



Construction d'une API sécurisée pour une application d'avis gastronomiques



VS CODE



Chrome



Réalisé par
EW. EL-KHABOU

Construire une API sécurisée pour une application d'avis gastronomiques

Sommaire

- 1 . [Scénario](#)
- 2 . [Objectifs et Mission](#)
- 3 . [Outils & Technologies utilisés](#)
- 4 . [Structure du site](#)
- 5 . [Comment utiliser l'API](#)
 - a . [Le Back-end](#)
 - b . [Le Front-end](#)
6. [Mesures de sécurité mises en place](#)
7. [Compétences évaluées](#)
8. [Soutenance & Évaluation](#)
9. [Démo / code](#)
10. [Bonus – 1 / Création du répertoire image si il n'existe pas](#)
11. [Bonus - 2 / \(fichier .env\)](#)
11. [Conclusion](#)

Construire une API sécurisée pour une application d'avis gastronomiques

Scénario :

Vous êtes développeur backend freelance et vous travaillez depuis quelques années sur des projets web pour des startups ou des grandes entreprises.

La semaine dernière, vous avez reçu un mail vous proposant un nouveau projet. La marque PIQUANTE, qui crée des sauces piquantes, connaît un franc succès.

L'entreprise souhaite désormais développer une application d'évaluation de ses sauces piquantes, appelée "Piquante", et même si l'application deviendra peut-être un magasin en ligne dans un futur proche, Sophie, la product owner de « Piquante », a décidé que le MVP du projet sera une application web permettant aux utilisateurs d'ajouter leurs sauces préférées et de liker ou disliker les sauces ajoutées par les autres utilisateurs.

La deadline fixée pour la réalisation du projet étant raisonnable, vous décidez d'accepter la mission, sachant que vos connaissances de la stack Node.js, Express et Mongo, et d'OWASP, sont parfaitement adaptées.

Vous trouverez ci-joint les spécifications pour l'API ainsi que le lien du Dépôt de l'application :

- [lien vers le repo du projet](#) où vous aurez accès à l'interface.
- [Spécifications techniques](#) de l'API

Objectifs et Mission :

- Construire le back-end et une **API REST** sécurisée pour une application d'avis gastronomiques “**PIQUANTE**”, une nouvelle application de So Pekocko qui permet aux utilisateurs de **consulter, ajouter, modifier, supprimer et de donner son avis ‘aimer ou pas aimer’** les sauces piquantes proposées par les autres utilisateurs.
- Utilisez un serveur Node.js, le framework Express, la base de données MongoDB, le plugin Mongoose, avec un hébergement sur MongoDB Atlas.
- L'API doit être sécurisé et respecter les normes OWASP et le GDPR.
- Le serveur frontal est déjà construit.
- Hébergement sur MongoDB Atlas
- Opérations relatives à la BDD réalisées avec mongoose

Outils & Technologies utilisés :

- JavaScript : Version 8.4.371.23
- Node.js : Version 14.18.0
- Node-sass : Version 4.14.
- Angular CLI : Version 12.2.8
- Express : Version 4.17.1
- MongoDB Atlas : Version 3.6.8
- Mongoose : Version 6.0.8
- Body-parser : Version 1.19.0
- Bcrypt : Version 5.0.1
- Dotenv : Version 10.0.0
- Jsonwebtoken : Version 8.5.1
- Mg-unique-validator : Version 2.0.4
- Multer : Version 1.4.3

Structure du site :

- Page d'inscription / connexion.
- Page d'accueil affichant toutes les sauces.
- Page affichant les informations d'une sauce spécifique, avec des options pour aimer / ne pas aimer la sauce.
- Page pour ajouter une nouvelle sauce.

Comment utiliser l'API :

Clonez ce dépôt.

Le Back-end :

- Dans le dossier « **backend** », et afin de vous connecter à la base de données en tant qu' « Admin » ou « Éditeur », copiez le fichier **.env** correspondant (envoyé séparément) dans le dossier « **backend** ».
- Installez nodemon. Exécutez **npm install**, Toujours dans le répertoire « **backend** »
- Exécutez **nodemon server**.
- Le serveur doit fonctionner sur **http://localhost:3000**.

