PROBLEMY BEZPIECZEŃSTWA KOMPUTEROWEGO W SYSTEMACH INFORMATYCZNYCH

Projekt: Resume Generator – zadanie indywidualne nr 1

inż. Julia Napora

Identyfikacja klas użytkowników

Użytkowników reprezentuje jedna klasa, osoba która jest w trakcie szukania lub zmiany pracy i potrzebują wygenerować swoje CV. Najczęściej korzystać będą z niej osoby, które stawiają swoje pierwsze kroki w branży, są od razu po studiach lub w ich trakcie.

Analiza preferencji i oczekiwań użytkowników docelowych

Użytkownicy docelowi szukają aplikacji pozwalającej na wygenerowanie CV w prosty i szybki sposób oraz wygenerowanie do niego szablonu. Przydatnym narzędziem będzie możliwość wydrukowania i zapisania do pliku pdf.

Analiza zadań

- Stworzyć czytelną i prostą w użyciu aplikację webową
- Umożliwić logowanie użytkownikom
- Umożliwić użytkownikom generowania CV
- Stworzyć jak najwięcej możliwości personalizacji dokumentu
- Formularz generowania CV powinien być przejrzysty.

Implementacja aplikacji

Aplikacja została wykonana przy pomocy Mongo DB w wersji chumrowej Express React JS Node.js.

Uruchomienie serwera .
W celu uruchomienia serwera należy za pomocą wiersza poleceń wejść do folderu, w którym znajduje się kod i uruchomić polecenie *npm install*, które instaluje wszystkie potrzebne zależności. Następni należy uruchomić polecenie npm start. W konsoli powinny pojawić się komunikaty:
[nodemon] restarting due to changes...
[nodemon] starting `node server.js`
Listening on port 8080...
Connected to database sucessfully

Uruchomienie warstwy klienta:
W celu uruchomienia warstwy klienta należy wpisać te same polecenia, które wykorzystywane były do uruchomienia serwera.

Baza łączy się automatycznie.

JSON Web Token (JWT) to standard bezpiecznego przesyłania informacji w formacie JSON. Informacje są zabezpieczane poprzez podpisywanie cyfrowe za pomocą klucza tajnego lub pary kluczy publicznego i prywatnego. JWT często służą do autoryzacji i weryfikacji dostępu do zasobów bez konieczności przechowywania ich po stronie serwera. Token składa się z trzech części: nagłówka, zawartości i sygnatury. Nagłówek zawiera informacje o tokenie, zawartość przechowuje roszczenia dotyczące użytkownika i dodatkowe dane, a sygnatura służy do weryfikacji integralności i autentyczności tokenu.

Token wygląda tak: xxxxx.yyyyy.zzzzz.

Schemat działania JWT:
Po zalogowaniu, klient wysyła na serwer swoje dane logowania (email i hasło). Serwer sprawdza je w swojej bazie danych MongoDB i jeśli są poprawne, serwer zwraca dwie wartości tokenów: accessToken z krótkim okresem ważności i refreshToken z dłuższym okresem ważności. Klient przechowuje te tokeny w swojej pamięci lokalnej, np. w LocalStorage. Podczas wykonywania żądania chronionego, klient przesyła accessToken jako nagłówek Authorization: Bearer <accessToken> w żądaniu API. Serwer weryfikuje poprawność tokenu przy użyciu modułu jsonwebtoken i odpowiada na żądanie lub zwraca błąd, jeśli weryfikacja nie powiedzie się. W tle, klient regularnie odnawia token za pomocą refreshToken, aby mieć aktualne dane i prawa użytkownika.

Elementy kodu:

Utworzenie modeli przechowujących dane w bazie.

```
import mongoose from "mongoose";

const Schema = mongoose.Schema;

const userSchema = new Schema({
    userName: {
        type: String,
        required: true,
    },
    email: {
        type: String,
        required: true,
        unique: true,
    },
    password: {
        type: String,
        required: true,
    },
    roles: {
        type: [String],
        enum: ["user", "admin", "super_admin"],
        default: ["user"],
    },
});

const User = mongoose.model("User", userSchema);

export default User;
```

Rys.1 Model przechowujący dane użytkownika.

```
import mongoose from "mongoose";

const Schema = mongoose.Schema;

const userTokenSchema = new Schema({
    userId: {
        type: Schema.Types.ObjectId,
        required: true,
    },
    token: {
        type: String,
        required: true,
    },
    createdAt: {
        type: Date,
        default: Date.now,
        expires: 30 * 86400, // 30 days
    },
});

const UserToken = mongoose.model("UserToken", userTokenSchema);

export default UserToken;
```

Rys. 2 Model przechowujący dane o tokenie.

Na rysunku 2 pokazany jest schemat danych dotyczących tokena jakie będą przechowywane w bazie. Można zauważyć, że przechowujemy tu datę jego utworzenia, oraz określamy jego ważność.

