Dossier P6\_new :

Terminal vsc :

Node.js v14.15.5

Backend : npm init

Dans package.json on a :

{

"name": "backend",

"version": "1.0.0",

"description": "P6\_Openclassrooms",

"main": "server.js",

"scripts": {

"test": "echo \"Error: no test specified\" && exit 1"

},

Nodemon : 2.0.7

nodemon server

frontend

npm install

npm start

Angular CLI : 12.2.2 déjà installé (pour la version de Angular CLI taper dans la console ng version)

Dans le dossier frontend : npm install --save node-sass@4.14

node-sass v4.14.1

.gitignore dans backend : node\_modules

\*) Créer server.js et tester le serveur backend :

const http = require('http');

const server = http.createServer((req, res) => {

res.end('Voilà la réponse du serveur !');

});

server.listen(process.env.PORT || 3000);

ok :

\*) installer 2 en 1 : express et dotenv :

Dans backend

npm i express dotenv

+ dotenv@10.0.0

+ express@4.17.1

Installation de dotenv et express : dans backend npm i express dotenv

Remplacez les scripts existants dans package.json par le script suivant afin que nous puissions exécuter notre application :

"start": "node server.js"

{

"main": " server.js",

"scripts": {

"start": "node server.js"

},

\*) créer app.js pour l'application express et modification du fichier server.js pour tenir compte de cette appli Express:

Installation

Here are the dependancies you need to install:

* NodeJS 12.14 or 14.0.
* Angular CLI 7.0.2.
* node-sass : make sure to use the corresponding version to NodeJS. For Nodcd..
* e 14.0 for instance, you need node-sass in version 4.14+.

On Windows, these installations require to use PowerShell in administrator mode.

Then, clone this repo and run npm install.

Usage

Run npm start. This should both run the local server and launch your browser.

If your browser fails to launch, or shows a 404 error, navigate your browser to <http://localhost:8080>.

The app should reload automatically when you make a change to a file.

Use Ctrl+C in the terminal to stop the local server.

Séances mentor :

01/09 :

Q1 : variable d'environnemnt pour les données sensibles bdd etc ..

MySQL est un exemple de base de données relationnelle - vous utiliseriez un ORM pour traduire entre vos objets dans le code et la représentation relationnelle des données.

Des exemples d'ORM sont [nHibernate](http://nhibernate.info/) , [Entity Framework](http://msdn.microsoft.com/en-us/data/ef.aspx) , [Dapper](http://code.google.com/p/dapper-dot-net/) et plus ...

MongoDB est un exemple de base de données de documents - vous utiliseriez un ODM pour traduire entre vos objets dans le code et la représentation documentaire des données (si nécessaire).

[Mandango](https://github.com/mandango/mandango) est un exemple d'ODM pour MongoDB.

Un ORM mappe entre un modèle objet et une base de données relationnelle. Un ODM mappe entre un modèle objet et une base de données de documents. MySQL n'est pas un ORM, c'est une base de données relationnelle, plus précisément une base de données SQL. MongoDB n'est pas un ODM, c'est une base de données de documents.

Essentiellement, un ORM utilise un pilote de base de données SQL comme ODBC, JDBC ou OLEDB pour traduire la notation d'objet en notation relationnelle et un ODM utilise une API JSON ou JSONB pour traduire la notation d'objet en notation de document.

Il existe différents types d'implémentations sous le capot.

PS: JSONB est une notation de document texte JSON stockée dans un format binaire tel qu'utilisé par MongoDB.

01/09/2021

1. ajout du middelware pour CORS
2. mise en place de la bdd mongoDB Atlas et de mongoose : new project : OC-P6\_piiquante (cluster name : OC-P6)
3. installation de mongoose dans backend :

À partir de la version 4.16 d'Express, bodyparser est inclus et vous n'avez pas besoin de l'installer.

Utilisez (express.json()) pour analyser le corps de la requête.

**A faire : finie**

**Étape 2 : Construire le parcours utilisateur (30 % d’avance)**

● Créez les éléments suivants :

X Modèle d'utilisateur ;

X Parcours utilisateur ; (signup, login, )

X Contrôleur d'utilisateur.

● L'utilisateur est en mesure d'effectuer les opérations suivantes :

○ Créer un compte ;

○ Se connecter et disposer d'un token valide.

● Consultez l'onglet réseau de Devtools pour plus d'informations.

⚠️**Les problèmes à connaître :**

● *Le mot de passe n'est pas haché.* Veillez à hacher le mot de passe.

● *Un utilisateur peut s'inscrire plusieurs fois avec la même adresse*

*électronique.* Assurez-vous que le code vérifie qu’une adresse

électronique est unique.