Le Front-end :

- Ce dossier est déjà construit, codé et fourni, il ne faut pas le toucher.
- Dans le dossier « **frontend** », exécutez **npm start**. Cela devrait à la fois exécuter le serveur local et lancer votre navigateur.
- Si votre navigateur ne démarre pas ou affiche une **erreur 404**, essayez l'adresse : <http://localhost:8080> ou bien <http://localhost:8081> ou bien <http://localhost:5500>.
- L'application devrait se recharger automatiquement lorsque vous modifiez un fichier.
- Utilisez Ctrl+C dans le terminal pour arrêter le serveur local

Mesures de sécurité mises en place :

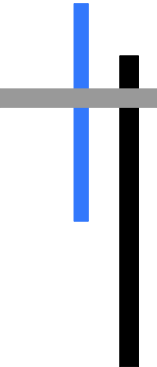
- Hashage du mot de passe utilisateur avec **bcrypt**.
- Manipulation sécurisée de la base de donnée avec **mongoose**.
- Vérification que l'e-mail utilisateur soit unique dans la base de données avec **mongoose-unique-validator**.
- Utilisation de variables d'environnement pour les données sensibles avec **dotenv**.
- Authentification de l'utilisateur par **token** avec **jsonwebtoken**.
- Protection des headers avec **helmet**.



Compétences évaluées :

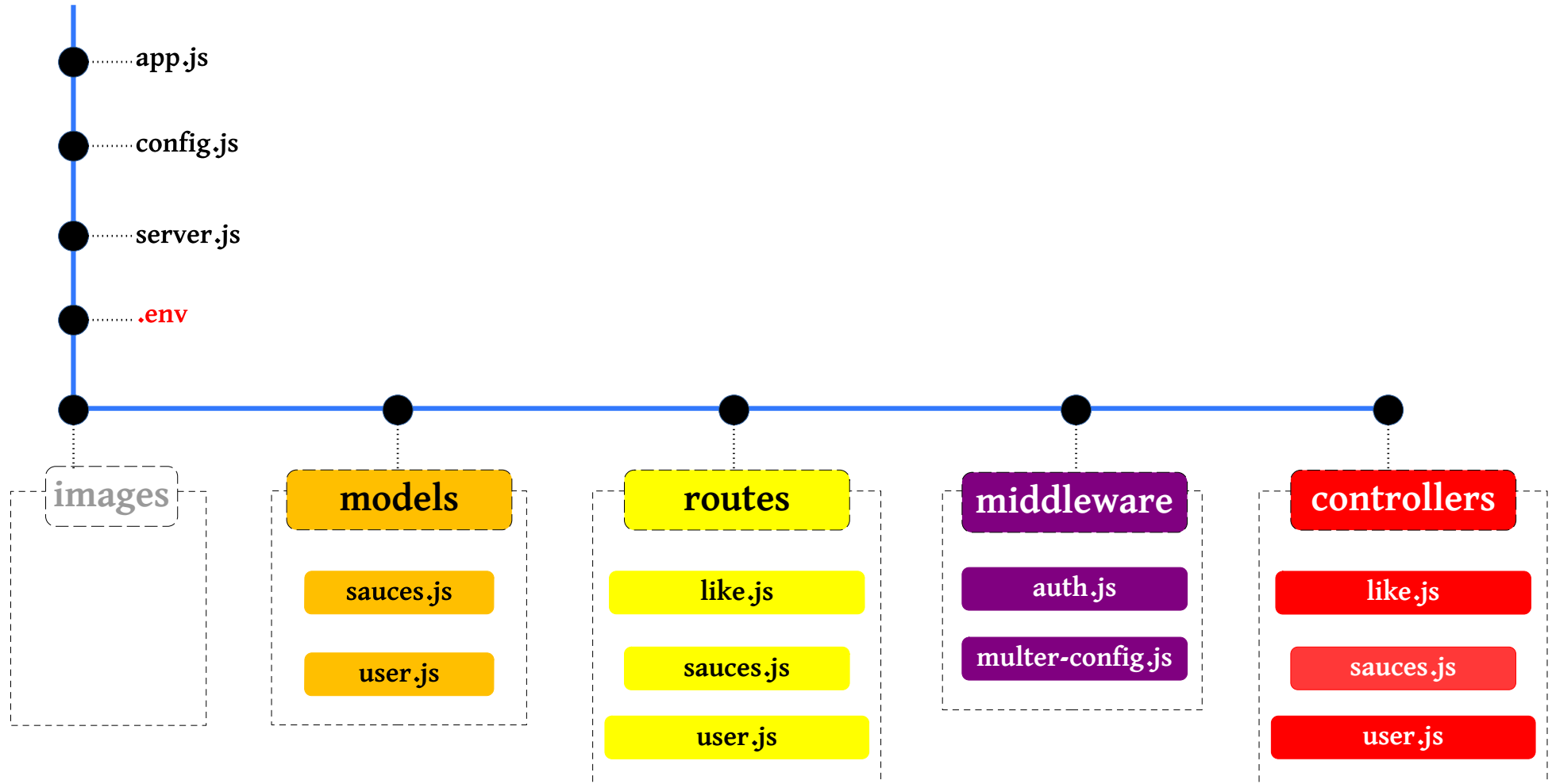
- Mettre en place un modèle de données logique conforme à la réglementation.
- Stocker les données en toute sécurité.
- Mettre en œuvre en toute sécurité les opérations CRUD.
- (CRUD = Create, Read, Update and Delete).
- Tokens d'authentification.
- Middleware d'authentification.
- Nodemon.

