Secured WebShop

Dario Chasi – CID2A
Vennes
32 périodes
M.Sonney

Table des matières

1 SPÉCIFICATIONS

Web Store

Création d'un site d'e-commerce sécurisé

Description

Au terme du projet, l'apprenti sera capable de construire une application node.js offrant un accès sécurisé et une gestion des rôles.

Matériel et logiciels à disposition

- Un ordinateur standard de la section informatique avec Docker Desktop

Prérequis

Modules 294, 335

Cahier des charges

1.1.1 Dockerisation

L'ensemble des services web sera conteneurisé.

1.1.2 Profil du client

Le client peut accéder à son propre profil en utilisant un lien tel que : https://localhost/users/john

Seul son profil lui sera rendu visible.

1.1.3 HTTPS

Il doit être possible d'accéder à votre site de e-commerce de manière sécurisée (https://localhost). Le port utilisé sera le 443.Le certificat sera auto-signé par OpenSSL.

1.1.4 Authentification par mot de passe

L'utilisateur devra s'authentifier par la page https://localhost/login

Le mot de passe sera hashé et salé avant d'être stocké dans la base de données (tablet_users)

1.1.5 Administration

Une page d'administration devra avoir un champ de recherche (Nom du visiteur) et permettre d'afficher tous les utilisateurs ayant tout ou partie de ce nom.

1.1.6 Protection contre les injections SQL

Votre page d'administration devra être protégée contre les injections SQL.

Sans utiliser sequelize ou tout autre ORM, votre site devra être robuste face au

1.1.7 Versioning

Votre code sera versionné sur Github et un .gitignore empêchera de versionner les binaires npm (dossiers node_modules). Votre dépôt sera partagé avec votre chef de projet.

Les points suivants seront évalués

- Le rapport
- Le journal de travail
- Une conclusion sur le travail fourni et sur l'attitude face au projet

Validation et conditions de réussite

- Compréhension du travail
- Possibilité de transmettre le travail à une personne extérieure pour le terminer, le corriger ou le compléter
- Etat de fonctionnement du produit livré

2 ANALYSE

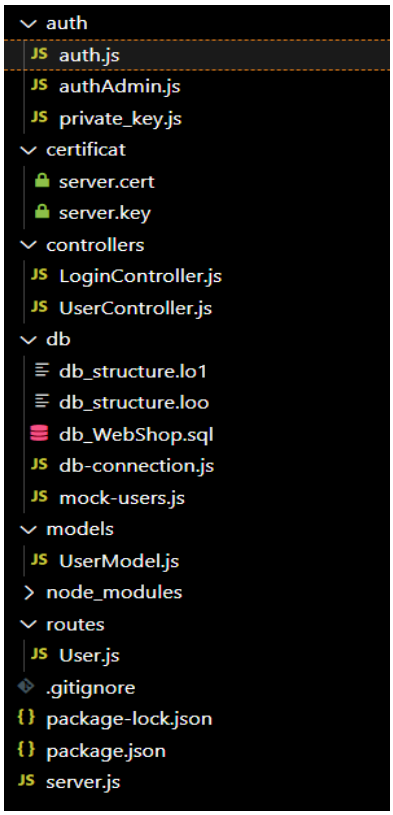
Document d'analyse et conception

Pour ce projet, un dossier zip nous est fourni avec des conteneurs Docker, ainsi que la restriction de ne pas utiliser Sequelize ni bcrypt. Notre site doit être accessible en utilisant HTTPS. Nous devons créer un site web protégé contre les injections SQL et gérer la création des rôles.