**\*/**

**Utilisateur**

**● email** : *String* — adresse e-mail de l'utilisateur **[unique]**

**● password** : *String* — mot de passe de l'utilisateur haché

//définir la fonction json de express comme middleware global pour l'application au lieu de bodyParser.json()

app.use(express.json());

\*/ optimisation de l'architecture :

1- routeur

Création du routeur pour les users : Dossier : routes/user.js : const router = express.Router();

Et dans app.js déplacer toutes la logique du routing et la remplacer par :

app.use('/api/auth', userRoutes)

2- contrôleur

Créez un dossier controllers dans votre dossier backend et créez un autre fichier user.js . Celui-ci sera notre contrôleur user

\*\*\*\*\*\*

02/09/2021

\*/

Pour s'assurer que deux utilisateurs ne peuvent pas utiliser la même adresse e-mail, nous utiliserons le mot clé unique pour l'attribut email du schéma d'utilisateur userSchema. Les erreurs générées par défaut par MongoDB pouvant être difficiles à résoudre, nous installerons un package de validation pour pré-valider les informations avant de les enregistrer :

Dans backend : npm install --save mongoose-unique-validator

\*/ création des routes et des contrôleurs de user :

dans backend/controleurs/user.js => exports.signup et exports.login

dans backend/routes.user.js =>

on aura :

dans controleurs/user.js :

exports.signup = (req, res, next) => { …

et

exports.login = (req, res, next) => {

dans routes/user.js :

const router = express.Router();

const userCtrl = require('../controllers/user');

et

router.post('/signup', userCtrl.signup);

router.post('/login', userCtrl.login);

et

module.exports = router;

dans app.js :

const userRoutes = require('./routes/user');

et

app.use('/api/auth', userRoutes)

et

module.exports = app;

\*/ Il nous faudra le package de chiffrement bcrypt pour notre fonction signup . Installer et importer :

Dans backend : npm install --save bcrypt

\*/ implémenter la fonction signup (ne pas oublier d'importer le modèle User !!!)

/\*

contrôleur pour l'enregistrement d'un nouvel utilisateur :

1- crypter le mdp, 2- créer nouvel user, 3- l'enregistrer dans la BDD

 \*/

exports.signup = (req, res, next) => {

    bcrypt.hash(req.body.password, 10) // c'est une Promise dont la résolution est le hash généré

        .then(hash => {

            const user = new User({

                email: req.body.email,

                password: hash

            });

            user.save()

                .then(() => res.status(201).json({

                    message: 'Utilisateur créé !'

                }))

                .catch(error => res.status(400).json({

                    error

                }));

        })

        .catch(error => res.status(500).json({

            error

        }));

};

* on commence par appeler la fonction hash() de bcrypt sur notre mot de passe et lui demandons de « saler » le mot de passe 10 fois.
* C'est une fonction asynchrone qui renvoie une Promise dans laquelle nous recevons le hash généré ;
* La résolution de la promesse permet, dans le bloc then, de créer un nouvel utilisateur et de l'enregistrer dans la BDD, en renvoyant une réponse de réussite en cas de succès, et des erreurs avec le code d'erreur en cas d'échec ;

\*/ implementer la fonction login, qui permet de vérifier si un utilisateur qui tente de se connecter dispose d'identifiants valides.

Les *tokens* d'authentification permettent aux utilisateurs de ne se connecter qu'une seule fois à leur compte. Au moment de se connecter, ils recevront leur *token* et le renverront automatiquement à chaque requête par la suite. Ceci permettra au back-end de vérifier que la requête est authentifiée.

Pour créer et vérifier les *tokens* d'authentification, il nous faudra un nouveau package :

Dans backend : npm install --save jsonwebtoken

Nous l'importerons ensuite dans notre contrôleur utilisateur :

const jwt = require('jsonwebtoken');

exports.login = (req, res, next) => {

    User.findOne({

            email: req.body.email //query filter de mongoose (trouver un user dans la BDD dont l'email correspond à ce qui été envoyé dans le body de la requête)

        })

        .then(user => { //la résolution de la promesse envoie un user

            if (!user) {

                return res.status(401).json({

                    error: 'Utilisateur non trouvé !'

                });

            } // si on a trouvé un user avec le bon email alors on compare les hash des mdp :

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

                .then(valid => { // la résolution de la promesse envoie un booléen

                    if (!valid) {

                        return res.status(401).json({ //401 : unauthorized

                            error: 'Mot de passe incorrect !'

