
RAPPORT D'AMÉLIORATION DU CODE

Ce document présente les améliorations de sécurité recommandées pour l'application **RecipeApp**, identifiées à la suite d'un audit de sécurité. L'analyse a mis en évidence plusieurs vulnérabilités critiques, notamment :

- Absence de validation des entrées utilisateur, exposant l'application aux injections SQL et attaques XSS.
- Stockage et gestion non sécurisés des mots de passe et des tokens d'authentification.
- Manque de protection contre les attaques CSRF et DoS.
- Exposition des identifiants de base de données dans le code source.

Afin de garantir la **confidentialité**, l'**intégrité** et la **disponibilité** des données, ce document propose des correctifs et des améliorations basées sur les meilleures pratiques en matière de sécurité des applications web.

Coté Back-end :

1. Sécuriser la base de données et les identifiants

Problème : Identifiants en dur de la base de données dans le fichier db.js

Solution : Utiliser des variables d'environnement dans le fichier .env

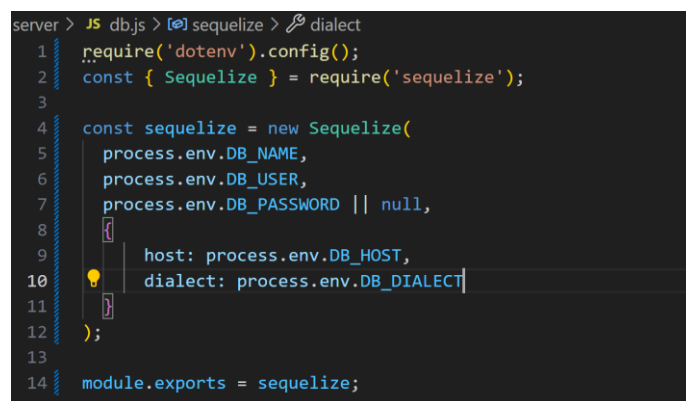
- **.env**



```
.env
1 DB_HOST=localhost
2 DB_USER=root
3 DB_PASSWORD=
4 DB_DIALECT=mysql
5 DB_NAME=recipeapp
```

Figure 1: Bout de code du fichier .env

- **Utilisation des identifiants de la base de données dans le fichier db.js**



```
server > JS db.js > [?] sequelize > dialect
1 require('dotenv').config();
2 const { Sequelize } = require('sequelize');
3
4 const sequelize = new Sequelize(
5   process.env.DB_NAME,
6   process.env.DB_USER,
7   process.env.DB_PASSWORD || null,
8   {
9     host: process.env.DB_HOST,
10    dialect: process.env.DB_DIALECT
11  }
12 );
13
14 module.exports = sequelize;
```

Figure 2: code amélioré de db.js

2. Stockage sécurisé de la clé JWT

Problème : La clé JWT stockés en clair et sa valeur est facile à deviner.

Solution : Régénération de la clé et utilisation d'une variable d'environnement dans .env.

- **.env**

```
DB_NAME=recipeapp
JWT_SECRET=97bdc7a0c17e830d5edf439606ae3ca32a7ae5c4fa73cbda1446f676f014061c7d2d66654cffe3d3e4cf8c57f09460750428b86053a5293b98fd0bcb5cf
```

Figure 3: bout de code du fichier .env

- **Exemple d'utilisation de la variable d'environnement dans user.js**

```
router.post("/login", async (req, res) => {
  const { username, password } = req.body;
  const user = await User.findOne({ where: { username: username } });

  if (!user) {
    return res.json({ message: "User Doesn't Exist !" });
  }

  const IsPasswordValid = await bcrypt.compare(password, user.password);

  if (!IsPasswordValid) {
    return res.json({ message: "Username or Password is Incorrect !" });
  }

  const token = jwt.sign({ id: user._id }, JWT_SECRET);
  res.json({ token, userId: user.id });
});
```

Figure 4: exemple d'utilisation de JWT_SECRET

3. Ajout d'un système de session avec Redis

Problème : Actuellement, le JWT est stocké dans un cookie, mais il n'y a pas de gestion de session sécurisée.

Solution : stocker les sessions avec Redis

- **Configurer Redis dans index.js :**

```
//Gérer les sessions avec Redis pour invalider les connexions à distance
const session = require('express-session');
const RedisStore = require('connect-redis')(session);
const redis = require('redis');

const redisClient = redis.createClient();

app.use(session({
  store: new RedisStore({ client: redisClient }),
  secret : process.env.JWT_SECRET,
  resave: false,
  saveUninitialized: false,
  cookie: {secure: true, httpOnly:true}
}));
```

Figure 5: bout de code de index.js.