Rys.3 Funkcja generująca token

Na rysunku 3 przedstawiono kod funkcji generującej token. Wykorzystuje on moduł jsonwebtoken oraz wcześniej zdefiniowany schemat. Funkcja generująca tokent wykonuje się asynchronicznie. Do szyfrowania wykorzystywane jest hasło, przechowywane w pliku .env. accesToken zapisywany jest w Local storage użytkownika w celu zabezpieczenia danych. Można wykorzystać również metodę przechowywania tokenu w plikach cookie, ale należy wtedy zabezpieczyć go flagą httponly, która zapewni bezpieczną komunikację. AccesToken po wylogowaniu użytkownika jest usuwany z pamięci lokalnej.

```
import UserToken from "../models/UserToken.js";
import jwt from "jsonwebtoken";

const verifyRefreshToken = (refreshToken) => {
  const privateKey = process.env.REFRESH_TOKEN_PRIVATE_KEY;

  return new Promise((resolve, reject) => {
    UserToken.findOne({ token: refreshToken }, (err, doc) => {
      if (!doc)
        return reject({ error: true, message: "Invalid refresh token" });

      jwt.verify(refreshToken, privateKey, (err, tokenDetails) => {
        if (err)
          return reject({ error: true, message: "Invalid refresh token" });
        resolve({
          tokenDetails,
          error: false,
          message: "Valid refresh token",
        });
      });
    });
  });
};

export default verifyRefreshToken;
```

Rys.4 Funkcja weryfikująca RefresherToken

Funkcja przedstawiona na rysunku 4 weryfikuje refresherToken. Pobiera z pliku .env klucz prywatny, znajduje refresherToken w bazie danych, jeżeli nie może znaleźć zwraca błąd. Jeżeli zwrócony token nie zgadza się to również zwraca błąd.

```
const router = Router();

// signup
router.post("/signUp", async (req, res) => {
  try {
    const { error } = signUpBodyValidation(req.body);
    if (error)
      return res
        .status(400)
        .json({ error: true, message: error.details[0].message });

    const user = await User.findOne({ email: req.body.email });
    if (user)
      return res
        .status(400)
        .json({ error: true, message: "User with given email already exist" });

    const salt = await bcrypt.genSalt(Number(process.env.SALT));
    const hashPassword = await bcrypt.hash(req.body.password, salt);

    await new User({ ...req.body, password: hashPassword }).save();

    res
      .status(201)
      .json({ error: false, message: "Account created sucessfully" });
  } catch (err) {
    console.log(err);
    res.status(500).json({ error: true, message: "Internal Server Error" });
  }
});
```

Rys.5 Zapytania wysyłane przez serwer odnośnie rejestracji użytkownika.

```
// login
router.post("/logIn", async (req, res) => {
  try {
    const { error } = logInBodyValidation(req.body);
    if (error)
      return res
        .status(400)
        .json({ error: true, message: error.details[0].message });

    const user = await User.findOne({ email: req.body.email });
    if (!user)
      return res
        .status(401)
        .json({ error: true, message: "Invalid email or password" });

    const verifiedPassword = await bcrypt.compare(
      req.body.password,
      user.password
    );
    if (!verifiedPassword)
      return res
        .status(401)
        .json({ error: true, message: "Invalid email or password" });

    const { accessToken, refreshToken } = await generateTokens(user);

    res.status(200).json({
      error: false,
      accessToken,
      refreshToken,
      message: "Logged in sucessfully",
    });
  } catch (err) {
    console.log(err);
    res.status(500).json({ error: true, message: "Internal Server Error" });
  }
});
```

Rys.6 Zapytania wysyłane przez serwer odnośnie logowania użytkownika.

```
import jwt from "jsonwebtoken";

const auth = async (req, res, next) => {
  const token = req.header("x-access-token");
  if (!token)
    return res
      .status(403)
      .json({ error: true, message: "Access Denied: No token provided" });

  try {
    const tokenDetails = jwt.verify(
      token,
      process.env.ACCESS_TOKEN_PRIVATE_KEY
    );
    req.user = tokenDetails;
    next();
  } catch (err) {
    console.log(err);
    res
      .status(403)
      .json({ error: true, message: "Access Denied: Invalid token" });
  }
};

export default auth;
```

Rys.7 Funkcja weryfikująca poprawność tokenu przy logowaniu

```
import { Router } from "express";
import auth from "../middleware/auth.js";
import roleCheck from "../middleware/roleCheck.js";

const router = Router();

router.get("/details", auth, (req, res) => {
    res.status(200).json({ message: "user authenticated." });
});

export default router;
```

Rys.8 Zapytanie weryfikujące token użytkownika wykorzystując funkcję z rys.7