                        });

                    } //si la comparaison est valide, on répond par l'envoi du token et du userId qui va avec

                    res.status(200).json({

                        userId: user.\_id,

                        token: jwt.sign({

                                userId: user.\_id //encoder le userId dans le token pour éviter qu'un user modifie des données d'un autre user

                            },

                            'RANDOM\_TOKEN\_SECRET', {

                                expiresIn: '24h'

                            }

                        )

                    });

                })

                .catch(error => res.status(500).json({

                    error

                }));

        })

        .catch(error => res.status(500).json({

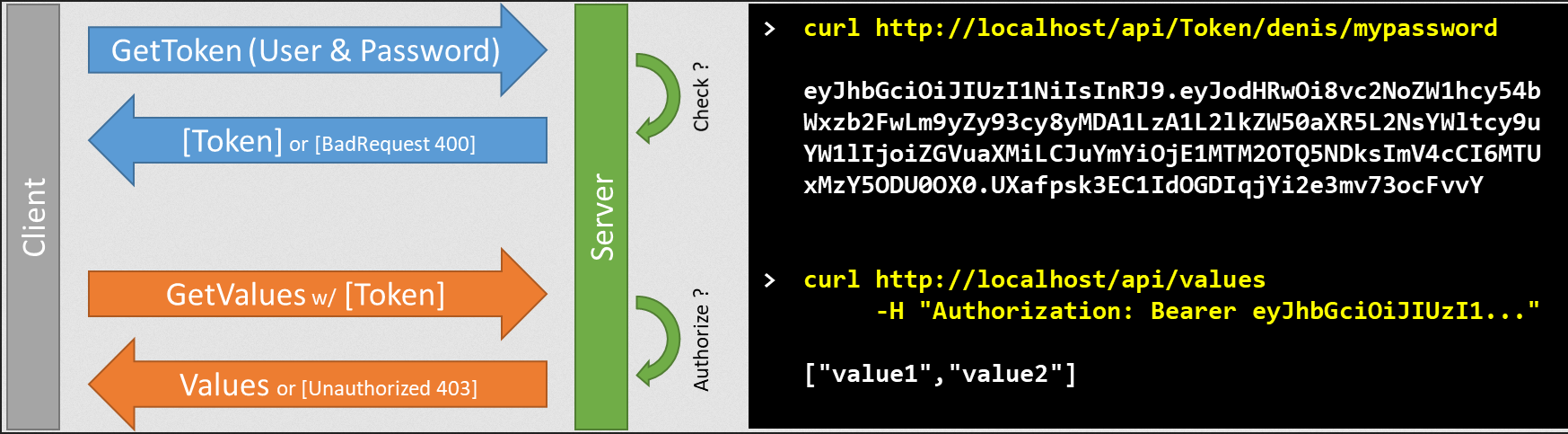
            error

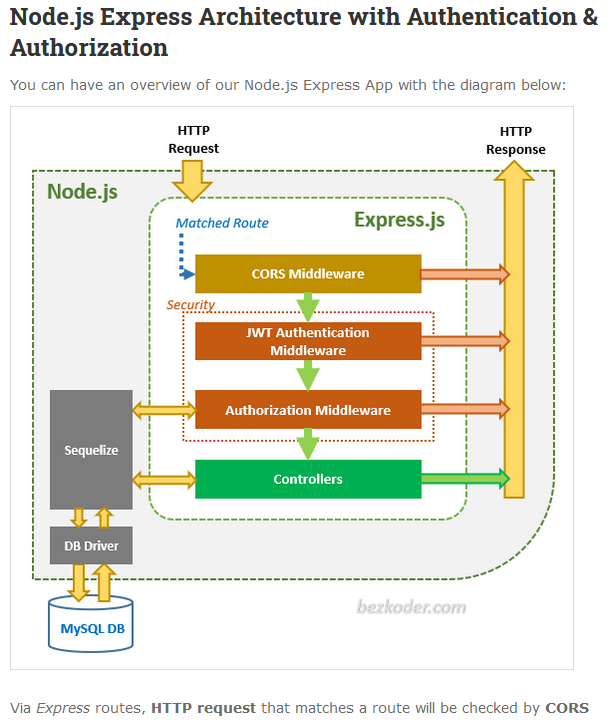
        }));

};

* nous utilisons notre modèle Mongoose pour vérifier que l'e-mail entré par l'utilisateur correspond à un utilisateur existant de la base de données :
  + dans le cas contraire, nous renvoyons une erreur 401 Unauthorized ,
  + si l'e-mail correspond à un utilisateur existant, nous continuons ;
* nous utilisons la fonction compare de bcrypt pour comparer le mot de passe entré par l'utilisateur avec le hash enregistré dans la base de données :
  + s'ils ne correspondent pas, nous renvoyons une erreur 401 Unauthorized et un message « Mot de passe incorrect ! » ;
  + s'ils correspondent, les informations d'identification de notre utilisateur sont valides. Dans ce cas, nous renvoyons une réponse 200 contenant l'ID utilisateur et un *token*. Ce *token*est une chaîne générique pour l'instant, mais nous allons le modifier et le crypter dans le prochain chapitre.
* nous utilisons la fonction sign dejsonwebtoken pour encoder un nouveau *token* ;
* ce *token* contient l'ID de l'utilisateur en tant que *payload* (les données encodées dans le *token*) ;
* nous utilisons une chaîne secrète de développement temporaire RANDOM\_SECRET\_KEY pour encoder notre *token* (à remplacer par une chaîne aléatoire beaucoup plus longue pour la production) ; (12 caractères => 70 bits => 37 ans en Force brute)
* nous définissons la durée de validité du *token* à 24 heures. L'utilisateur devra donc se reconnecter au bout de 24 heures ;
* nous renvoyons le *token* au front-end avec notre réponse.