3 RÉALISATION

Structure du code

Voici la structure de mon code pour le bon fonctionnement de mon application, en séparant les fichiers concernant l'authentification, les certificats, la base de données, les contrôleurs, les modèles et les routes dans différents répertoires :



Installation de packages

Le zip de base reçu par l'enseignant manqué de certain packages, j'ai commencé par installer :

- jsonwebtoken
- crypto
- cors
- mysql2
- http-server

Authentification

3.1.1 Vérification du token

Pour cette première authentification, j'ai créé un système qui vérifie l'authenticité du token à l'aide d'une clé privée, qui, dans un cas réel, devrait être stockée ailleurs.

```
const jwt = require("jsonwebtoken");
const { privateKey } = require("./private_key.js");

module.exports.auth = (req, res, next) => {
  const authorizationHeader = req.headers.authorization;

  if (!authorizationHeader) {
    const message =
      "Vous n'avez par fourni de jeton d'authentification. Ajoutez-en un dans l'en tête de la requête.";
    return res.status(401).json({ message });
  } else {
    const token = authorizationHeader.split(" ")[1];
    const decodedToken = jwt.verify(
      token,
      privateKey,
      (error, decodedToken) => {
        if (error) {
          const message =
            "L'utilisateur n'est pas autorisé à accéder à cette resource.";
          return res.status(401).json({ message, data: error });
        }
        const userId = decodedToken.userId;
        if (req.body.userId && req.body.userId !== userId) {
          const message = "L'indentifiant de l'utilisateur est invalide";
          return res.status(401).json({ message });
        } else {
          next();
        }
      }
    );
  }
};
```

3.1.2 Authentification Admin

Pour la deuxième authentification l'utilisateur doit être admin pour y accéder au différents ressources.

```
const jwt = require("jsonwebtoken");
const { privateKey } = require("./private_key.js");

module.exports.authAdmin = async (req, res, next) => {
  const authorizationHeader = req.headers.authorization;

  if (!authorizationHeader) {
    const message = `Vous n'avez pas fourni de jeton d'authentification. Ajoutez-en un dans l'en-tête de la requête.`;
    return res.status(401).json({ message });
  }

  const token = authorizationHeader.split(" ")[1];

  try {
    const decodedToken = jwt.verify(token, privateKey);
    const isAdmin = decodedToken.isAdmin;

    // Vérifie si l'utilisateur est admin
    if (!isAdmin) {
      const message = `L'accès est restreint aux administrateurs uniquement.`;
      return res.status(403).json({ message });
    }
    next();
  } catch (error) {
    const message = `L'utilisateur n'est pas autorisé à accéder à cette ressource.`;
    return res.status(401).json({ message, data: error });
  }
};
```

Https

Avec OpenSSL préinstallé sur le PC de l'école, j'ai utilisé la commande suivante pour obtenir les certificats autosignés :
openssl req -nodes -new -x509 -keyout server.key -out server.cert

Une fois cette étape complétée, j'ai ajouté les certificats à mon projet et configuré mon server.js pour son bon fonctionnement :

```
const express = require("express");
const fs = require("fs");
const https = require("https");
const cors = require("cors");
const app = express();
app.use(express.json());

const userRoute = require("./routes/User");

var corsOptions = {
  origin: `http://localhost:5173`,
  optionsSuccessStatus: 200, // For legacy browser support
};
app.use(cors(corsOptions));

app.use("/users", userRoute);
app.use("/login", userRoute);

const options = {
  key: fs.readFileSync("certificat/server.key"),
  cert: fs.readFileSync("certificat/server.cert"),
};

https.createServer(options, app).listen(443, () => {
  console.log("Server is running on https://localhost:443");
});
```

Comme https et fs sont déjà présents sur node, j'ai ne pas eu besoin d'installer aucun package supplémentaire.

