**Création d’un site d’e-commerce sécurisé**



William Trelles – Classe CID 2A

ETML-Lausanne-Vennes

Professeur : Gaël Sonney

Table de matière

[1 Objectif du projet 2](#_Toc167872205)

[2 Dockerisation 3](#_Toc167872206)

[3 Conceptualisation (schéma) 3](#_Toc167872207)

[4 HTTPS 3](#_Toc167872208)

[5 Authentication 4](#_Toc167872209)

[6 Protection contre les injections SQL 6](#_Toc167872210)

[7 GitHub 6](#_Toc167872211)

[8 Code 6](#_Toc167872212)

[9 Conclusion 6](#_Toc167872213)

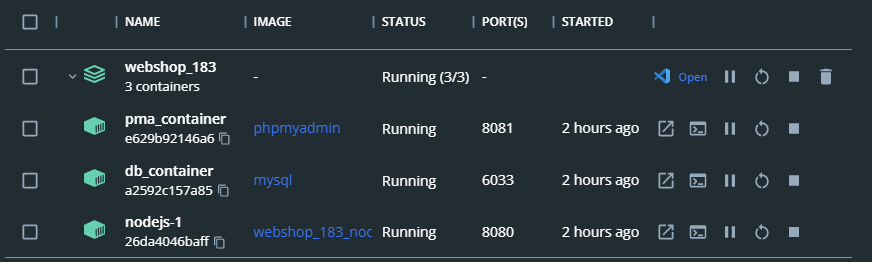
# Objectif du projet

Dans ce projet, nous allons plonger dans le monde de Node.js pour construire une application web robuste. Notre objectif principal est de créer une application qui offre un accès sécurisé aux utilisateurs et une gestion des rôles.

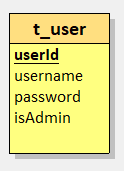
Tout d'abord, nous allons explorer les bases de Node.js. Cela comprend la configuration d'un serveur web et la mise en place de routes pour gérer les différentes demandes des utilisateurs.

# Dockerisation

Pour ce projet j’utilise 3 containers qui sont les suivants :



# Conceptualisation (schéma)



Dans le but de conceptualiser la base de données j’ai créé la table suivante t\_user, qui contient un userId, un username, mot de passe et un booléen.

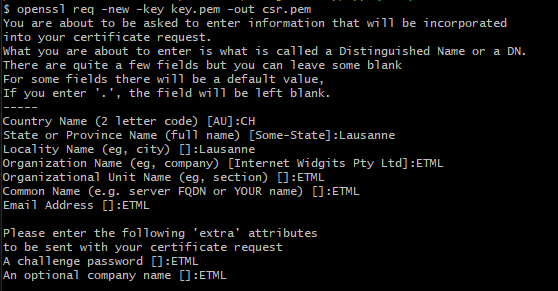
userId : Identifiant unique de l'utilisateur

username : Nom d'utilisateur pour la connexion.

Mot de passe : Stocké de manière sécurisée (haché et salé).

Rôle administrateur : Booléen indiquant les privilèges d'administration.

# HTTPS



Commandes utilisées pour la génération du certificat avec OpenSSL.

**openssl genrsa -out key.pem**

Cette commande génère une paire de clés RSA. La clé privée est stockée dans le fichier key.pem. Cette clé privée sera utilisée pour signer le certificat et pour établir des connexions sécurisées.

**openssl req -new -key key.pem -out csr.pem**

Cette commande crée une demande de signature de certificat en utilisant la clé privée générée précédemment. Le csr contient des informations sur l'entité qui nous seront demandés pour la création du certificat. Le CSR est enregistré dans le fichier csr.pem.

**openssl x509 -req -days 9999 -in csr.pem -signkey key.pem -out cert.pem**

Cette commande signe la demande de certificat à l'aide de la clé privée pour générer un certificat **auto-signé**. Le certificat est valable pour 9999 jours (-days 9999). Le certificat signé est enregistré dans le fichier cert.pem.

**rm csr.pem**

Cette commande supprime le fichier de la demande de signature de certificat puisqu'il n'est plus nécessaire une fois que le certificat a été généré.

