# Construire une API sécurisée pour une application d'avis gastronomique

## Objectif du projet

Implémenter un modèle logique de données conformément à la réglementation

Stocker des données de manière sécurisée

Mettre en œuvre des opérations CRUD de manière sécurisée

## Contexte

Après une dernière année en tant que développeur back-end indépendant sur différents projets de taille et de difficultés variées, J'ai reçu une demande d'aide pour un nouveau projet pour la marque de condiment Piiquante. En effet, elle veut développer une application web de critique des sauces piquantes appelée « Hot Takes ».

## Présentation du projet

Le projet sera présenté sur 3 points:

- Explication du fonctionnement du code, en particulier le middleware d'authentification.
- Explication de la structure du code.
- Explication des méthodes pour sécuriser la base de données selon le RGPD et l'OWASP.

### Explication du fonctionnement du code, middleware d'authentification

Lorsqu'un utilisateur se connecte, il reçois un token encodé. Le front-end va alors lier ce token à chaque requête et le back-end pourra vérifier que celles-ci sont authentifiées.

Pour cela, on a créé le middleware d'authentification, il aura besoin du package "jsonwebtoken". Ce package permet de créer et vérifier des tokens d'authentification.

Voici la structure du code du login utilisateur qui se trouve dans le contrôleur user.js

*exports.login = (req, res, next) => {*

On chercher d'abord l'utilisateur dans la base de donnée

*User.findOne({ email: req.body.email })*

*.then(user => {*

*if (!user) {*

Si celui-ci n'est pas trouvé on renvoi une erreur "Utilisateur non trouvé !"

*return res.status(401).json({ error: 'Utilisateur non trouvé !' });*

*}*

On compare le mot de passe utilisateur avec celui de la base de donnée

```
  bcrypt.compare(req.body.password, user.password)
   .then(valid => {
    if (!valid) {
```

Si différent on renvoi une erreur "Mot de passe incorrect !"

```
     return res.status(401).json({ error: 'Mot de passe incorrect !' });
    }
    res.status(200).json({
```

Sinon on renvoi l'objet avec un token encodé contenant le userid, un clé et une durée d'expiration de celui-ci

```
     userId: user._id,
     token: jwt.sign(
      { userId: user._id },
      process.env.TOKEN,
      { expiresIn: '24h' }
     )
    });
   })
   .catch(error => res.status(500).json({ error }));
 })
 .catch(error => res.status(500).json({ error }));
};
```

Une fois l'utilisateur connecté, à chaque requête on utilisera le middleware d'authentification pour vérifié que celle-ci est authentique.

exemple: afficher une sauce

*router.get('/:id',**auth**, sauceCtrl.getOneSauce);*

Avant d'exécuter le code d'affichage d'une sauce, on passe par le middleware d'authentification.

Celui est codé de la manière suivante:

initialisation du package "jsonwebtoken"

```
const jwt = require('jsonwebtoken');
require('dotenv').config()
module.exports = (req, res, next) => {
 try {
```

On récupère seulement le token du header authorization de la requête

```
  const token = req.headers.authorization.split(' ')[1];
```

Décode le token en le comparant à celui présent dans la fonction de login

```
const decodedToken = jwt.verify(token, process.env.TOKEN );
```

on récupére le userid

```
const userId = decodedToken.userId;
```

```
req.auth = { userId };
```

On vérifie s'il y a un userId dans la requête et que celui ci est différent de l'user Id alors

```
if (req.body.userId && req.body.userId !== userId) {
```