Modèle

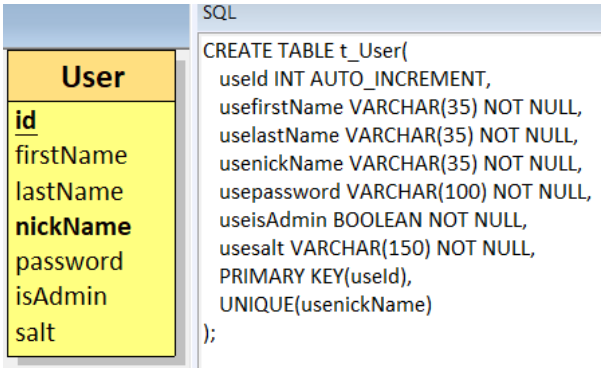
J'ai ainsi créé un modèle pour mes utilisateurs, qui empêche l'utilisation de caractères spéciaux, l'insertion de valeurs nulles, impose une longueur maximale de caractères et une longueur minimale pour le mot de passe :

```
const validateUser = (user) => {
  const errors = [];
  const regex = /^[A-Za-z0-9\s]+$/;

  if (!user.firstName || user.firstName.trim().length === 0) {
    errors.push("Le prénom de l'utilisateur ne peut pas être vide.");
  } else {
    if (user.firstName.length > 35) {
      errors.push(
        "Le prénom de l'utilisateur doit contenir au maximum 35 caractères."
      );
    }
    if (!regex.test(user.firstName)) {
      errors.push(
        "Seules les lettres, les chiffres et les espaces sont autorisées pour le prénom."
      );
    }
  }

  if (!user.lastName || user.lastName.trim().length === 0) {
    errors.push("Le nom de l'utilisateur ne peut pas être vide.");
  } else {
    if (user.lastName.length > 35) {
      errors.push(
        "Le nom de l'utilisateur doit contenir au maximum 35 caractères."
      );
    }
    if (!regex.test(user.lastName)) {
      errors.push(
        "Seules les lettres, les chiffres et les espaces sont autorisées pour le nom."
      );
    }
  }

  if (!user.nickName || user.nickName.trim().length === 0) {
    errors.push("Le nom d'utilisateur ne peut pas être vide.");
  } else {
    if (user.nickName.length > 35) {
      errors.push(
        "Le nom d'utilisateur doit contenir au maximum 35 caractères."
      );
    }
    if (!regex.test(user.nickName)) {
      errors.push(
        "Seules les lettres, les chiffres et les espaces sont autorisées pour le nom d'utilisateur."
      );
    }
  }

  if (!user.password || user.password.trim().length === 0) {
    errors.push("Le mot de passe ne peut pas être vide.");
  } else {
    if (user.password.length < 4) {
      errors.push("Le mot de passe doit contenir au moins 4 caractères.");
    }
  }

  if (typeof user.isAdmin !== "boolean") {
    errors.push("Le rôle d'administrateur doit être un booléen.");
  }

  return errors;
};

module.exports = { validateUser };
```

```
let users = [
  {
    id: 1,
    firstName: "John",
    lastName: "Doe",
    nickName: "JohnDoe",
    password: "1234",
    isAdmin: true,
  },
  {
    id: 2,
    firstName: "Jane",
    lastName: "Doe",
    nickName: "JaneDoe",
    password: "4321",
    isAdmin: true,
  },
  {
    id: 3,
    firstName: "Nima",
    lastName: "Zarrabi",
    nickName: "ScoobyDoo",
    password: "BabyGirl",
    isAdmin: false,
  },
];

module.exports = users;
```

3.1.3 *Table users*

Table appartenant à ce modèle



Insertion base de données

3.1.4 Connexion à la base de données

Pour faire la connexion à ma base des données j'ai créé un pool avec les différentes informations de ma base des données.

```
const pool = mysql.createPool({
  host: "localhost",
  user: "root",
  password: "root",
  database: "db_WebShop",
  port: 6033,
});
```

3.1.5 Hachage de mot de passe

Pour hasher le mot de passé, j'ai créé un salt avec le nickname pour le rendre unique, puis j'ai le rajouté au hachage de mon mot de passe.

```
function generateSalt(nickName) {
  return crypto.createHash("sha256").update(nickName).digest("hex");
}

async function hashPassword(password, salt) {
  return crypto
    .createHash("sha256")
    .update(password + salt)
    .digest("hex");
}
```