Soutenance & Évaluation :

- Évaluation et Soutenance planifiées pour le Mardi 19 Octobre 2021 à 14H00.
 - Cette Évaluation sera assurée par [Monsieur Ibrahima CISS](#) , Ingénieur logiciel développant pour iOS, macOS et le Web.
- 

Répartition des fichiers dans le **Back-End**

Racine du Back-End



Démo / code :

Nous commençons par télécharger Node depuis son site officiel <https://NodeJS.org> et nous l'installons, cela installera aussi Node Package Manager ou npm, outil précieux pour l'installation des packages nécessaires à la création de vos projets.

Puis, Il vous faudra également la CLI Angular pour pouvoir faire tourner le serveur de développement sur lequel sera exécuté le code du front-end. Pour l'installer, exécutez la commande suivante à partir de votre console :

```
npm install -g @angular/cli
```

Puis Nous créons un répertoire appelé « *frontend* » dans lequel nous allons cloner le Dépôt du projet avec la commande :

```
git clone https://github.com/OpenClassrooms-Student-Center/Web-Developer-P6
```

Une fois cela est fait, nous pouvons faire ce qui suit :

```
cd frontend  
npm install  
ng serve
```

Cela installera toutes les dépendances requises par l'application *frontend* et lancera le serveur de développement

Désormais, si vous accédez à <http://localhost:4200>, vous devriez voir l'interface suivante :
(en supposant que vous avez bien suivi les étapes ci-dessus) :





HOT TAKES

THE WEB'S BEST HOT SAUCE
REVIEWS

SIGN UP **LOGIN**

Email

Password

LOGIN

Nous devrions aussi créer le répertoire backend à la racine de notre projet, et exécuter la commande ***npm init***.

Initialisation du projet :

Pour éviter de redémarrer le serveur à chaque fois qu'on exécute une nouvelle commande, Nous installons globalement sur la machine un outil très pratique qui nous évite le redémarrage manuel du serveur a chaque fois, et qui redémarre le serveur automatiquement après chaque nouvelle instruction enregistrée, ou une nouvelle version des fichiers

```
npm install -g nodemon
```

A partir de maintenant, nous utiliserons au départ la nouvelle commande ***nodemon server***, au lieu de ***node server***.