<https://www.npmjs.com/package/jsonwebtoken>





Via Express routes, **HTTP request** that matches a route will be checked by **CORS Middleware** before coming to **Security** layer.

**Security** layer includes:

* JWT Authentication Middleware: verify SignUp, verify token
* Authorization Middleware: check User’s roles with record in database

If these middlewares throw any error, a message will be sent as HTTP response.

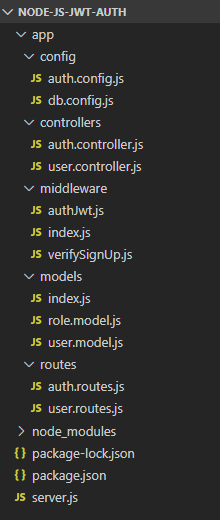
**Controllers** interact with MySQL Database via Sequelize and send **HTTP response** (token, user information, data based on roles…) to client.

## Technology

* Express 4.17.1
* bcryptjs 2.4.3
* jsonwebtoken 8.5.1
* Sequelize 5.21.3
* MySQL

## Project Structure

This is directory structure for our Node.js Express application:



– **config**

* configure MySQL database & Sequelize
* configure Auth Key

– **routes**

* auth.routes.js: POST signup & signin
* user.routes.js: GET public & protected resources

– **middlewares**

* verifySignUp.js: check duplicate Username or Email
* authJwt.js: verify Token, check User roles in database

– **controllers**

* auth.controller.js: handle signup & signin actions
* user.controller.js: return public & protected content

– **models** for Sequelize Models

* user.model.js
* role.model.js

– server.js: import and initialize neccesary modules and routes, listen for connections.

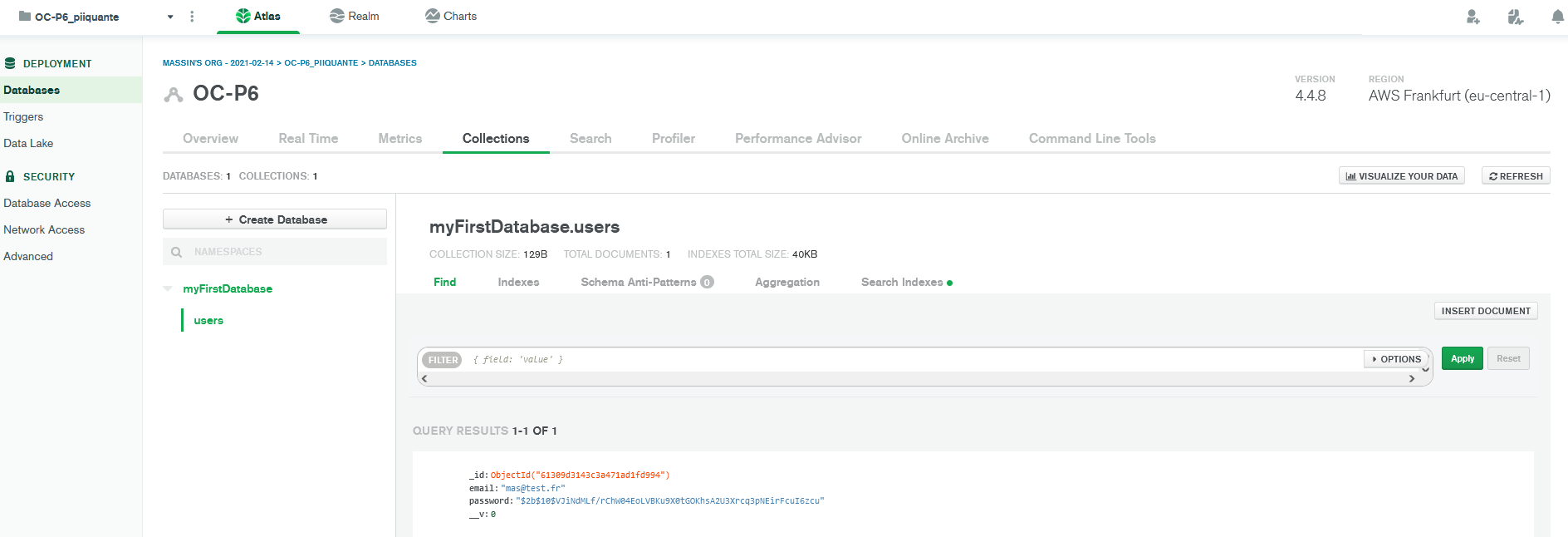
*JSON Web Token (JWT) est un standard ouvert (*[*RFC 7519*](https://tools.ietf.org/html/rfc7519)*) pour échanger de l’information de manière sécurisée via un jeton signé. Par exemple un serveur pourrait émettre un jeton possédant l’affirmation “utilisateur identifié en tant qu’administrateur” et le fournir au client. Le client pourrait alors vérifier le jeton pour prouver que l’utilisateur est identifié en tant qu’administrateur (Wikipedia).*

