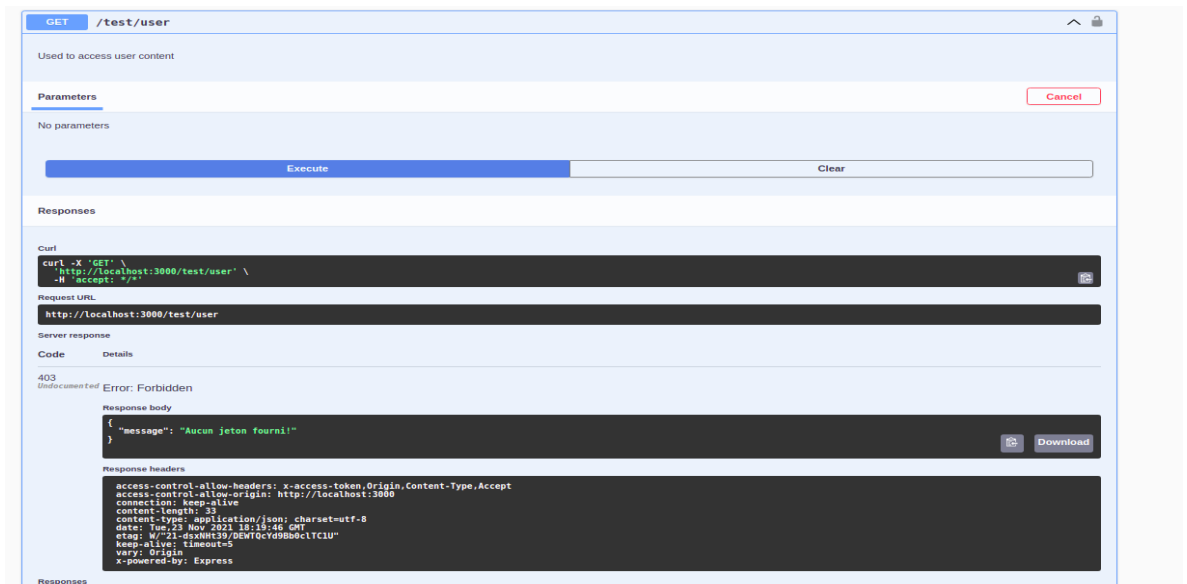
Lorsque vous essayez de vous connecter avec un e-mail ou un mot de passe erroné, vous devriez obtenir une erreur:



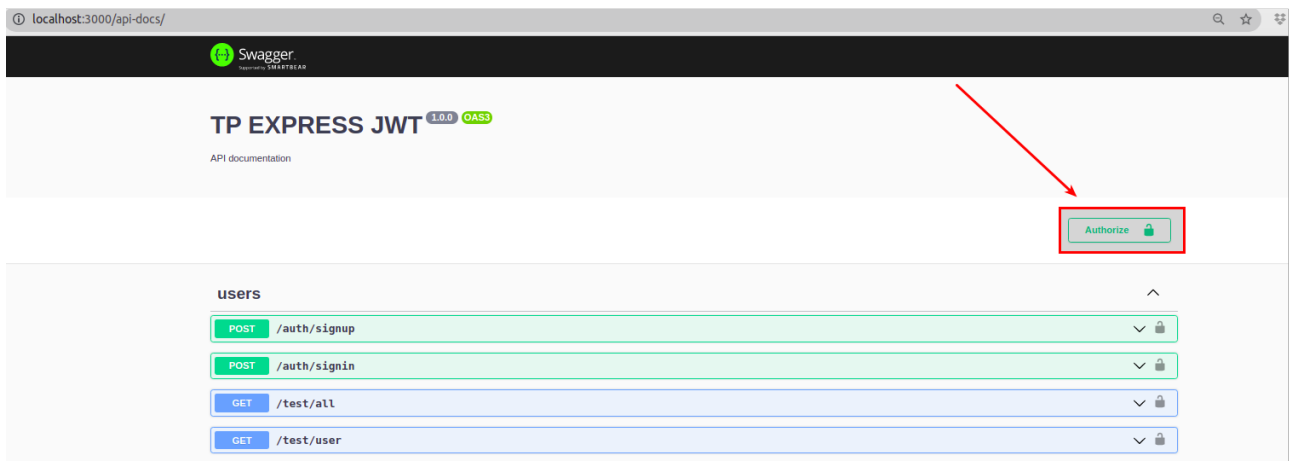
La première fois que vous vous connectez avec succès, vous obtiendrez le jeton:



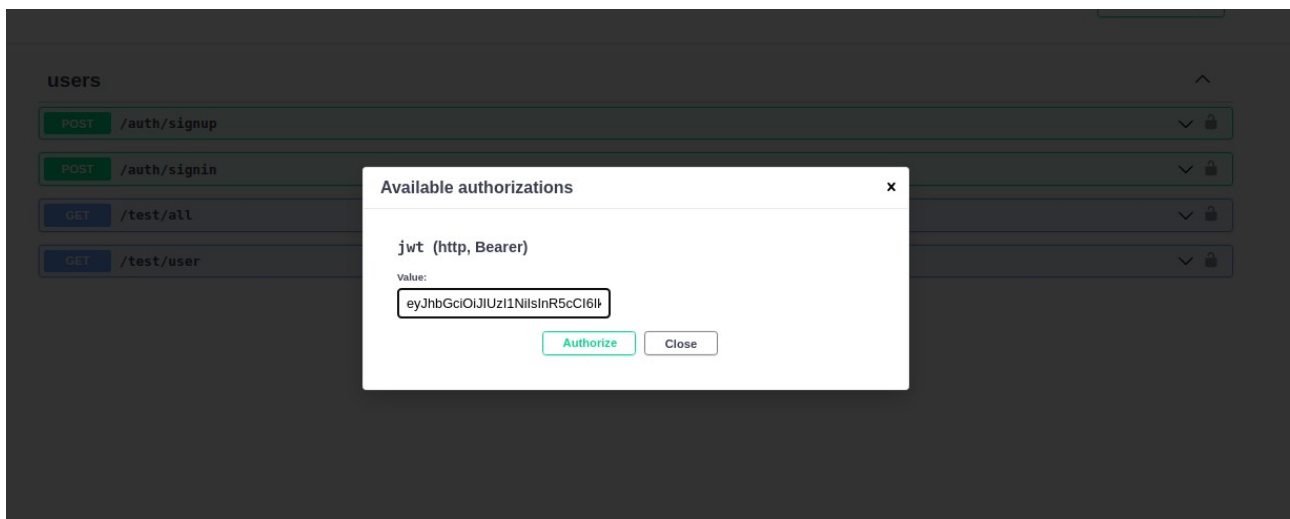
Si vous essayez d'accéder à une page privée sans fournir le jeton, vous devriez obtenir un message d'erreur:

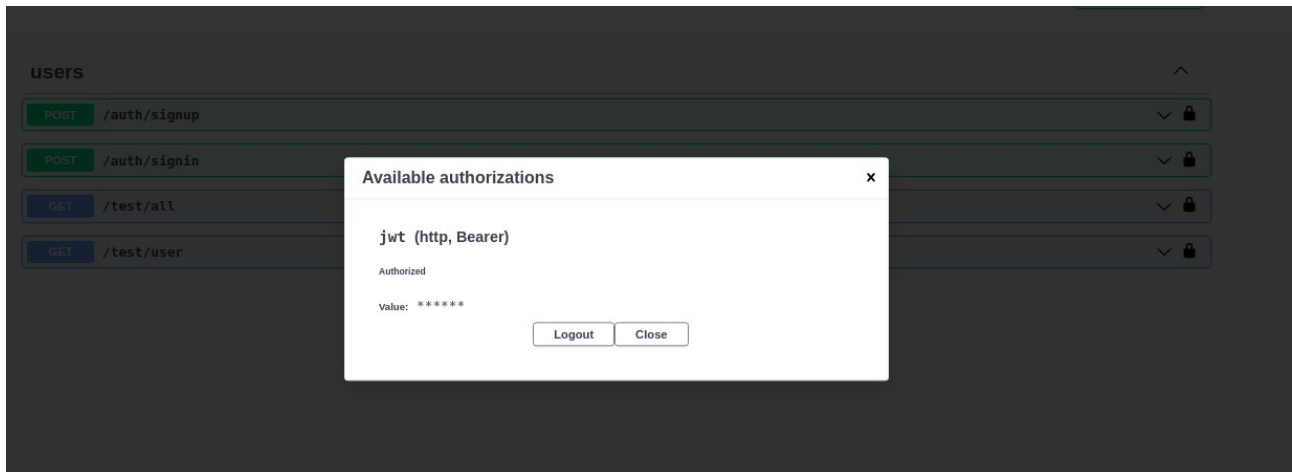


Afin d'ajouter le jeton à l'en-tête HTTP dans Swagger, cliquez sur le bouton “Authorize”:



Ajoutez ensuite le jeton et cliquez sur “Authorize”:





Le jeton est ajouté à l'attribut “**authorization**” dans l'en-tête de la requête HTTP. Vous pouvez désormais accéder au contenu privé:



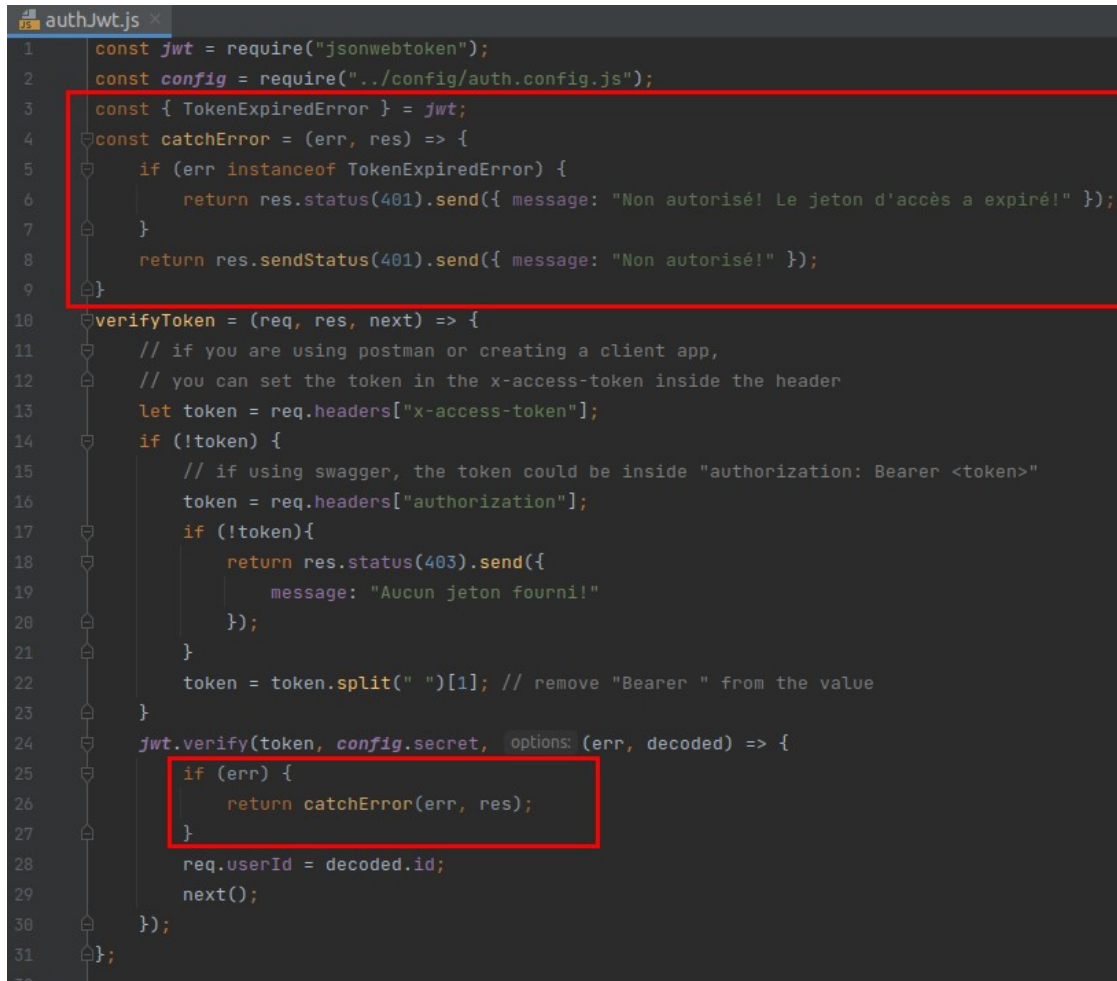
## Ajouter la fonctionnalité RefreshToken à l'application