Installation du framework Express :

Coder des serveurs Web en Node pur est possible, mais long et laborieux. En effet, cela exige d'analyser manuellement chaque demande entrante. L'utilisation du framework Express simplifie ces tâches, en nous permettant de déployer nos API beaucoup plus rapidement. Installons-le maintenant.

Nous procédons à l'installation de ce fameux framework par la commande suivante, qui enregistrera Express dans le package.json

```
npm install --save express
```

Nous commencerons par créer deux fichiers à la racine du répertoire ***backend*** et qui sont :

app.js et *server.js*

Nous allons maintenant mettre Express dans notre constante, ou application ***app.js***, et nous allons aussi l'exporter de sorte que tous les autres fichiers poussent l'utiliser notamment notre serveur NODE ; et afin que notre application trouve le moyen de répondre. Configurons une réponse simple pour nous assurer que tout fonctionne correctement, en ajoutant une simple fonction ***aux lignes 5 et 6***.

```
1  const express = require('express');
2
3  const app = express();
4
5  app.use((req, res) => {
6    res.json({ message: 'Votre requête a bien été reçue !' });
7  });
8
9  module.exports = app;
```

De la même façon, nous créons aussi le serveur *server.js* tout en lui indiquant le port sur lequel l'application reçoit nos requêtes http, et bien évidemment, vérifier immédiatement le bon fonctionnement de tout cela à l'aide d'un outil comme ***POSTMAN*** ou ***TALEND API TESTER***.

```
1  const http = require('http');
2  const app = require('./app');
3
4  app.set('port', process.env.PORT || 3000);
5  const server = http.createServer(app);
6
7  server.listen(process.env.PORT || 3000);
```

Ajoutez des middleware :

Une application Express est fondamentalement une série de fonctions appelées middleware. Chaque élément de *middleware* reçoit les objets `request` et `response`, peut les lire, les analyser et les manipuler, le cas échéant. Le *middleware* Express reçoit également la méthode `next`, qui permet à chaque *middleware* de passer l'exécution au *middleware* suivant. Voyons comment tout cela fonctionne.

```
29 //Récupère un produit par l'id
30 // nous utilisons la méthode get() pour récupérer une (sauce); Nous utilisons
31 // deux-points : en face du segment dynamique de la route pour la rendre accessible en tant que paramètre ;
32 // nous utilisons ensuite la méthode findOne() dans notre modèle "Sauce" pour trouver la "Sauce" unique ayant le
33 // même _id que le paramètre de la requête ; cette "Sauce" est ensuite retournée dans une Promise et envoyée au
34 // front-end ; si aucune "Sauce" n'est trouvée ou si une erreur se produit, nous envoyons une erreur 404 au
35 // front-end, avec l'erreur générée.
36 exports.getOneSauce = (req, res, next) => {
37   Sauce.findOne({
38     _id: req.params.id
39   }).then(
40     (sauce) => {
41       //retour promise status OK
42       res.status(200).json(sauce);
43     }
44   ).catch(
45     (error) => {
46       //retour promise erreur serveur
47       res.status(404).json({
48         error: error
49       });
50     }
51   );
52 };
```

Erreurs de CORS :

Dans notre cas, nous avons 2 origines : le backend fonctionne sur le port 3000 = <http://localhost:3000> et le frontend fonctionne sur le port 8081 = <http://localhost:8081> .