Personnellement, c’est une solution que je trouve élégante car

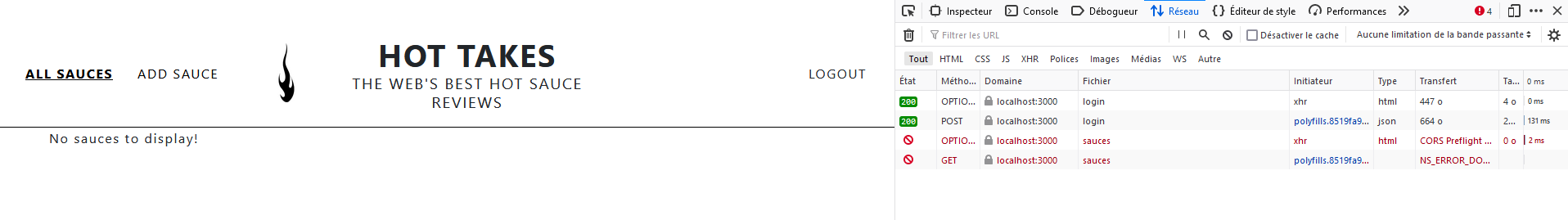
* Elle est simple à mettre en place : **quelques lignes de configuration** et l’utilisation de l’attribut [**Authorize**],
* Elle me permet de gérer facilement **mes propres règles de sécurité** lors de la vérification du nom d’utilisateur et du mot de passe,
* Et elle est surtout très facile à tester dans les applications clientes : il suffit d’ajouter un en-tête HTTP "**Authorization: Bearer [Token]**".

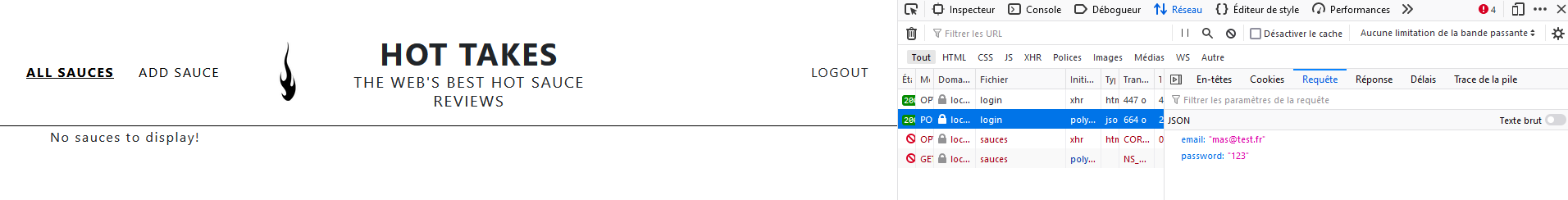
**jsonwebtoken** functions such as verify() or sign() use algorithm that needs a secret key (as String) to encode and decode token.

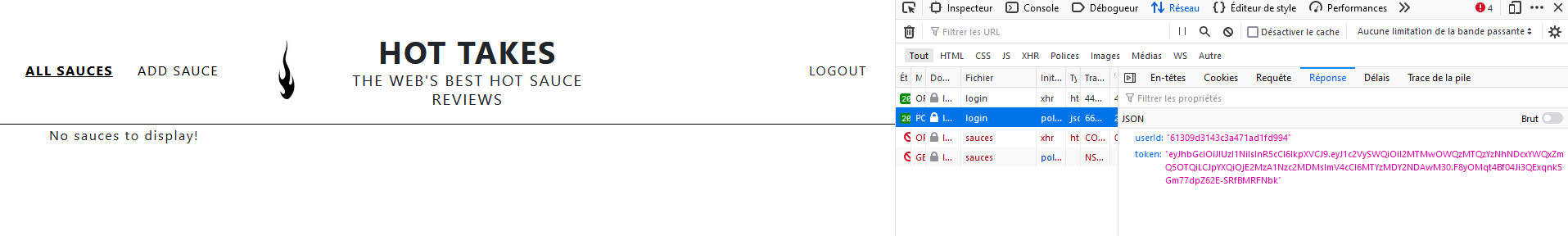
Test : signup d'un nouvel utilisateur : dans mongoDB Atlas le nouvel utilisateur est bien enregistré : (email : [mas@test.fr](mailto:mas2@test.fr), pwd : 123)