3.1.6 *Fonction insertUsers*

Ensuite pour l'insertion correcte des utilisateurs présents dans mon mock des utilisateurs, je me connecte à ma base de données, puis je vérifie si les données insérées correspondent à mon modèle. Je m'assure également de hasher le mot de passe avant de l'insérer dans ma base de données avec le salt correspondant.

```
async function insertUsers(users) {
  const conn = await pool.getConnection();
  try {
    await conn.beginTransaction();

    for (const user of users) {
      const validationErrors = validateUser(user);
      if (validationErrors.length > 0) {
        console.error("User validation failed:", validationErrors);
        throw new Error("User validation failed");
      }

      const { firstName, lastName, nickName, password, isAdmin } = user;
      if (await userExists(nickName)) {
        console.log(`User ${nickName} already exists. Skipping insertion.`);
        continue;
      }

      const salt = generateSalt(nickName);
      const hashedPassword = await hashPassword(password, salt);

      const sql = `INSERT INTO t_User (usefirstName, uselastName, usenickName, usepassword, usesalt, useisAdmin) VALUES (?, ?, ?, ?, ?, ?)`;
      await conn.execute(sql, [
        firstName,
        lastName,
        nickName,
        hashedPassword,
        salt,
        isAdmin,
      ]);
    }

    await conn.commit();
    console.log("Users have been successfully inserted.");
  } catch (error) {
    console.error("Error inserting users:", error);
    await conn.rollback();
  } finally {
    conn.release();
  }
}
```

Ensuite, j'utilise cette fonctionne pour insérer mes utilisateurs.

```
const users = require("./mock-users");

insertUsers(users).catch((error) => {
  console.error("Failed to insert users:", error);
});
```

Contrôleurs

3.1.7 Login

Pour mon contrôleur de login, l'utilisateur doit entrer son pseudo et son mot de passe. Une fois cela fait, j'utilise le salt sauvegardé au préalable et je compare les hashes.

```
async function hashPassword(password, salt) {
  return crypto
    .createHash("sha256")
    .update(password + salt)
    .digest("hex");
}

module.exports.login = async (req, res) => {
  try {
    console.log("Request body:", req.body);

    const [rows] = await pool.execute(
      "SELECT * FROM t_User WHERE usenickName = ?",
      [req.body.nickName]
    );
    const user = rows[0];

    console.log("User found:", user);

    if (!user) {
      const message = "L'utilisateur demandé n'existe pas";
      return res.status(404).json({ message });
    }

    const salt = user.usesalt;
    const hashedPassword = await hashPassword(req.body.password, salt);

    console.log("Hashed password:", hashedPassword);
    console.log("User stored password:", user.usepassword);

    if (hashedPassword !== user.usepassword) {
      const message = "Le mot de passe est incorrecte.";
      return res.status(401).json({ message });
    } else {
      const token = jwt.sign(
        { userId: user.id, isAdmin: user.useisAdmin },
        privateKey,
        {
          expiresIn: "1y",
        }
      );
      const message = "L'utilisateur a été connecté avec succès";
      return res.json({ message, data: user, token });
    }
  } catch (error) {
    console.error("Error during login:", error);
    const message =
      "L'utilisateur n'a pas pu être connecté. Réessayez dans quelques instants";
    return res.status(500).json({ message, data: error });
  }
};
```

3.1.8 Users All

Contrôleur pour l'obtention de tous les utilisateurs avec des requêtes préparées, un message affiché aux utilisateurs et un autre message affiché dans la console.

```
module.exports.getAll = async (req, res) => {
  try {
    const [users] = await pool.query("SELECT * FROM t_User");
    res.json({ data: users });
  } catch (error) {
    const message = "Failed to retrieve users:";
    console.error({ msg: message, data: error });
    const Publicmessage = "Internal Server Error";
    res.status(500).send({ msg: Publicmessage });
  }
};
```