Hors que CORS qui signifie « **Cross Origin Resource Sharing** » et qui est un système de sécurité qui, par défaut, bloque les appels HTTP d'être effectués entre des serveurs différents, ce qui empêche donc les requêtes malveillantes d'accéder à des ressources sensibles. Et puisque nous souhaiterions que les deux origines puissent communiquer entre eux, nous devrions ajouter des headers à notre objet response .

```
40 // CORS (Cross-Origin Resource Sharing): Suite aux problemes CORS détectés par le Browser qui refuse
41 // (sécurité par défaut) d'executer nos requete car le front et le back sont differents : port 3000 et port 4200,
42 // nous devons inserer ces headers qui permettent de :
43 // 1- d'accéder à notre API depuis n'importe quelle origine ( '*' ) ;
44 // 2- d'ajouter les headers mentionnés aux requêtes envoyées vers notre API (Origin , X-Requested-With , etc.) ;
45 // 3- d'envoyer des requêtes avec les méthodes mentionnées ( GET ,POST , etc.).
46 app.use((req, res, next) => {
47   res.setHeader('Access-Control-Allow-Origin', '*');
48   res.setHeader('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content, Accept, Content-Type, Authorization');
49   res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS');
50   next();
51 });
```

Recevez des articles de l'application front-end : **body-parser** :

Même avant de configurer une base de donnée, nous pouvons intercepter les requêtes **POST** envoyées à notre route route api/sauce venues du formulaire « ajouter une sauce » du **frontend**.

nous devons être capables d'extraire l'objet **JSON** de la demande. Il nous faudra le package **body-parser** . Nous l'installerons en tant que dépendance de production à l'aide de npm :

```
npm install --save body-parser
```

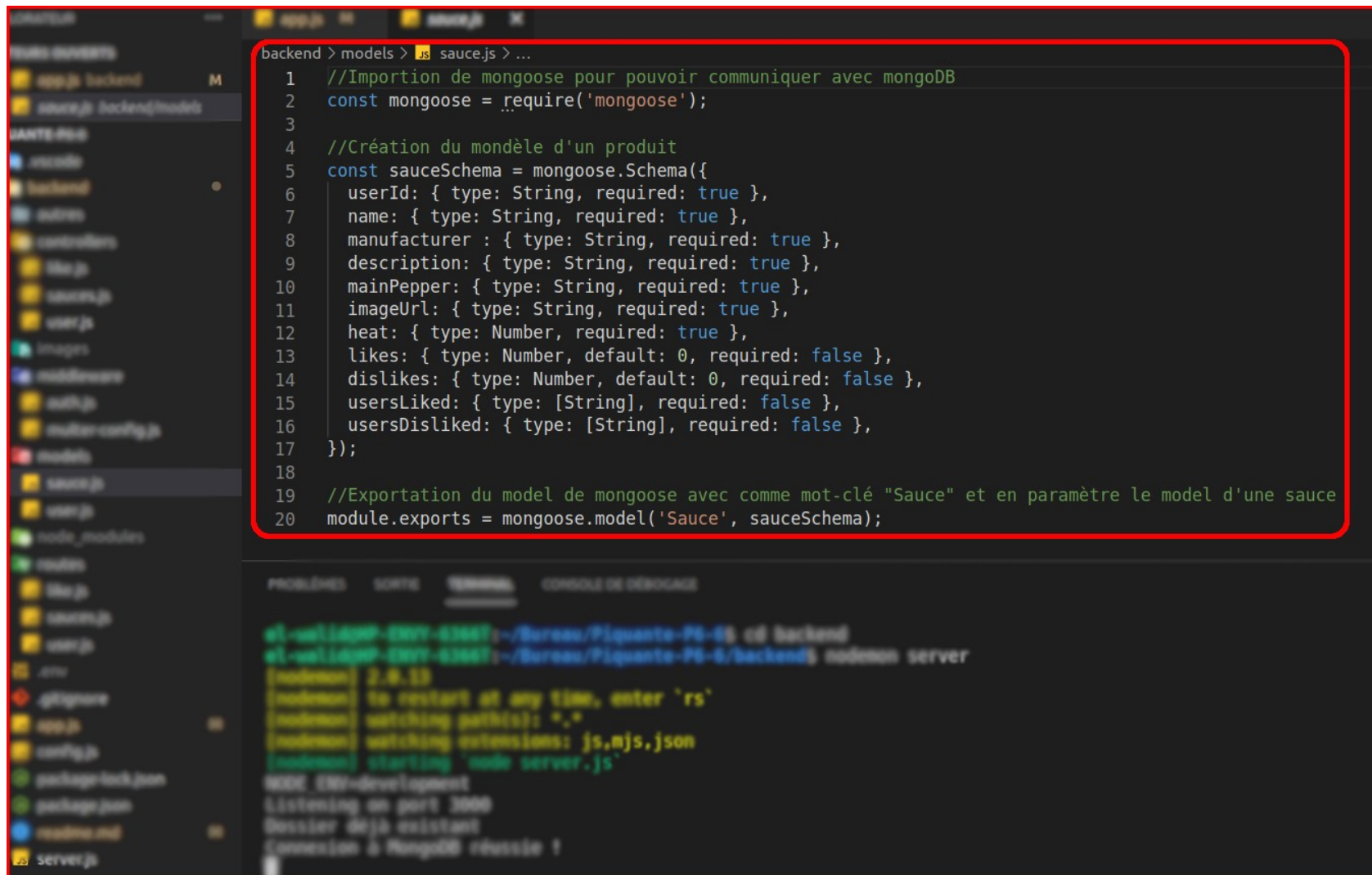
Recevez des articles de l'application front-end : **body-parser** :