Pour ne pas devoir se reconnecter sur une longue période de temps, tout en gardant la possibilité de ban ou de changer les droits d'un utilisateur sur cette période, il est important de mettre en place un mécanisme pour régénérer automatiquement le token. Le jeton d'actualisation (refresh token) a une valeur et un délai d'expiration différents du jeton d'accès. Régulièrement, nous configurons le délai d'expiration de Refresh Token plus long que celui d'Access Token.

Ouvrez **config/auth.config.js** et mettez-le à jour comme suit:

```
module.exports = {  
  secret: "my_secret", // vous pouvez modifier cette valeur  
  jwtExpiration: 60,    // 1 min  
  jwtRefreshExpiration: 120, // 2 min  
};
```

Nous devons maintenant mettre à jour le fichier **middlewares/authJwt.js** pour capturer l'erreur **TokenExpiredError** dans la fonction **verifyToken()**.



```
1  const jwt = require("jsonwebtoken");  
2  const config = require("../config/auth.config.js");  
3  const { TokenExpiredError } = jwt;  
4  const catchError = (err, res) => {  
5    if (err instanceof TokenExpiredError) {  
6      return res.status(401).send({ message: "Non autorisé! Le jeton d'accès a expiré!" });  
7    }  
8    return res.sendStatus(401).send({ message: "Non autorisé!" });  
9  }  
10 verifyToken = (req, res, next) => {  
11   // if you are using postman or creating a client app,  
12   // you can set the token in the x-access-token inside the header  
13   let token = req.headers["x-access-token"];  
14   if (!token) {  
15     // if using swagger, the token could be inside "authorization: Bearer <token>"  
16     token = req.headers["authorization"];  
17     if (!token){  
18       return res.status(403).send({  
19         message: "Aucun jeton fourni!"  
20       });  
21     }  
22     token = token.split(" ")[1]; // remove "Bearer " from the value  
23   }  
24   jwt.verify(token, config.secret, options: (err, decoded) => {  
25     if (err) {  
26       return catchError(err, res);  
27     }  
28     req.userId = decoded.id;  
29     next();  
30   });  
31 };
```

## Créer un modèle de Refresh Token

Ce modèle Sequelize a une relation un-à-un (one-to-one) avec le modèle utilisateur (user). Il contient le champ **expiryDate** dont la valeur est définie en ajoutant la valeur **config.jwtRefreshExpiration**.

Il existe 2 méthodes statiques :

- **createToken**: utilisez la bibliothèque **uuid** pour créer un jeton aléatoire et enregistrez un nouvel objet dans la base de données MySQL
- **verifyExpiration**: comparez la date d'expiration à la date actuelle pour vérifier l'expiration