3.1.9 Users nickname

Contrôleur pour l'obtention des utilisateurs par rapport à son nickname avec des requêtes préparées, un message affiché aux utilisateurs et un autre message affiché dans la console.

```
module.exports.getUserByNickname = async (req, res) => {
  try {
    const nickname = req.params.nickname;
    const [users] = await pool.query(
      "SELECT * FROM t_User WHERE usenickName LIKE ?",
      [`%${nickname}%`]
    );
    if (users.length === 0) {
      const message = "User not found";
      return res.status(404).send({ msg: message });
    }
    return res.json({ data: users });
  } catch (error) {
    const message = "Failed to retrieve user by nickname:";
    console.error({ msg: message, data: error });
    const Publicmessage = "Internal Server Error";
    res.status(500).send({ msg: Publicmessage });
  }
};
```

3.1.10 AddUser

La route addUser est la même que pour l'insertion automatique des utilisateur présents dans le mock

```
module.exports.addUser = async (req, res) => {
  const user = req.body;

  const errors = validateUser(user);
  if (errors.length > 0) {
    throw new Error(errors.join(", "));
  }

  const sql = `INSERT INTO t_User (usefirstName, uselastName, usenickName, usepassword, usesalt, useisAdmin) VALUES (?, ?, ?, ?, ?, ?)`;

  let conn;
  try {
    conn = await pool.getConnection();
    await conn.beginTransaction();

    const salt = generateSalt(user.nickName);
    const hashedPassword = await hashPassword(user.password, salt);

    const [result] = await conn.query(sql, [
      user.firstName,
      user.lastName,
      user.nickName,
      hashedPassword,
      salt,
      user.isAdmin,
    ]);

    await conn.commit();
    const message = "User successfully added";
    return res.status(201).json({ msg: message, data: result });
  } catch (error) {
    if (conn) {
      await conn.rollback();
    }
    const message = "Failed to insert user: ";
    return res.status(500).json({ msg: message, data: error.message });
  } finally {
    if (conn) {
      conn.release();
    }
  }
};
```

Routes

Ensuite l'utilisation des différents moyens d'authentification dépendant des routes

```
router.get("/", authAdmin, controller.getAll);

router.get("/:nickname", auth, controller.getUserByNickname);
router.post("/", controllerLog.login);

router.post("/addUser", authAdmin, controller.addUser);

module.exports = router;
```

FrontEnd

Création du frontend avec vue.js. (npm run dev pour l'activer)
L'utilisateur doit entrer ses identifiants pour effectuer une recherche des utilisateurs et doit être administrateur pour afficher tous les utilisateurs.

Home

Nickname: Enter nickname Password: Enter password Login

Search Users: Enter nickname Submit

Get All Users

4 CONCLUSION

Bilan des fonctionnalités demandées

L'utilisateur peut accéder à son compte avec son pseudo. Le rôle administrateur a été créé ainsi que les certificats pour le HTTPS. Pour les contrôleurs, des requêtes préparées ont été mises en place, et l'authentification a été ajoutée aux routes. Des limitations de caractères et des restrictions sur les caractères spéciaux ont été implémentées pour éviter les injections SQL. Le mot de passe est hashé avec un salt. Des vues en Vue.js ont été créées pour la barre de recherche, la connexion et l'affichage de tous les utilisateurs.

Bilan personnel

Pour ce projet, j'aurais souhaité recevoir un backend plus avancé afin de pouvoir expérimenter plus sur les autres moyens de sécurité. J'ai apprécié les restrictions, telles que l'interdiction d'utiliser Sequelize ou bcrypt, car elles m'ont permis d'apprendre de nouvelles méthodes de création d'un backend, avec les contrôleurs et les routes séparés chose que nous n'avons pas vu durant le module backend. Cependant, ce choix ne m'a pas permis de progresser à la vitesse souhaitée.

5 DIVERS

Webographie

OpenSSL - CrypTool
Form Data Validation in Node.js with express-validator (stackabuse.com)
Using express-validator for data validation in NodeJS | by Chau Nguyen | Medium
https://www.cockroachlabs.com/docs/stable/create-security-certificates-openssl

6 ANNEXES

Code

Journal de travail

Présentation