Nous créons un compte gratuit sur MongoDB Atlas, et apres sa configuration, nous revenons vers notre fichier app.js afin d'installer mongoose afin de faciliter les interactions avec notre base de données. Pour l'installer, nous utiliserons la commande suivante :

```
npm install --save mongoose
```

Une fois tout cela est fait, et grâce à **Mongoose**, nous allons maintenant commencer à créer des **schémas** de nos données.

Voici un exemple de Shéma de donnée, qui consiste en un objet 'Sauce'.



```
backend > models > .js sauce.js > ...
1 //Importation de mongoose pour pouvoir communiquer avec mongoDB
2 const mongoose = require('mongoose');
3
4 //Création du modèle d'un produit
5 const sauceSchema = mongoose.Schema({
6   userId: { type: String, required: true },
7   name: { type: String, required: true },
8   manufacturer : { type: String, required: true },
9   description: { type: String, required: true },
10  mainPepper: { type: String, required: true },
11  imageUrl: { type: String, required: true },
12  heat: { type: Number, required: true },
13  likes: { type: Number, default: 0, required: false },
14  dislikes: { type: Number, default: 0, required: false },
15  usersLiked: { type: [String], required: false },
16  usersDisliked: { type: [String], required: false },
17 });
18
19 //Exportation du model de mongoose avec comme mot-clé "Sauce" et en paramètre le model d'une sauce
20 module.exports = mongoose.model('Sauce', sauceSchema);
```

```
al-wali@MP-ENV-61667: ~/Bureau/Piquante-PG-6$ cd backend
al-wali@MP-ENV-61667: ~/Bureau/Piquante-PG-6/backend$ nodemon server
[nodemon] 2.0.13
[nodemon] to restart at any time, enter `rs`
[nodemon] watching path(s): *.*
[nodemon] watching extensions: js,mjs,json
[nodemon] starting 'node server.js'
NODE_ENV=development
Listening on port 3000
Dossier déjà existant
Connexion à MongoDB réussie !
```


Implémentation de la route GET pour récupérer toutes les sauces :

```
// Dans l'exemple ci-dessous, nous utilisons la méthode find() dans notre modèle Mongoose
// afin de renvoyer un tableau contenant toutes les "Sauces" dans notre base de données.
// À présent, si nous ajoutons une "Sauces" , elle doit s'afficher immédiatement sur notre
// page d'articles en vente.
exports.getAllSauces = (req, res, next) => {
  Sauce.find().then(
    (sauces) => {
      //retour promise status OK
      res.status(200).json(sauces);
    }
  ).catch(
    (error) => {
      //retour erreur requête
      res.status(400).json({
        error: error
      });
    }
  );
};
```

