

---

# RAPPORT D'AUDIT

---

## Présentation du projet :

Le projet analysé est une application web nommée **RecipeApp**, qui fournit une plateforme permettant aux utilisateurs de créer, enregistrer et partager des recettes de cuisine. Cette application est conçue coté serveur avec Express.js et Node.js avec une base de donnée MySQL et repose sur React.js coté client. L'application inclut des fonctionnalités clés telles que l'authentification des utilisateurs, la gestion des recettes, et l'interaction avec des API externes pour enrichir l'expérience utilisateur. Mais comme tout système, elle nécessite une évaluation rigoureuse de ses aspects sécuritaires afin d'assurer la confidentialité, l'intégrité et la disponibilité des données manipulées.

## Méthodologie suivie pour l'audit :

Pour réaliser cet audit de sécurité, une approche systématique a été adoptée, combinant des analyses manuelles et l'utilisation d'outils d'évaluation de code. Les étapes suivantes ont été suivies :

### 1. Collecte des sources

- Analyse du code source fourni, organisé en différents fichiers (backend, configuration, client, etc.).
- Identification des modules critiques, notamment ceux liés à l'authentification, la gestion des données sensibles et les interactions avec la base de données.

### 2. Inspection manuelle du code

- Lecture détaillée du code pour repérer les pratiques à risque, telles que l'utilisation de données sensibles en clair, les requêtes SQL non paramétrées, et les entrées utilisateur non validées.
- Recherche de fonctions dangereuses pouvant introduire des vulnérabilités (par exemple, `document.write`, `eval`, etc.).

### 3. Utilisation d'outils d'analyse

- Analyse statique du code pour détecter les erreurs et incohérences courantes.
- Vérification des dépendances et bibliothèques utilisées pour identifier des versions obsolètes ou vulnérables.

### 4. Focus sur les aspects critiques

- **Gestion des cookies** : Vérification de l'utilisation des attributs `httpOnly` et `secure`.
- **JSON Web Tokens** : Inspection des données contenues dans le payload et des durées d'expiration.
- **Requêtes SQL** : Contrôle de l'utilisation de requêtes paramétrées pour prévenir les injections SQL.
- **Validation des entrées utilisateur** : Vérification de la gestion des entrées pour prévenir les failles XSS.
- **En-têtes HTTP** : Identification des en-têtes manquants et des configurations non sécurisées.

### 5. Documentation des vulnérabilités

- Chaque faille identifiée a été documentée avec sa description, son emplacement dans le code, et son impact potentiel sur la sécurité de l'application.

Cette méthodologie a permis de mener une analyse approfondie, tout en garantissant une couverture complète des différents aspects sécuritaires de l'application.

## Analyse

L'audit couvrira à la fois les aspects backend et frontend du projet. Nous commencerons par une analyse approfondie du backend avant d'aborder le frontend.

### 1. Côté back-end

L'audit du backend portera sur l'ensemble des fichiers du côté serveur, afin d'analyser et d'identifier d'éventuelles améliorations ou incohérences.

#### - **Index.js**

```
13 // Routes
14 app.use("/auth", userRouter);
15 app.use("/recipes", RecipeRouter);
16
17 // Connexion à la base de données et démarrage du serveur
18 (async () => {
19   try {
20     await sequelize.authenticate();
21     console.log("Connexion réussie à la base de données MySQL `recipe-app` !");
22
23     await sequelize.sync({ alter: true });
24     console.log("Les tables sont synchronisées avec succès.");
25
26     // Démarrage du serveur
27     app.listen(3001, () => {
28       console.log("SERVER STARTED on http://localhost:3001")
29     });
30   } catch (error) {
31     console.error("Erreur lors de la connexion à la base de données :", error);
32     process.exit(1);
33   }
34 }
35 )();
```

Figure 1: Bout de code du fichier index.js

#### Les failles détectées

##### 1. Manque de validation des entrées utilisateur

Les requêtes entrantes ne semblent pas être validées, laissant potentiellement place à des attaques (XSS, injection SQL, etc.).

##### 2. Absence d'en-têtes de sécurité

Les en-têtes HTTP pour protéger contre des attaques comme le clickjacking, XSS ou l'extraction de contenu ne sont pas configurés.

## - db.js

```
1 import { Sequelize } from "sequelize";
2
3 const sequelize = new Sequelize("recipe_app", "root", "", {
4   host: "localhost",
5   dialect: "mysql",
6 });
7
8 export default sequelize;
```

Figure 2: Bout de code du fichier db.js

### Les failles détectées

#### 1. Identifiants de base de données exposés dans le code source

Les identifiants de la base de données (nom de la base, utilisateur, mot de passe) sont directement codés en dur. Cela peut entraîner des risques en cas de fuite du code source.