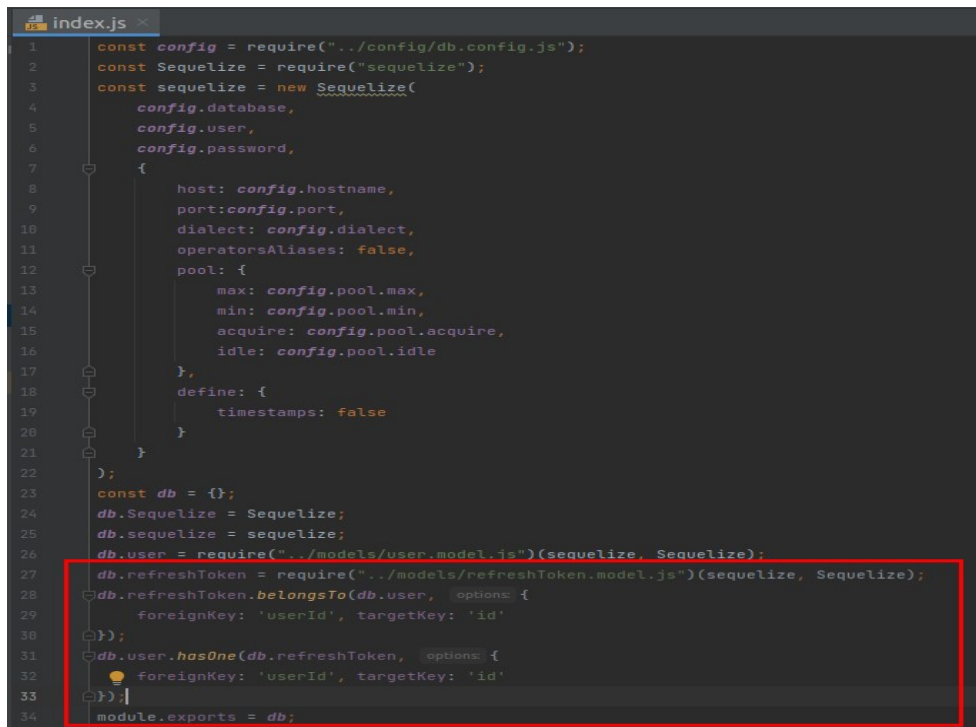
Créez le fichier **models/refreshToken.model.js** à l'aide de cette commande:

**\$ touch models/refreshToken.model.js**

```
const config = require("../config/auth.config");
const { v4: uuidv4 } = require("uuid");
module.exports = (sequelize, Sequelize) => {
  const RefreshToken = sequelize.define("refreshToken", {
    token: {
      type: Sequelize.STRING,
    },
    expiryDate: {
      type: Sequelize.DATE,
    },
  });
  RefreshToken.createToken = async function (user) {
    let expiredAt = new Date();
    expiredAt.setSeconds(expiredAt.getSeconds() + config.jwtRefreshExpiration);
    let _token = uuidv4();
    let refreshToken = await this.create({
      token: _token,
      userId: user.id,
      expiryDate: expiredAt.getTime(),
    });
    return refreshToken.token;
  };
  RefreshToken.verifyExpiration = (token) => {
    return token.expiryDate.getTime() < new Date().getTime();
  };
  return RefreshToken;
};
```

Nous devons mettre à jour le fichier **models/index.js** afin de ne pas oublier d'utiliser **belongsToMany** et **hasOne** pour configurer l'association avec le modèle User.

Exportez ensuite le modèle **RefreshToken** dans **models/index.js**:



```
1  const config = require("../config/db.config.js");
2  const Sequelize = require("sequelize");
3  const sequelize = new Sequelize(
4    config.database,
5    config.user,
6    config.password,
7    {
8      host: config.hostname,
9      port: config.port,
10     dialect: config.dialect,
11     operatorsAliases: false,
12     pool: {
13       max: config.pool.max,
14       min: config.pool.min,
15       acquire: config.pool.acquire,
16       idle: config.pool.idle
17     },
18     define: {
19       timestamps: false
20     }
21   });
22
23   const db = {};
24   db.Sequelize = Sequelize;
25   db.sequelize = sequelize;
26   db.user = require("../models/user.model.js")(sequelize, Sequelize);
27   db.refreshToken = require("../models/refreshToken.model.js")(sequelize, Sequelize);
28   db.refreshToken.belongsTo(db.user, {
29     foreignKey: 'userId', targetKey: 'id'
30   });
31   db.user.hasOne(db.refreshToken, {
32     foreignKey: 'userId', targetKey: 'id'
33   });
34   module.exports = db;
```

## API Node.js Express Rest pour le JWT Refresh Token

Nous devons mettre à jour le fichier **controllers/auth.controller.js** et ajouter ce qui suit:

- ajouter le refreshToken à la réponse de la méthode "signin"
- ajouter l'API POST pour créer un nouveau jeton d'accès à partir du jeton d'actualisation (refresh token) reçu