Récupération d'un seul produit par son id :

```
//Récupère un produit par l'id
// nous utilisons la méthode get() pour récupérer une (sauce); Nous utilisons
// deux-points : en face du segment dynamique de la route pour la rendre accessible en tant que paramètre ;
// nous utilisons ensuite la méthode findOne() dans notre modèle "Sauce" pour trouver la "Sauce" unique ayant le
// même _id que le paramètre de la requête ; cette "Sauce" est ensuite retournée dans une Promise et envoyée au
// front-end ; si aucune "Sauce" n'est trouvée ou si une erreur se produit, nous envoyons une erreur 404 au
// front-end, avec l'erreur générée.
exports.getOneSauce = (req, res, next) => {
  Sauce.findOne({
    _id: req.params.id
  }).then(
    (sauce) => {
      //retour promise status OK
      res.status(200).json(sauce);
    }
  ).catch(
    (error) => {
      //retour promise erreur serveur
      res.status(404).json({
        error: error
      });
    }
  );
};
```

Modification d'un Produit :


```
// Modification d'un produit :  
// Ci-dessus, nous exploitons la méthode updateOne() dans notre modèle "Sauce" . Cela nous permet de mettre à jour la  
// "Sauce" qui correspond à l'objet que nous passons comme premier argument. Nous utilisons aussi le paramètre id  
// passé dans la demande et le remplaçons par la "Sauce" passée comme second argument.  
exports.modifySauce = (req, res, next) => {  
  //créer un objet et cherche si req.file existe déjà  
  const sauceObject = req.file ?  
  {  
    //Si modification de l'image de l'objet, on récupère le body du produit en parse  
    ...JSON.parse(req.body.sauce),  
    //et on modifie l'imageUrl  
    imageUrl: `${req.protocol}://${req.get('host')}/images/${req.file.filename}`  
    //Si modification de chaîne de caractère on modifie le body  
  } : { ...req.body };  
  Sauce.updateOne({ _id: req.params.id }, { ...sauceObject, _id: req.params.id })  
    //retour promise status OK  
    .then(() => res.status(200).json({ message: 'Objet modifié !'}))  
    //retour erreur requête  
    .catch(error => res.status(400).json({ error }));  
};
```

Suppression d'un Produit :

```
// Suppression d'un produit
// La méthode deleteOne() de notre modèle fonctionne comme findOne() et updateOne() dans le sens où nous lui passons
// un objet correspondant au document à supprimer. Nous envoyons ensuite une réponse de réussite ou d'échec au front-end.
exports.deleteSauce = (req, res, next) => {
  Sauce.findOne({ _id: req.params.id })
    .then(sauce => {
      //Récupère le nom du fichier
      //split => récupère avant ../images/ et après /images/...
      //avant ../images positionnement 0 //après /images/... positionnement 1
      //donc on choisi 1 car permettant de récupérer l'url de l'image qui est après le dossier images
      const filename = sauce.imageUrl.split('/images/')[1];
      // fs.unlink supprime le nom du fichier dans le dossier images
      fs.unlink(`images/${filename}`, () => {
        //callback retour on supprime également le produit par son id
        Sauce.deleteOne({ _id: req.params.id })
          //retour promise status OK
          .then(() => res.status(200).json({ message: 'Objet supprimé !'}))
          //retour erreur requête
          .catch(error => res.status(400).json({ error }));
      });
    });
  //retour erreur communication avec le serveur
  .catch(error => res.status(500).json({ error }));
};
```

Une fois que nous aurons dans notre fichier **app.js** :

1. Notre logique pour se connecter à MongoDB
2. Tous ce qui est relatif au CORS
3. On a configuré Body-Parser
4. Et on a toutes nos routes



nous allons réorganiser la structure de notre **back-end** pour en faciliter la compréhension et la gestion, nous transférons donc toutes nos routes créées du fichier **app.js** vers le fichier **/routes/sauces**.

Nous créons donc un routeur Express tout en remplaçant toutes les occurrences de **app** par **router** , car nous enregistrons les routes dans notre routeur.