## - USER.js

```
4 const User = sequelize.define("User", {
5   username: {
6     type: DataTypes.STRING,
7     allowNull: false, // Équivalent de `required: true`
8     unique: true, // Username doit être unique
9   },
10  savedRecipes: {
11    type: DataTypes.STRING,
12  },
13  password: {
14    type: DataTypes.STRING,
15    allowNull: false, // Équivalent de `required: true`
16  },
17 });
18
19 export default User;
```

Figure 3: Bout de code du fichier USER.js

### Les failles détectées

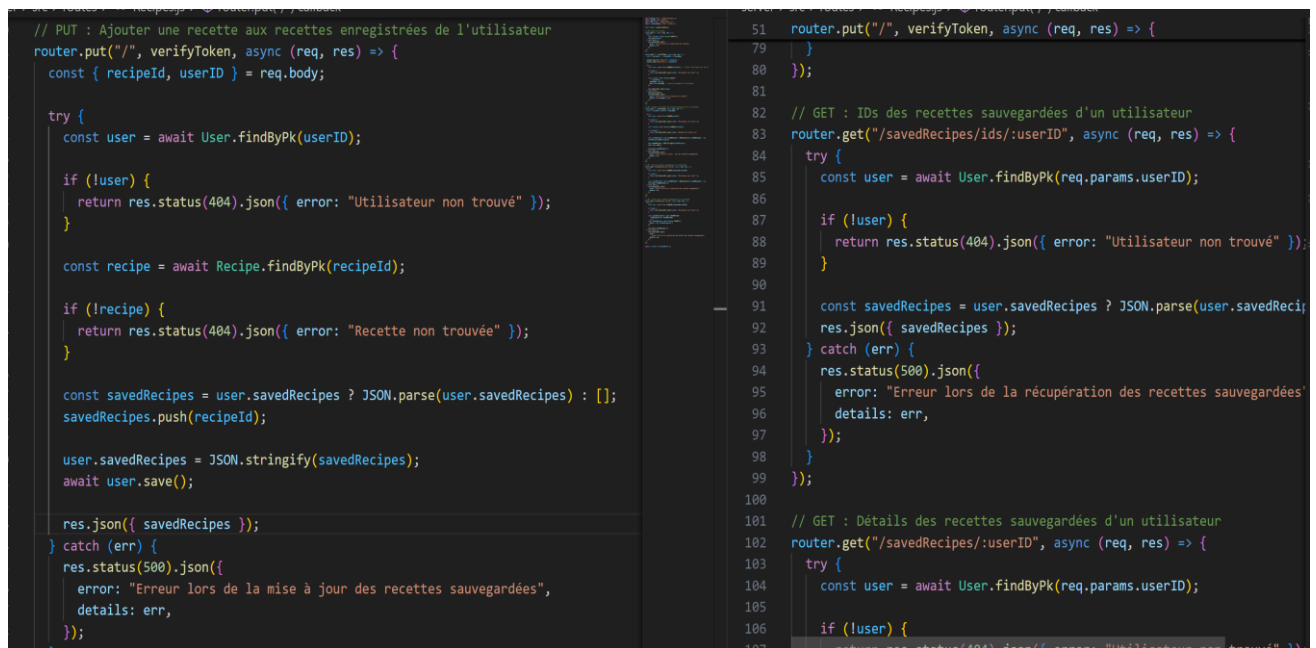
#### 1. Absence de validation des caractères autorisés

Un utilisateur pourrait soumettre des noms d'utilisateur contenant des caractères malveillants (par exemple, des scripts) qui pourraient être exploités pour des attaques XSS.

#### 2. Absence du cryptage du mot de passe

Le champ password est défini comme une chaîne de caractères sans mécanisme pour stocker les mots de passe de manière sécurisée. Les mots de passe sont donc potentiellement stockés en clair dans la base de données.

## - RecipeRoute.js



```
// PUT : Ajouter une recette aux recettes enregistrées de l'utilisateur
router.put("/", verifyToken, async (req, res) => {
  const { recipeId, userID } = req.body;

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

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

    const recipe = await Recipe.findByPk(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();

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

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

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

    const savedRecipes = user.savedRecipes ? JSON.parse(user.savedRecipes) : [];
    res.json({ savedRecipes });
  } catch (err) {
    res.status(500).json({
      error: "Erreur lors de la récupération des recettes sauvegardées",
      details: err,
    });
  }
});

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

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

    const savedRecipes = user.savedRecipes ? JSON.parse(user.savedRecipes) : [];
    // ... (code pour récupérer les détails des recettes) ...
  } catch (err) {
    res.status(500).json({
      error: "Erreur lors de la récupération des détails des recettes sauvegardées",
      details: err,
    });
  }
});
```

Figure 4: Bout de code du fichier RECIPEROUTE.js

Les failles détectées :

### 1. Absence de validation des données en entrée

Aucune validation ou désinfection des données envoyées par les utilisateurs via req.body ou req.params n'est effectuée. Cela expose les routes aux attaques suivantes :

- Injection SQL : Des données malveillantes pourraient être injectées dans les requêtes SQL.
- XSS (Cross-Site Scripting) : Les données non vérifiées pourraient être utilisées dans des réponses HTTP ou stockées en base.