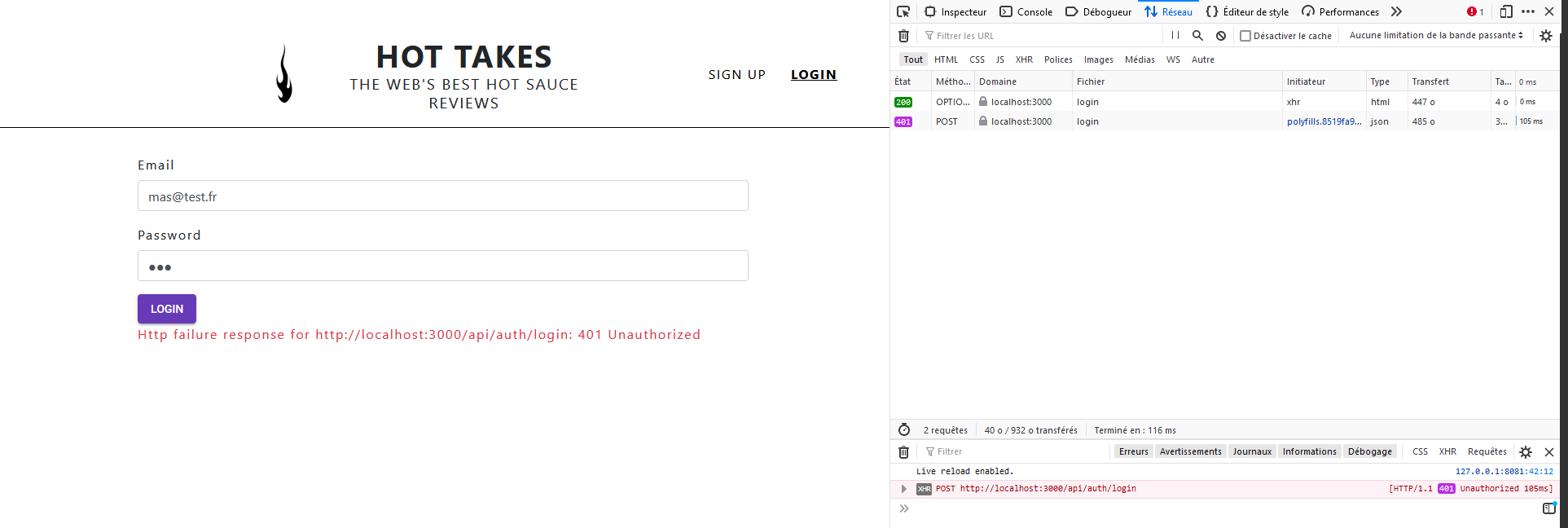
Si on essaye de se logger avec ces identifaints : ca marche :