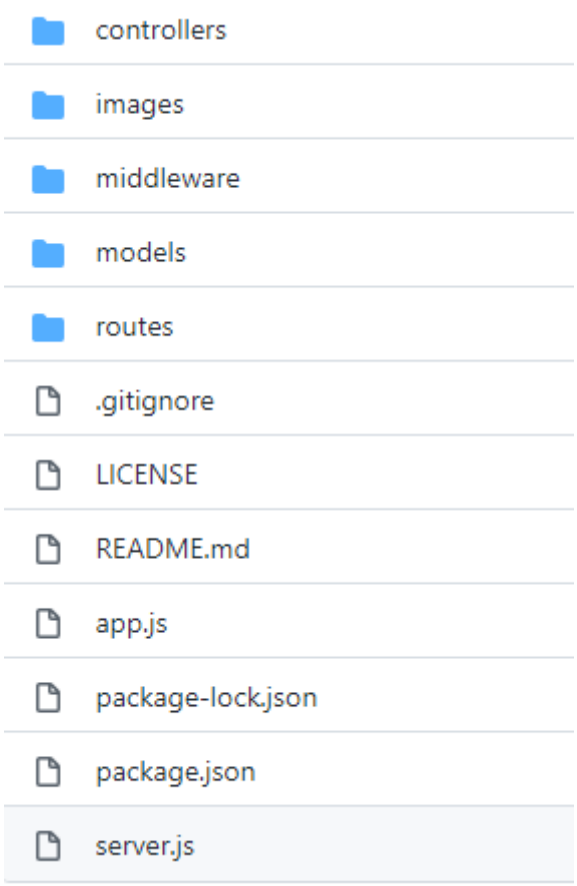
On renvoie l'erreur

```
  throw 'Invalid user ID';
 } else {
  next();
 }
} catch {
 res.status(401).json({
  error: new Error('Invalid request!')
 });
}
};
```

## Explication de la structure du code

### Structure générale

Le code est structuré de la manière suivante:




J'ai pris le parti de compartimenter mon code.

L'application et le server sont à la base du projet ensuite les fichiers sont dispatché selon leur fonctionnalité dans des répertoires spécifiques.  Cela permet au développeur d'aller directement dans le bon répertoire si il y a une modification à faire.

- routes : Liste de toutes les routes de l'applications
- models : la définition de schéma du modèle
- middleware : Ensemble de fonction qui s'exécute lors de la requête au serveur
- controllers : Les contrôleurs gèrent tout la logique de traitement
- images : répertoire où seront stockés les images sauvegardé par l'application.

**structure du code app.js**

Pour créer cette application, nous utilisons Express qui est un framework de NodeJS et nous utiliseront MongoDB comme base de donnée.

1. on se connecte à la base de donnée

*mongoose.connect (process.env.DATABASE_MONGO,*

*{ useNewUrlParser: true,*

*useUnifiedTopology: true })*

*.then(() => console.log('Connexion à MongoDB réussie !'))*

*.catch(() => console.log('Connexion à MongoDB échouée !'));*

2. On prend toutes les requêtes qui on un contenu JSON et on le met à disposition dans l'objet request

app.use(express.json());

3. pour que le front-end et le back-end communiquent correctement et de manière sécurisés nous allons permettre de faire accéder l'api depuis n'importe quelle origine('*'), ajouter des headers aux requêtes envoyées vers notre API et permettre l'envoi de requêtes par les méthodes mentionnée

*app.use((req, res, next) => {*

*res.setHeader('Access-Control-Allow-Origin', '');**

*res.setHeader('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content, Accept, Content-Type, Authorization');*

*res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS');*

*next();*

*});*

4. on enregistre nos routeurs pour toutes les demandes effectuées vers /api/auth ou /api/sauces/ et nous enregistrons le répertoire de sauvegarde des images

   *app.use("/images", express.static(path.join(__dirname, 'images')));*

   *app.use('/api/auth',userRoutes);*

   *app.use('/api/sauces',sauceRoutes);*

**Exemple de route-controller**

Dans le fichier route sauce.js, nous utilisons le package Express appelé Router.

*const express = require('express');*

*const router = express.Router();*

Le Router aide à créer une liste de routes et à les associer à un fichier de contrôleur contenant le code d'implémentation.

Pour l'exemple nous prendrons la définition de la route qui permet de gérer les like et les dislike

Il faut tout d'abord déclarer le chemin du contrôleur

*const sauceCtrl = require('../controllers/sauce');*

ensuite lier la route post au contrôleur

*router.post('/:id/like', auth, sauceCtrl.likeSauce);*

Les paramétres auth et multer sont deux middleware d'authentification (auth) et de téléchargement de fichier (multer)

Dans le fichier contrôleur sauce.js, nous retrouvons le code d'implémentation de la gestion des like/dislike.