Après ce processus nous pourrons accéder à l’application en utilisant https :

<https://localhost/>

# Authentication

import jwt from "jsonwebtoken";

import { privateKey } from "./private\_key.mjs";

import { User } from "../db/sequelize.mjs"; // Importa el modelo de usuario desde tu archivo sequelize

const auth = 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 });

  } else {

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

    try {

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

      req.userId = decodedToken.userId;

      //Gestion de rôle admin

      const user = await User.findByPk(req.userId);

      if (!user || !user.isAdmin) {

        const message = `L'utilisateur n'est pas autorisé à accéder à cette ressource.`;

        return res.status(401).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 });

    }

  }

};

export { auth };

1. Vérification d'un jeton JWT dans l'en-tête de la requête.
2. Si un jeton est trouvé, il le vérifie en utilisant une clé privée.
3. Si le jeton est valide, l'ID de l'utilisateur est extrait et stocké dans la requête.
4. Ensuite, il vérifie si l'utilisateur est un administrateur.
5. Si l'utilisateur est authentifié et est un administrateur, il passe à la prochaine étape.
6. Sinon, il renvoie une erreur d'authentification.

# GitHub

<https://github.com/witrellesp/P_APP_183-Secured-WebShop-William.git>

# Code

**SERVEUR.MJS**

import express from "express";

import https from "https";

import fs from "fs";

import { initDb, sequelize } from "./db/sequelize.mjs";

// Création de l'application Express

const app = express();

// Configuration des options HTTPS avec les fichiers de clé et de certificat

const options = {

  key: fs.readFileSync("key.pem"),

  cert: fs.readFileSync("cert.pem"),

};

// Définition d'une route pour la page d'accueil

app.get("/", (req, res) => {

  res.send("Home Page");

});

// Authentification de la base de données

sequelize

  .authenticate()

  .then(() => {

    console.log("La connexion à la base de données a bien été établie");

  })

  .catch((error) => {

    console.error("Impossible de se connecter à la DB", error);

  });

// Initialisation db

initDb();

// Importation et utilisation des routes des utilisateurs

import { userRouter } from "./routes/User.mjs";

app.use("/users", userRouter);

// Importation et utilisation de la route de connexion

import { loginRouter } from "./routes/login.mjs";

app.use("/login", loginRouter);

// Gestion des erreurs 404

app.use(({ res }) => {

  const message =

    "Impossible de trouver la ressource demandée ! Vous pouvez essayer une autre URL.";

  res.status(404).json(message);

});

// Création et démarrage du serveur HTTPS

https.createServer(options, app).listen(443, () => {

  console.log("Server running on port 443");

});

LOGIN.MJS

import express from "express";

import bcrypt from "bcrypt";

import jwt from "jsonwebtoken";

import { User } from "../db/sequelize.mjs";

import { privateKey } from "../auth/private\_key.mjs";

// Création du routeur

const loginRouter = express();

// Traiter les requêtes au format JSON

loginRouter.use(express.json());

// Définition de la route POST pour la connexion

loginRouter.post("/", (req, res) => {

// Recherche de l'utilisateur dans la base de données en fonction du nom d'utilisateur

  User.findOne({ where: { username: req.body.username } })

    .then((user) => {

      if (!user) {

        // Si l'utilisateur n'est pas trouvé, renvoyer un message d'erreur 404

        const message = `L'utilisateur demandé n'existe pas`;

        return res.status(404).json({ message });

      }

// Comparaison du mot de passe fourni avec le mot de passe haché stocké dans la base de données

      bcrypt

        .compare(req.body.password, user.password)

        .then((isPasswordValid) => {

          if (!isPasswordValid) {

    // Si le mot de passe est incorrect, renvoyer un message d'erreur 401

            const message = `Le mot de passe est incorrect.`;

            return res.status(401).json({ message });

          } else {

// Si le mot de passe est valide, générer un jeton JWT pour l'authentification

            const token = jwt.sign({ userId: user.id }, privateKey, {

              expiresIn: "1y",

            });

            // Renvoyer un message et les données utilisateur et le jeton JWT

            const message = `L'utilisateur a été connecté avec succès`;

            return res.json({ message, data: user, token });

          }

        });

    })

    .catch((error) => {

      // En cas d'erreur lors de la connexion de l'utilisateur, renvoyer un message d'erreur avec les détails de l'erreur

      const message = `L'utilisateur n'a pas pu être connecté. Réessayez dans quelques instants`;

      return res.json({ message, data: error });

    });

});

export { loginRouter };

Une fois que l’utilisateur est logé, le token généré pourra être utilisé pour la navigation entre les différentes routes.