Mettez à jour la méthode de connexion comme suit:

```
auth.controller.js
24 exports.signin = (req, res) => {
25   console.log(req.body);
26   User.findOne({
27     where: {
28       emailId: req.body.emailId
29     }
30   })
31   .then(async (user) => {
32     if (!user) {
33       return res.status(404).send({ message: "Utilisateur non trouvé." });
34     }
35     let passwordIsValid = bcrypt.compareSync(
36       req.body.password,
37       user.password
38     );
39     if (!passwordIsValid) {
40       return res.status(401).send({
41         accessToken: null,
42         message: "Mot de passe incorrect!"
43       });
44     }
45     let token = jwt.sign( payload: { id: user.id }, config.secret, options: {
46       expiresIn: config.jwtExpiration
47     });
48     let refreshToken = await RefreshToken.createToken(user);
49     res.status(200).send({
50       id: user.id,
51       username: user.emailId,
52       accessToken: token,
53       refreshToken: refreshToken,
54     });
55   })
56   .catch(err => {
57     res.status(500).send({ message: err.message });
58   });
59 }
```

Ajoutez la méthode **refreshToken**:

```
exports.refreshToken = async (req, res) => {
  const { refreshToken: requestToken } = req.body;
  if (requestToken == null) {
    return res.status(403).json({ message: "Le jeton d'actualisation est requis!" });
  }
  try {
    let refreshToken = await RefreshToken.findOne({ where: { token: requestToken } });
    console.log(refreshToken)
    if (!refreshToken) {
      res.status(403).json({ message: "Le jeton d'actualisation n'est pas dans la base de données!" });
      return;
    }
    if (RefreshToken.verifyExpiration(refreshToken)) {
      RefreshToken.destroy({ where: { id: refreshToken.id } });
      res.status(403).json({
```

```

        message: "Le jeton d'actualisation a expiré. Veuillez faire une nouvelle demande de connexion",
      });
      return;
    }
    const user = await refreshToken.getUser();
    let newAccessToken = jwt.sign({ id: user.id }, config.secret, {
      expiresIn: config.jwtExpiration,
    });
    return res.status(200).json({
      accessToken: newAccessToken,
      refreshToken: refreshToken.token,
    });
  } catch (err) {
    return res.status(500).send({ message: err });
  }
};

```

```

exports.refreshToken = async (req, res) => {
  const { refreshToken: requestToken } = req.body;
  if (requestToken == null) {
    return res.status(403).json({ message: "Le jeton d'actualisation est requis!" });
  }
  try {
    let refreshToken = await RefreshToken.findOne({ where: { token: requestToken } });
    console.log(refreshToken);
    if (!refreshToken) {
      res.status(403).json({ message: "Le jeton d'actualisation n'est pas dans la base de données!" });
      return;
    }
    if (RefreshToken.verifyExpiration(refreshToken)) {
      RefreshToken.destroy({ where: { id: refreshToken.id } });
      res.status(403).json({
        message: "Le jeton d'actualisation a expiré. Veuillez faire une nouvelle demande de connexion (sign in)",
      });
      return;
    }
    const user = await refreshToken.getUser();
    let newAccessToken = jwt.sign({ payload: { id: user.id }, config.secret, options: {
      expiresIn: config.jwtExpiration,
    });
    return res.status(200).json({
      accessToken: newAccessToken,
      refreshToken: refreshToken.token,
    });
  } catch (err) {
    return res.status(500).send({ message: err });
  }
};

```

vérifier si un refreshToken est envoyé dans la requête HTTP  
 vérifier si le refreshToken envoyé est présent dans la base de données  
 Si le refreshToken a expiré, demandez à l'utilisateur de se reconnecter  
 Envoyer un nouvel accessToken à l'utilisateur

## Définir la route pour l'API JWT Refresh Token

Dans **routes/auth.routes.js**, ajoutez cette ligne de code:

```
app.post("/auth/refreshToken", controller.refreshToken);
```