### 2. Manque de contrôle d'accès dans certaines routes

Certaines routes sensibles, comme PUT et les routes /savedRecipes, ne vérifient pas si l'utilisateur authentifié (via verifyToken) est bien le propriétaire des données qu'il modifie.

### 3. Absence de protection contre les attaques par déni de service (DoS)

Certaines routes (comme GET /) permettent de récupérer toutes les recettes sans pagination, ce qui peut entraîner des problèmes de performance en cas de forte charge.

### 4. Mauvaise gestion des tokens dans verifyToken

Le middleware verifyToken semble être utilisé pour sécuriser certaines routes, mais aucune logique de vérification stricte n'est décrite ici. Si ce middleware n'est pas correctement implémenté, toutes les routes pourraient être exposées.

## - **UserRoute.js**

```
8 router.post("/register", async (req, res) => {
9   const { username, password } = req.body;
10  const user = await User.findOne({ where: { username: username } });
11  console.log(user);
12
13  if (user) {
14    return res.json({ message: "User already exists !" });
15  }
16
17  const hashedPassword = await bcrypt.hash(password, 10);
18  const newUser = new User({
19    username: username,
20    password: hashedPassword,
21  });
22  await newUser.save();
23  res.json({ message: "User Registered Successfully !" });
24 });
25
26 router.post("/login", async (req, res) => {
27   const { username, password } = req.body;
28   const user = await User.findOne({ where: { username: username } });
29
30   if (!user) {
31     return res.json({ message: "User Doesn't Exist !" });
32   }
33
34   const IsPasswordValid = await bcrypt.compare(password, user.password);
35
36   if (!IsPasswordValid) {
37     return res.json({ message: "Username or Password is Incorrect" });
38   }
39   const token = jwt.sign({ id: user._id }, "secret");
40   res.json({ token, userId: user.id });
41 });
42
43 export { router as userRouter };
44
45 export const verifyToken = (req, res, next) => {
46   const token = req.headers.authorization;
47
48   if (token) {
49     jwt.verify(token, "secret", (err) => {
50       if (err) return res.sendStatus(403);
51       next();
52     });
53   } else {
54     res.sendStatus(401);
55   }
56 }
```

Figure 5: Bout de code du fichier userROUTE.js

### Les failles détectées

#### 1. Mauvaise gestion des mots de passe

Bien que bcrypt soit utilisé, aucune validation ou désinfection des mots de passe n'est effectuée avant leur hachage. Cela peut permettre l'injection de caractères malveillants.

Aucune restriction n'est imposée sur la complexité ou la longueur des mots de passe.

#### 2. Utilisation d'une clé secrète statique dans JWT

La clé secrète "secret" est codée en dur et n'est pas sécurisée. Si elle est compromise, un attaquant pourrait générer des tokens valides.

Aucun mécanisme de rotation ou expiration des tokens n'est mis en place.

#### 3. Vérification insuffisante des tokens

La fonction verifyToken n'effectue pas de validation approfondie.

Aucun contrôle sur l'identité de l'utilisateur (ex. : un token valide mais volé pourrait être utilisé).

#### 4. Absence de protection contre les attaques CSRF

Bien que CSRF soit moins problématique avec des tokens JWT utilisés dans les en-têtes, des attaques restent possibles si le token est exposé ou mal géré.

## 2. Coté front-end

Après avoir analysé le coté serveur de l'application, il est nécessaire pour compléter l'audit de faire une analyse coté client.

## - **Auth.js**

### Les failles détectées

#### 1. Stockage du token JWT dans les cookies

Le token JWT est stocké dans un cookie, mais rien n'indique que le cookie est sécurisé (HttpOnly, Secure, etc.).

Sans ces options, le token peut être exposé à des attaques XSS (Cross-Site Scripting).

## 2. Utilisation de localStorage pour l'ID utilisateur

Les données stockées dans localStorage sont accessibles via JavaScript, ce qui les rend vulnérables aux attaques XSS.