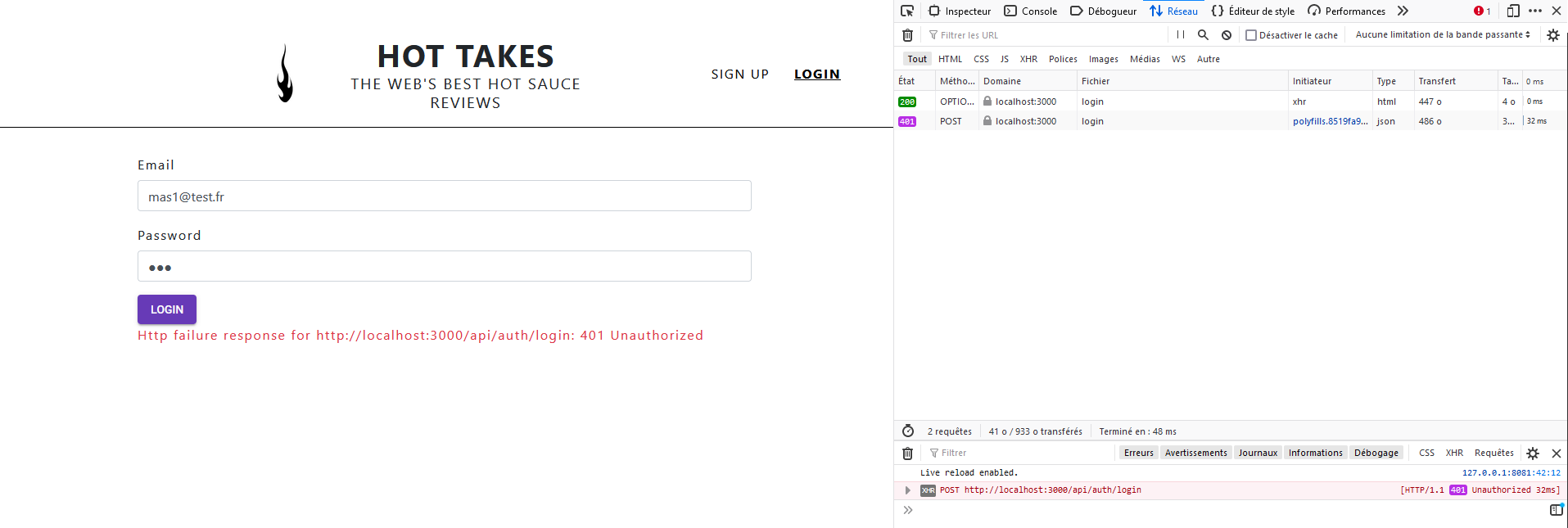




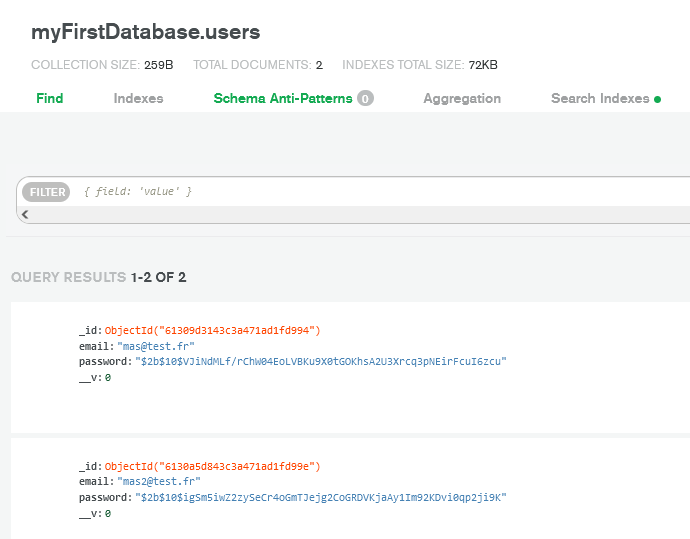
Si on essaye de logger avec le bon email mais pas le bon mdp : echec :



Si on essaye de logger avec un faux email mais le bon mdp : echec :



\*- enregistrement d'un 2ème utilisateur (email : [mas2@test.fr](mailto:mas2@test.fr), pwd : 456)



\*/ Commit et push : ok

////// 03/09/2021

**A faire : finie**

**Étape 3 :** **Démarrer le middleware (40 % d’avance)**

X +h : sécuriser toutes les routes avec le middleware d'authentification

X Ajout de multer pour les images.

X Ajout d’authorize pour la validation des tokens.

○ Authorize doit être ajoutée avant de commencer à construire le parcours pour les sauces car l'authentification est nécessaire pour qu'un utilisateur puisse effectuer une action sur le parcours des sauces.

⚠️**Les problèmes à connaître :**

● *Les images importées sont manquantes.*

● *Multer n'est pas correctement configuré.*

● *Le chemin statique n'a pas été ajouté à l'application pour fournir les*

*images.* Assurez-vous d'ajouter le chemin statique à l'application.

📃**Ressources**

● Configuration de multer (en anglais) .

● Configuration de multer (en français)

multer, un package qui nous permet de gérer les fichiers entrants dans les requêtes HTTP. Commençons par installer multer et par créer un élément de middleware pour gérer ces fichiers entrants.

1/ Dans backend : npm install --save multer 🡺 multer@1.4.3

2/ crée dossier backend/images

3/ crée un middleware : middlewares/multer-config.js

4/ créer un middleware d'autorisation : middlewares/auth.js

Pour protéger les routes sensibles de notre API : vérifie le token reçu depuis le frontend (vérifie l'authenticité du token mais aussi que si la requête contient un userId, ce userId correspond bien userId encodé dans le token) et permettre uniquement aux requêtes authentifiée de passer.

Dans backend crée le fichier middlewares/authorize.js

Ce middleware va être appliqué avant les contrôleurs des routes (sauces) pour les sécuriser : d'où si tous les vérifications de ce middleware sont ok on passe la requête avec next() au prochain middleware.

Dans routes/sauces importer ce middleware, et le placer devant le contrôleur de chaque route qu'on veut protéger.

\*/ Commit et push : ok

04/09/2021

\*/ ABONDON DU TUTO POUR LES ROLES (a faire dans un projet à part) :

<https://www.bezkoder.com/node-js-mongodb-auth-jwt/#Create_Middleware_functions>

retour au dernier commit en utilisant git clean et git reset :

P6\_new> git clean

P6\_new> git clean -xf

P6\_new> git reset --hard HEAD

**A faire :**

**Étape 4 :** **Construire la route Sauce de l’API (70 % d’avance)**

● Créez les éléments suivants :

X Le Modèle Sauce ;

X La Route Sauce ;

○ Le Contrôleur Sauce.

● Autorisez toutes les fonctions en utilisant middleware Authorize.