NB : **/api/sauces** doit être supprimé de chaque segment de route. Si cela supprime une chaîne de route, veuillez à laisser une barre oblique **/** . Nous devons aussi remplacer l'élément **app.use()** final par **app.get()** puisque cette route ne concerne que les demandes **GET**.

Les Logiques ROUTING et METIER :

Afin de mieux comprendre notre code, nous allons mettre toutes les routes logiques dans le fichier **/routes/sauces** avec des noms sementiques précis, et mettre toutes les logiques fonctions dans un autre fichier **/controllers/sauces**

Voir le fichier **/routes/sauces.js** en entier

Voir le fichier **/controllers/sauces.js** en entier



Puisque tout fonctionne bien pour consulter toutes les sauces créées par les différents utilisateurs, ainsi que la création, la modification et la suppression de nouvelles sauces, nous allons maintenant passer à la création des logiques de *signup* et *login* :

Les Logiques de Signup et de LogIn :

Le model *user.js*

```
//Importation de mongoose pour pouvoir communiquer avec mongoDB
const mongoose = require('mongoose');

//Utiliser la méthode de mongoose "mongoose-unique-validator" permettant de définir un mail user unique
const uniqueValidator = require('mongoose-unique-validator');

const userSchema = mongoose.Schema({
  // 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
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true }
});

// 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
userSchema.plugin(uniqueValidator);

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

La route *user.js*

```
1 //Importation de express dans une constante
2 const express = require('express');
3
4 //Appliquer les routes à express dans une constante
5 const router = express.Router();
6
7 //Importer le fichier /controllers/user.js
8 const userCtrl = require('../controllers/user');
9
10
11 //Routes pour avec la création et l'authentification de user
12 router.post('/signup', userCtrl.signup);
13 router.post('/login', userCtrl.login);
14
15 //Exporter les routes
16 module.exports = router;
```

Voir le fichier /controllers/user.js en entier

BONUS N° 1 :

Dans le cas où le répertoire (/Images) n'existe pas, nous avons ajouté ce qui suit afin que le serveur vérifie son existence, et le créer s'il n'existe pas. Pour ce fait, nous avons inséré ce qui suit dans le fichier :

/middleware/multer-config.js

```
17 //Création du dossier "images" s'il n'existe pas
18 fs.mkdir("./images",function(image){
19     if(!image || (image && image.code === 'EEXIST')){
20         console.log("Dossier déjà existant")
21     } else {
22         console.log("Votre dossier a bien été créé")
23     }
24 })
```


BONUS N° 2 :

Pour accroître la sécurité des noms d'utilisateur et des mots de passes, nous avons créé sur la racine un fichier **config.js** responsable de l'authentification à l'aide du fichier masquée **.env** contenant les vrais Mots de passe et username, sans jamais savoir qu'il s'agit d'une base de donnée MongoDB.

```
backend > js config.js > ...
1  const dotenv = require('dotenv');
2  const path = require('path');
3
4  dotenv.config({path: path.resolve(__dirname, `${process.env.NODE_ENV}.env`)});
5  module.exports = {
6    NODE_ENV : 'development',
7    DB_USERNAME : process.env.DB_USERNAME,
8    DB_PASSWORD : process.env.DB_PASSWORD,
9    HOST : 'localhost',
10   PORT : 3000
11 }
```

CONCLUSION

Bien qu'il reste beaucoup à faire, Grace à ce projet, nous avons enrichie beaucoup notre connaissance en terme de codage et en javascript, notamment dans les termes et modules suivants :

API : REST, CRUD, Express + **Les Base de donnée noSQL :** Mongodb, Mongoose, Mongo Atlas, et La différence entre une base de donnée SQL et noSql, + Multer, Middleware, Http code, Token / JWT, Bcrypt, Mongoose-unique-validator, + **Les Variable d'environnement :** dotenv, .env, config.js + l'OWASP.

Grace à ce projet, je sais maintenant :

- 1) *Implémenter un modèle logique de données conformément à la réglementation*
- 2) *Stocker des données de manière sécurisée*
- 3) *Mettre en œuvre des opérations CRUD de manière sécurisée*