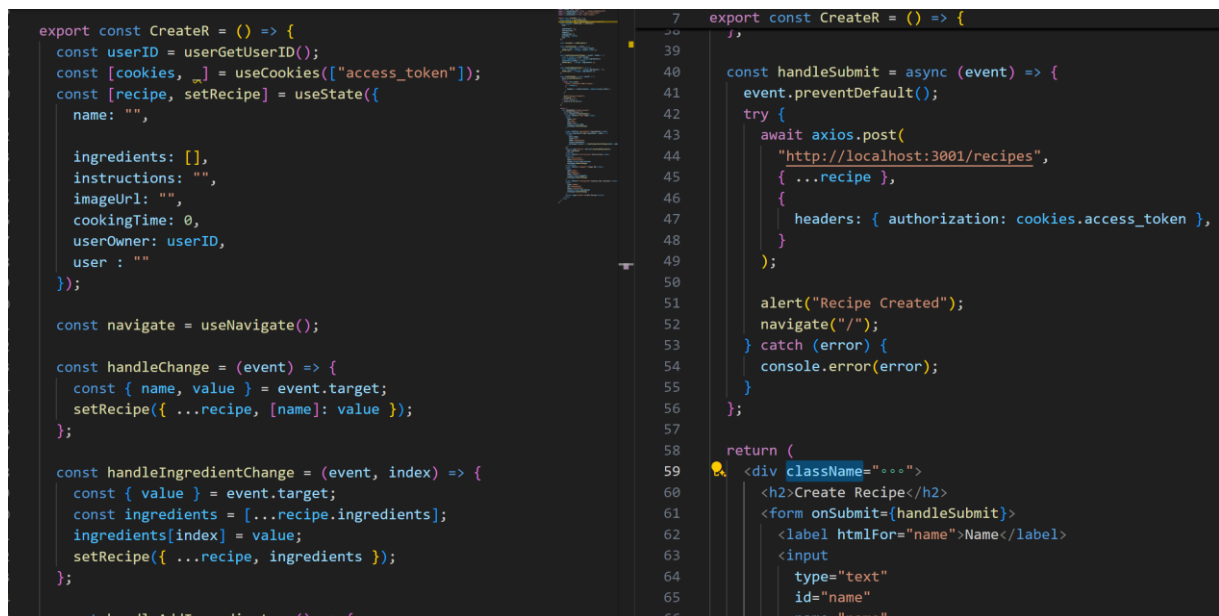
## 3. Manque de validation des entrées utilisateur

Aucune validation ou encodage n'est effectué sur les entrées utilisateur (username et password). Cela pourrait permettre des attaques par injection.

## 4. Manque de mesures contre les attaques CSRF

Aucun mécanisme de protection contre les attaques CSRF (Cross-Site Request Forgery) n'est mentionné.

### - Create-recipe.js



```
export const CreateR = () => {
  const userID = userGetUserID();
  const [cookies, _] = useCookies(["access_token"]);
  const [recipe, setRecipe] = useState({
    name: "",
    ingredients: [],
    instructions: "",
    imageUrl: "",
    cookingTime: 0,
    userOwner: userID,
    user: ""
  });

  const navigate = useNavigate();

  const handleChange = (event) => {
    const { name, value } = event.target;
    setRecipe({ ...recipe, [name]: value });
  };

  const handleIngredientChange = (event, index) => {
    const { value } = event.target;
    const ingredients = [...recipe.ingredients];
    ingredients[index] = value;
    setRecipe({ ...recipe, ingredients });
  };

  const handleAddIngredient = () => {
    const newIngredient = prompt("Enter ingredient name");
    if (newIngredient) {
      setRecipe({
        ...recipe,
        ingredients: [...recipe.ingredients, newIngredient]
      });
    }
  };

  const handleSubmit = async (event) => {
    event.preventDefault();
    try {
      await axios.post(
        "http://localhost:3001/recipes",
        { ...recipe },
        { headers: { authorization: cookies.access_token } }
      );
      alert("Recipe Created");
      navigate("/");
    } catch (error) {
      console.error(error);
    }
  };

  return (
    <div className="container">
      <h2>Create Recipe</h2>
      <form onSubmit={handleSubmit}>
        <label htmlFor="name">Name</label>
        <input
          type="text"
          id="name"
          name="name"
        />
      </form>
    </div>
  );
};
```

Figure 6: Bout de code du fichier CreateRecipe.js

## 1. Manque de vérification côté serveur du jeton JWT

L'application suppose que le jeton JWT envoyé dans l'en-tête authorization est valide. Si le backend ne valide pas correctement ce jeton, un utilisateur malveillant pourrait soumettre de fausses requêtes.

### - UseGetUserID.js (hook)

```
1 export const userGetUserID = () => {
2   console.log(window.localStorage.getItem("userID"));
3   return window.localStorage.getItem("userID");
4 };
5
```

Figure 7: Bout de code du fichier UseGetUserId.js

## **1. Risque d'exposition d'informations sensibles**

Le hook récupère directement l'userID à partir du localStorage, ce qui est une méthode de stockage accessible à tout script exécuté sur la page (y compris des scripts injectés malveillants via XSS).