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 :

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 :

Étape 2 : Construire le parcours utilisateur

**30 %** d’avance

● Créez les éléments suivants :

X Modèle d'utilisateur ;

X Parcours utilisateur ; (signup, login, )

○ 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 controleurs 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

\*/ implementer 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) ;
* 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.