SEQUELIZE.MJS

import { DataTypes, Sequelize } from "sequelize";

import { UserModel } from "../models/user.mjs";

import users from "./users-mock.mjs"; // Import des données utilisateur fictives

import bcrypt from "bcrypt";

// Création d'une instance Sequelize pour se connecter à la base de données

const sequelize = new Sequelize(

  "db\_web", // Nom de la DB qui doit exister

  "root", // Nom de l'utilisateur

  "root", // Mot de passe de l'utilisateur

  {

    host: "localhost",

    port: "6033",

    dialect: "mysql",

    logging: false,

  }

);

// Création du modèle d'utilisateur en utilisant le modèle importé et les types de données Sequelize

const User = UserModel(sequelize, DataTypes);

// Fonction pour initialiser la base de données

let initDb = async () => {

  try {

    await sequelize.sync({ force: true }); // Synchronisation avec la base de données, force:true supprime et recrée toutes les tables

    await importUsers(); // Import des utilisateurs

    console.log(

      "La base de données db\_passion\_lecture a bien été synchronisée"

    );

  } catch (error) {

    console.error(

      "Erreur lors de l'initialisation de la base de données :",

      error

    );

  }

};

// Fonction pour importer les utilisateurs fictifs dans la base de données

const importUsers = async () => {

  for (const user of users) {

    const hash = await bcrypt.hash(user.password, 10);

// Hashage du mot de passe avec un sel

    const createdUser = await User.create({

      username: user.username,

      password: hash,

      isAdmin: user.isAdmin,

    });

    console.log(createdUser.toJSON());

// Affichage des données de l'utilisateur importé

  }

};

// Export des éléments nécessaires

export { sequelize, initDb, User };

USER.MJS

import express from "express";

import { User } from "../db/sequelize.mjs";

import { auth } from "../auth/auth.mjs";

const userRouter = express();

// Récupérer tous les utilisateurs

userRouter.get("/", auth, async (req, res) => {

  try {

    const users = await User.findAll({

      attributes: ["id", "username", "isAdmin"],

    });

    return res.json({ users });

  } catch (error) {

    return res.status(500).json(error);

  }

});

// Récupérer un utilisateur par son ID

userRouter.get("/id/:id/", auth, async (req, res) => {

  try {

    const user = await User.findByPk(req.params.id);

    if (!user) {

      return res

        .status(404)

        .json({ message: "L'utilisateur demandé n'existe pas." });

    }

    return res.json({

      message: `L'utilisateur dont l'id vaut ${user.id} a bien été récupéré.`,

      data: user,

    });

  } catch (error) {

    return res.status(500).json({

      message: "L'utilisateur n'a pas pu être récupéré.",

      data: error,

    });

  }

});

// Récupérer un utilisateur par son nom d'utilisateur

userRouter.get("/:username", auth, async (req, res) => {

  try {

    const user = await User.findOne({

      where: { username: req.params.username },

    });

    if (!user) {

      return res

        .status(404)

        .json({ message: "L'utilisateur demandé n'existe pas." });

    }

    return res.json({

      message: `L'utilisateur dont le nom d'utilisateur est ${user.username} a bien été récupéré.`,

      data: user,

    });

  } catch (error) {

    return res.status(500).json({

      message: "L'utilisateur n'a pas pu être récupéré.",

      data: error,

    });

  }

});

export { userRouter };

# Conclusion

Ce projet m’a aidé à comprendre les étapes nécessaires pour une bonne sécurisation d’une application. J’ai trouvé très intéressante la réalisation de ce projet qui vise à nous former dans la création sécure de sites web et applications. La utilisation du https, le hachage du mot de passe pour ne le garder en dur dans la base de données et l’utilisation d’un token JWT sont des concepts que j’ai mieux compris durant la réalisation de ce projet.