4. Protéger l'API avec des en-têtes de sécurité HTTP

Problème : absence d'en-tête de sécurité.

Solution : Utilisation de Helmet pour améliorer la sécurité des cookies et en-têtes http ;

- **L'import de helmet dans le fichier index.js**

```
import { recipeRouter } from '../routes/recipes.js';
import helmet from "helmet";

const app = express();

app.use(express.json());
app.use(cors());
app.use(helmet());
// Routes
```

Figure 6: bout de code de index.js

5. Valider les entrées utilisateurs avant les requêtes

Problème : Les inputs des utilisateurs ne sont pas vérifiés.

Solution : Vérifier les inputs avant d'exécuter les requêtes et ajouter une validation des données avec Joi.

- **Utilisation de Joi :**

```
import { verifyToken } from './users.js';
import Joi from "joi";

const router = express.Router();

const recipeSchema = Joi.object({
  recipeId: Joi.number().integer().positive().required(),
  userID: Joi.number().integer().positive().required()
});
```

Figure 7: bout de code de routes/recipes.js

- **Vérifier les données dans req.body avant l'exécution des requêtes**

```
// PUT : Ajouter une recette aux recettes enregistrées de l'utilisateur
router.put("/", verifyToken, async (req, res) => {
  const { error } = recipeSchema.validate(req.body);
  if (error) {
    return res.status(400).json({ error: "Données invalides", details: error.details });
  }
  const { recipeId, userID } = req.body;

  try {
    const user = await User.findPk(userID);

    if (!user) {
      return res.status(404).json({ error: "Utilisateur non trouvé" });
    }

    const recipe = await Recipe.findPk(recipeId);

    if (!recipe) {
      return res.status(404).json({ error: "Recette non trouvée" });
    }

    const savedRecipes = user.savedRecipes ? JSON.parse(user.savedRecipes) : [];
    savedRecipes.push(recipeId);

    user.savedRecipes = JSON.stringify(savedRecipes);
    await user.save();
  } catch (error) {
    return res.status(500).json({ error: "Erreur serveur" });
  }
});
```

Figure : bout de code de routes/recipes.js

6. Empêcher les injections SQL (Op.in)

La solution : Utilisation de Op.in pour sécuriser findAll().

L'avantage : Convertit les valeurs en **nombres** et empêche des injections SQL via des chaînes de caractères.

```
// GET : Détails des recettes sauvegardées d'un utilisateur
router.get("/savedRecipes/:userID", async (req, res) => {
  try {
    const user = await User.findById(req.params.userID);

    if (!user) {
      return res.status(404).json({ error: "Utilisateur non trouvé" });
    }

    const savedRecipesIds = user.savedRecipes
      ? JSON.parse(user.savedRecipes)
      : [];

    const savedRecipes = await Recipe.findAll({
      where: { id: { [Op.in]: savedRecipesIds.map(Number) } }, // Sécurise les ID
    });

    res.json({ savedRecipes });
  } catch (err) {
    res.status(500).json({
      error:
        "Erreur lors de la récupération des détails des recettes sauvegardées",
      details: err,
    });
  }
});
```

Figure 8: bout de code de routes/recipes.js

7. Configuration d'une expiration pour les JWT

Problème : Le JWT n'a pas de durée d'expiration, ce qui cause des sessions infinies (un token volé reste valide indéfiniment), accès non révoqué (un user qui change de mot de passe peut toujours utiliser son ancien token) ou encore un risque d'exploitation (si un attaquant récupère un JWT, il peut l'utiliser indéfiniment).

Solution : Ajouter une expiration au JWT.

- Ajout de expiresIn dans la route /login :

```
router.post("/login", async (req, res) => {
  const { username, password } = req.body;
  const user = await User.findOne({ where: { username: username } });

  if (!user) {
    return res.json({ message: "User Doesn't Exist !" });
  }

  const IsPasswordValid = await bcrypt.compare(password, user.password);

  if (!IsPasswordValid) {
    return res.json({ message: "Username or Password is Incorrect !" });
  }

  const token = jwt.sign({ id: user.id }, JWT_SECRET, { expiresIn: '1h' });
  res.json({ token, userId: user.id });
});
```

Figure 9: bout de code de la fonction login dans routes/user.js

- Ajout de verify token dans route/users.js :

```
export const verifyToken = (req, res, next) => {
  const token = req.headers.authorization;
  if (token) {
    jwt.verify(token, "secret", (err) => {
      if (err) return res.sendStatus(403);
      next();
    });
  } else {
    res.sendStatus(401);
  }
};
```

Figure 10: la fonction verifyToken

COTE FRONT END :

1. Utilisation de SessionStorage au lieu de Local Storage

Problème : Le stockage de userID dans localStorage peut causer des attaques XSS, un accès non contrôlé (si un user se connecte sur un ordinateur public et oublie de se déconnecter laisse son userID exposé).