Ajoutez la définition swagger pour la documentation de cette route et testez votre application à l'aide de swagger. Lorsque vous vous connectez pour la première fois, vous obtiendrez un jeton d'actualisation et une nouvelle ligne sera ajoutée au tableau MySQL **refreshTokens**:

```

mysql> describe refreshTokens;
+-----+-----+-----+-----+-----+-----+
| Field | Type | Null | Key | Default | Extra |
+-----+-----+-----+-----+-----+-----+
| id    | int(11) | NO | PRI | NULL | auto_increment |
| token | varchar(255) | YES | | NULL | |
| expiryDate | datetime | YES | | NULL | |
| userId | int(11) | YES | MUL | NULL | |
+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)

mysql> select * from refreshTokens;
+-----+-----+-----+-----+
| id | token | expiryDate | userId |
+-----+-----+-----+-----+
| 1 | bb38e908-f0ad-4f8b-989c-7aa37da926a2 | 2021-11-29 00:16:36 | 142 |
+-----+-----+-----+-----+

```

Lorsque votre token est expiré, vous recevrez ce message:

GET /test/user
Used to access user content

Parameters

No parameters

ExecuteClear

Response

```
curl -X 'GET' \
  http://localhost:3000/test/user \
  -H 'accept: */*' \
  -H 'Authorization: Bearer eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.ejYpZCtGMHT0yLC3pYYk0lOjE2Mzg5NDQ4NzYsIiwiaWQiOiJCMTEMTyZDE0NDkxNm0uBAAJPAJCX4TKLYx4L0U4RM-rCe4vLG0rXnA6IG2-g'
```

Request URL  
http://localhost:3000/test/user

Server response

Code	Details
401	<b>Error: Unauthorized</b>

Response body

```
{
  "message": "Non autorisé! Le jeton d'accès a expiré!"
}
```

Download

Response headers

```
access-control-allow-headers: x-access-token,Origin,Content-Type,Accept
access-control-allow-origin: http://localhost:3000
connection: keep-alive
content-length: 57
content-type: application/json; charset=utf-8
date: Mon, 29 Nov 2021 00:15:40 GMT
etag: W/"39-3KwVamT0bcnkJyvocMseAmcq"
keep-alive: timeout=5
vary: Origin
x-powered-by: Express
```

Response	Description	Links
200	Resource returned successfully	No links
400	Bad request	No links
500	Internal server error	No links

Ensuite, vous pouvez envoyer votre jeton d'actualisation à **/auth/refreshToken** route pour obtenir un nouveau jeton d'accès.

POST

/auth/refreshToken

Used to refresh token

Parameters

Cancel

Reset

No parameters

Request body required

application/json

```
{
  "refreshToken": "bb38e988-f0ad-4f8b-989c-7aa37da926a2"
}
```

Execute

Clear

Responses

Curl

```
curl -X 'POST' \
  'http://localhost:3000/auth/refreshToken' \
  -H 'accept: */*' \
  -H 'Authorization: Bearer ey2hb6ci0IjIuZlInIiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTQyLjYpYXQ1b2ZMc2g4NDQ4NzYsInV4cCI6MTYzODc0E0NTA1NW0uB4AYtPAjCXATXklyXALDU4RmRrCmE2-g' \
  -H 'Content-Type: application/json' \
  -d '{
    "refreshToken": "bb38e988-f0ad-4f8b-989c-7aa37da926a2"
  }'
```

Request URL

http://localhost:3000/auth/refreshToken

Server response

Code

Details

200

Response body

```
{
  "accessToken": "eyJ2hb6ciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6MTQyLjYpYXQ1b2ZMc2g4NDQ5OTQsInV4cCI6MTYzODc0E0NTA1NW0uB4AYtPAjCXATXklyXALDU4RmRrCmE2-g:X85un7aRkEvc5uMcw",
  "refreshToken": "bb38e988-f0ad-4f8b-989c-7aa37da926a2"
}
```

Download

Response headers

```
access-control-allow-headers: x-access-token,Origin,Content-Type,Accept
access-control-allow-origin: http://localhost:3000
connection: keep-alive
content-length: 212
content-type: application/json; charset=utf-8
date: Mon, 29 Nov 2021 08:18:34 GMT
etag: W/"44-gjALPfm0CRK7nRVKH7c"
keep-alive: timeout=5
vary: Origin
x-powered-by: Express
```