*exports.likeSauce = (req, res, next) => {*

Si like est à 1

*if (req.body.like === 1) { *

On modifie celui dont l'ID est égale à l'ID envoyé dans les paramètres de requêtes avec Likes a 1 et userId dans le tableau usersLiked

*Sauce.updateOne( *

*{ _id: req.params.id },*

*{ $inc: { likes: 1 },$push: { usersLiked: req.body.userId }},*

*{ _id: req.params.id }*

*)*

Si l'opération s'est bien passée on renvoi "j'aime" à la console sinon on renvoi le message d'erreur

*.then(() => res.status(200).json({ message: 'J'aime' }))*

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

*}

Si like est à -1

else if (req.body.like === -1) { *

On modifie celui dont l'ID est égale à l'ID envoyé dans les paramètres de requêtes avec Dislikes a 1 et userId dans le tableau userDisLiked

*Sauce.updateOne( *

*{ _id: req.params.id },*

*{ $inc: { dislikes: 1 },$push: { usersDisliked: req.body.userId }},*

*{ _id: req.params.id }*

*)*

Si l'opération s'est bien passée on renvoi "Je n'aime pas" à la console sinon on renvoi le message d'erreur

*.then(() => res.status(200).json({ message: 'Je n'aime pas' }))*

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

*} else {*

sinon on trouve l'id de la sauce à modifier

*Sauce.findOne({ _id: req.params.id })*

*.then((sauce) => {*

si userId est présent dans le tableau usersLiked alors

*if (sauce.usersLiked.includes(req.body.userId)){ *

On modifie celui dont l'ID est égale à l'ID envoyé dans les paramètres de requêtes avec Like a -1 et en enlevant userId dans le tableau usersLiked

*Sauce.updateOne( *

*{ _id: req.params.id },*

*{ $inc: { likes: -1 },$pull: { usersLiked: req.body.userId }},*

*{ _id: req.params.id }*

*)*

Si l'opération s'est bien passée on renvoi "Je n'aime plus" à la console sinon on renvoi le message d'erreur

*.then(() => res.status(200).json({ message: 'Je n'aime plus' }))*

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

*} *

On modifie celui dont l'ID est égale à l'ID envoyé dans les paramètres de requêtes avec Dislike a -1 et en enlevant userId dans le tableau usersDisliked

*else if (sauce.usersDisliked.includes(req.body.userId)){*

*Sauce.updateOne( *

*{ _id: req.params.id },*

*{ $inc: { dislikes: -1 }, $pull: { usersDisliked: req.body.userId }},*

*{ _id: req.params.id }*

*)*

Si l'opération s'est bien passée on renvoi "Je commence à aimer" à la console sinon on renvoi le message d'erreur

*.then(() => res.status(200).json({ message: 'Je commence à aimer' }))*

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

*}}*

*)*

*.catch((error) => res.status(400).json({ message : 'ça marche pas' }));*

*}}*

## Explication des méthodes pour sécuriser la base de données selon le RGPD et l'OWASP.

**Mot de passe utilisateur crypté**

Lors de la création d'un utilisateur, on va crypter son mot de passe en utilisant l'algorithme de hashage bcrypt.

Dés lors à chaque connexion nous allons comparer le mot de passe entré avec le hash enregistré dans la base de donnée

**Utilisation d'un token d'authentification**

Lors de la connexion d'un utilisateur, le back-end va renvoyé au front-end un token qui contiendra son userid, une clé et date d'expiration de ce token.

Ce token sera ensuite envoyé par le front-end dés qu'il aura une demande à faire au back-end.

Le back-end récupère le token et le compare à celui envoyé lors du login. Si la comparaison est la requête est executé.

**CORS**

CORS signifie « **Cross Origin Resource Sharing** »

C'est un systeme de sécurité qui bloque les appels HTTP entre des serveurs différents, ce qui empêche donc les requêtes malveillantes d'accéder à des ressources sensibles.

Comme le back-end et le front-end ont deux origines différentes, il faut ajouter des headers dans l'objet response.

Nous devons donc ajouter dans notre application le code suivant qui permet de faire accéder l'api depuis n'importe quelle origine('*'), ajouter des headers aux requêtes envoyées vers notre API et permettre l'envoi de requêtes par les méthodes mentionnée

```
app.use((req, res, next) => {

 res.setHeader('Access-Control-Allow-Origin', '*');

 res.setHeader('Access-Control-Allow-Headers', 'Origin, X-Requested-With, Content, Accept, Content-Type, Authorization');

 res.setHeader('Access-Control-Allow-Methods', 'GET, POST, PUT, DELETE, PATCH, OPTIONS');

 next();

});
```