Solution : utilisation de sessionStorage pour que les données soient automatiquement supprimées lorsque l'utilisateur ferme l'onglet ou le navigateur.

- Remplacer localStorage par sessionStorage dans Auth.js :

```
const Login = () => {
  const [username, setUsername] = useState("");
  const [password, setPassword] = useState("");

  const [, setCookies] = useCookies(["access_token"]);

  const navigate = useNavigate();
  const onSubmit = async (event) => {
    event.preventDefault();
    try {
      const response = await axios.post("http://localhost:3001/auth/login", {
        username,
        password,
      });
      setCookies("access_token", response.data.token);
      window.sessionStorage.setItem("userID", response.data.userId);
      console.log("res", window.sessionStorage.getItem("userID"));
      navigate("/");
      console.log(response);
    } catch (err) {
      console.error(err);
    }
  };
};
```

Figure 11: bout de code de auth.js.

2. Utilisation des en-têtes CSRF :

Problème : L'application manque de protection contre les attaques CSRF ce qui peut engendrer une exploitation d'une session active à travers une requête malveillante ou sinon forcer une action sans le consentement de l'utilisateur.

Solution : ajouter un token CSRF qui est généré par le serveur et envoyé avec chaque requête sensible.

- **Générer le token dans index.js :**

```

1
2 app.use(express.json());
3 app.use(cookieParser());
4 app.use(cors({credentials: true, origin: "http://localhost:3000"}));
5 app.use(helmet());
6
7 //middleware CSRF Protection
8 const csrfProtection = csrf({ cookie: true }); // Utilisation des cookies pour stocker le token CSRF
9 app.use(csrfProtection);
10 //Route pour récupérer le token CSRF et l'envoyer au front
11 app.get("/csrf-token", (req, res) => {
12   res.json({ csrfToken: req.csrfToken() });
13 });
14
15 // Routes

```

Figure 12: bout de code de index.js

- **Utiliser le token dans le front (auth.js)**

```

1
2 const Login = () => {
3   const [username, setUsername] = useState("");
4   const [password, setPassword] = useState("");
5   const [, setCookies] = useCookies(["access_token"]);
6   const navigate = useNavigate();
7   const [csrfToken, setCsrfToken] = useState("");
8
9   //Récupérer le token CSRF au chargement de la page
10  useEffect(() => {
11    const fetchCsrfToken = async () => {
12      try{
13        const response = await axios.get("http://localhost:3001/csrf-token", { withCredentials: true });
14        setCsrfToken(response.data.csrfToken);
15      }catch(err){
16        console.error("Erreur lors de la récupération du token CSRF", err);
17      }
18    };
19    fetchCsrfToken();
20  }, []);
21
22  const onSubmit = async (event) => {
23    event.preventDefault();
24    try {
25      const response = await axios.post("http://localhost:3001/auth/login", {
26        username,
27        password,
28      }, {
29        headers: {
30          "X-CSRF-Token": csrfToken
31        },
32        withCredentials: true,
33      });
34      setCookies("access_token", response.data.token);
35      window.sessionStorage.setItem("userId", response.data.userId);
36    } catch (err) {
37      console.error("Erreur lors de la connexion", err);
38    }
39  };
40 }

```

Figure 13: bout de code de auth.js

3. **Limitier les requêtes pour éviter les attaques DoS :**

Problème : L'application ne limite pas le nombre de requêtes ce qui peut engendrer une surcharge du server (un attaquant peut envoyer un grand nombre de requêtes pour faire planter l'application) ou sinon une attaque brute-force (un attaquant peut tester des milliers de combinaisons de mots de passe sur la route /login)

Solution : Limiter le nombre de requêtes par IP en utilisant express-rate-limit.

- **Ajout de la limitation globale dans index.js**

```

1 //Limite de requêtes
2 const limiter = rateLimit({
3   windowMs: 15 * 60 * 1000, // 15 minutes
4   max: 100, // limit each IP to 100 requests per windowMs
5   message: "Trop de requêtes effectuées depuis cette adresse IP, veuillez réessayer plus tard",
6   headers: true,
7 });
8 app.use(limiter);
9
10 // Routes
11 app.use("/auth", userRouter);
12 app.use("/recipes", RecipeRouter);

```

Figure 14: bout de code de index.js

- **Limiter uniquement certaines routes sensibles (auth/login)**

```
const router = express.Router();

//Limiter les tentatives de connexion:
const loginlimiter = ratelimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 5,
  message: "Trop de tentatives de connexion, veuillez réessayer plus tard",
});
```

Figure 15: bout de code de auth.js.