● L'utilisateur est en mesure d'effectuer les opérations suivantes :

○ Ajouter une nouvelle sauce ;

○ Supprimer une sauce ;

○ Voir toutes les sauces.

● Consultez l'onglet réseau de Devtools pour plus d'informations.

⚠️**Les problèmes à connaître :**

● *Erreur 401 (l'utilisateur n'est pas autorisé).*

● *Multer ne sauvegarde pas les images.*

● *Les images ne sont pas affichées sur le frontend.*

📃**Ressources**

● Ajout ou suppression d'un fichier de gitignore.

● Les méthodes des tableaux expliquées.

\*/ créer le modèle sauce : models/sauce.js

\*/ ajout du middleware auth aux routes des sauces pour les protéger

\*/ ajout du controleur createSauce, pour l'image, il faut ajouter dans app.js

const path = require('path'); pour faciliter la recherche dans le système des fichiers

et

app.use('/images', express.static(path.join(\_\_dirname, 'images')));

donc pour toutes requêt envoyé à /images on sert le dossier static 🡺on a indiqué à l'appli. Comment répondre aux reqêtes de ctte route get

\*/

GET /api/sauces - Array of sauces

Renvoie un tableau de toutes les sauces de la base de données.

GET /api/sauces/:id - Single sauce

Renvoie la sauce avec l’\_id fourni.

POST /api/sauces { sauce: String,image: File } { message: String }

Capture et enregistre l'image, analyse la sauce transformée en chaîne de caractères et l'enregistre dans la base de données en définissant correctement son imageUrl. Initialise les likes et dislikes de la sauce à 0 et les usersLiked et usersDisliked avec des tableaux vides. Remarquez que le corps de la demande initiale est vide ; lorsque multer est ajouté, il renvoie une chaîne pour le corps de la demande en fonction des données soumises avec le fichier.

PUT /api/sauces/:id EITHER Sauce as JSON OR { sauce:String,image: File } { message: String }

Met à jour la sauce avec l'\_id fourni. Si une image est téléchargée, elle est capturée et l’imageUrl de la sauce est mise à jour. Si aucun fichier n'est fourni, les informations sur la sauce se trouvent directement dans le corps de la requête (req.body.name, req.body.heat, etc.). Si un fichier est fourni, la sauce transformée en chaîne de caractères se trouve dans req.body.sauce. Notez que le corps de la demande initiale est vide ; lorsque multer est ajouté, il renvoie une chaîne du corps de la demande basée sur les données soumises avec le fichier.

DELETE /api/sauces/:id - { message: String }

Supprime la sauce avec l'\_id fourni.

POST /api/sauces/:id/like { userId: String,like: Number }{ message: String }

Définit le statut « Like » pour l' userId fourni. Si like = 1, l'utilisateur aime (= like) la sauce. Si like = 0, l'utilisateur annule son like ou son dislike. Si like = -1, l'utilisateur n'aime pas (=dislike) la sauce.

L'ID de l'utilisateur doit être ajouté ou retiré du tableau approprié. Cela permet de garder une trace de leurs préférences et les empêche de liker ou de ne pas disliker la même sauce plusieurs fois : un utilisateur ne peut avoir qu'une seule valeur pour chaque sauce. Le nombre total de « Like » et de « Dislike » est mis à jour à chaque nouvelle notation.

Étape 5 : Terminer la route Sauce de l’API

**100 %** d’avance

🎯**Partie du livrable :** API complété

📌**Recommandations :**

● Exécutez l'application en tant qu'utilisateur pour vérifier que toutes

les fonctions ont été correctement mises en oeuvre, testez :

○ Les deux types de demandes :

■ Avec un fichier présent ;

■ Sans fichier.

○ Les trois scénarios de la fonction « like » (1, 0, -1) ;

■ L’utilisateur peut liker ou ne pas aimer une sauce (ou

aucun des deux)

○ Seul le propriétaire de la sauce peut modifier ou supprimer

une sauce existante.

⚠️**Les problèmes à connaître :**

● *Erreur 401 (l'utilisateur n'est pas autorisé).*

● *Multer ne sauvegarde pas les images.*

● *Les images ne sont pas affichées sur le frontend.*

● *Les données ne sont pas modifiées lorsque l'utilisateur tente de*

*modifier une sauce existante.*

● *La fonction « modifier » échoue lorsqu'une image est téléchargée ou*

*modifiée.*

● *La fonction « like » échoue lorsque l'utilisateur essaie de liker ou de*

*ne pas aimer une sauce plusieurs fois.*

● *Le propriétaire de la sauce ne peut pas voir les boutons « modifier »*

*et « supprimer ».* L'identifiant de la Sauce doit être valide et ne pas

contenir de faute de frappe, car seul le propriétaire de la Sauce peut

la modifier ou la supprimer.

🎉**Projet terminé !**