Beata Pańczyk

Szkielety programistyczne - weryfikacja tokena

Efektem działania projektu z poprzedniego laboratorium jest aplikacja serwerowa obsługująca endpointy do rejestracji i logowania użytkownika. W rezultacie poprawnego logowania, serwer wysyła odpowiedź (response) z tokenem, który jest przechowywany po stronie klienta w localStorage przeglądarki (alternatywnie można skorzystać z cookie). W celu uzyskania dostępu dla autoryzowanych użytkowników, aplikacja kliencka musi w żądaniu wysyłanym do API przekazać token, który wymaga weryfikacji po stronie serwera. Po prawidłowej weryfikacji dopiero będzie przesyłana odpowiedź z żądanymi danymi.
Celem niniejszego laboratorium jest rozbudowanie projektu o elementy uwierzytelniania za pomocą tokenu JWT. Token od klienta trafia wraz z żądaniem do serwera, jest parsowany, a następnie weryfikowany pod kątem prawidłowości, uprawnień, ważności itp.

JWT posiada trzy kluczowe elementy:

- Header – przechowuje informacje na temat algorytmu szyfrowania oraz typie tokena,
- Payload – dowolny przekazywany ładunek. Najczęściej są w nim przechowywane informacje na temat roli i praw użytkownika, czy też czasu ważności tokena,
- Verify Signature – podpis cyfrowy, składa się z zaszyfrowanego elementu Header i Paylod.

Sygnatura to podpis cyfrowy potwierdzający autentyczność danych zawartych w tokenie. Weryfikacja sygnatury daje pewność, że nadawca jest tym za kogo się podaje. Sygnaturę tworzy się korzystając z zakodowanego (określonym algorytmem) nagłówka i ładunku, oraz z podanego przez nas (lub wygenerowanego po stronie serwera) ciągu znaków (secret key).

W pliku modelu ***user.js*** do wygenerowania tokena JWT zastosowano moduł ***jsonwebtoken*** i metodę **sign** w sposób następujący:

```
var jwt = require('jsonwebtoken');
//podpis domyślnie szyfrowany algorytmem HMAC SHA256
const token = jwt.sign({ _id: this._id }, process.env.JWTPRIVATEKEY, {
        expiresIn: "7d"
    })
```

Weryfikację tokena można zrealizować za pomocą metody verify:

```
jwt.verify(token, process.env.JWTPRIVATEKEY, (err, decoded) => {…} );
```

Aplikacja kliencka pobiera token z localStorage i dołącza go do żądania w nagłówku **'x-access-token'.**
Na serwerze dostęp do przekazanego w nagłówku żądania tokena realizuje kod:

```
let token = req.headers["x-access-token"];
```

UWAGA! Kolejność instrukcji w głównym pliku index.js serwera powinna być następująca:

- najpierw wszystkie importy modułów (**require**)
- utworzenie aplikacji express: **const app = express()**
- zastosowanie formatu JSON do wymiany danych: **app.use(express.json())**
- nie blokowanie przez przeglądarkę wysyłania żądań do serwera z innych lokalizacji: **app.use(cors())**
- import reguł routingu i dodanie funkcji middleware (autentykacji) do wskazanych tras
- połączenie z bd (w zasadzie może być w dowolnym miejscu): **connection()**
- **app.listen(port, …)**

Zadanie 7.2.
Etap I Dodatkowe elementy w aplikacji serwerowej

1. Utworzenie katalogu **middleware** i dodanie do niego pliku **tokenVerification.js**:

```
const jwt = require("jsonwebtoken")

function tokenVerification(req, res, next) {
  //pobranie tokenu z nagłówka:
  let token = req.headers["x-access-token"];
  if (!token) {
    res.status(403).send({ message: "No token provided!" });
  }
  //jeśli przesłano token - weryfikacja jego poprawności:
  jwt.verify(token, process.env.JWTPRIVATEKEY, (err, decodeduser) => {
    if (err) {
      console.log("Unauthorized!")
      res.status(401).send({ message: "Unauthorized!" });
    }
    console.log("Token poprawny, użytkownik: "+decodeduser._id)
    req.user = decodeduser
    next()
  })
}
module.exports = tokenVerification
```

2. Dodanie dodatkowej trasy do pliku users.js (z folderu **routes)**, np.:

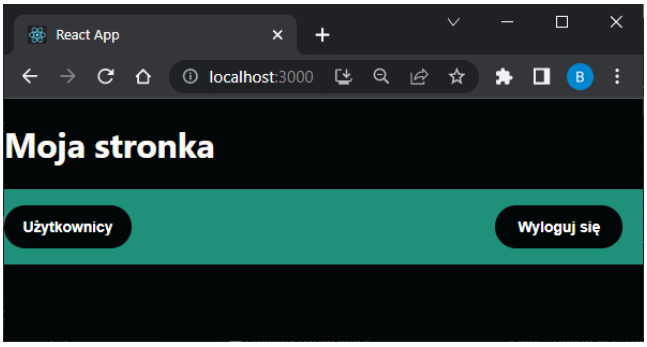
```
router.get("/", async (req, res) => {
  //pobranie wszystkich użytkowników z bd:
  User.find().exec()
    .then(async () => {
      const users = await User.find();
       //konfiguracja odpowiedzi res z przekazaniem listy użytkowników:
      res.status(200).send({ data: users, message: "Lista użytkowników" });
    })
    .catch(error => {
      res.status(500).send({ message: error.message });
    });
})
```

3. Dodanie do pliku **index.js importu funkcji middleware**:

```
  const tokenVerification = require('./middleware/tokenVerification')
…//…
  //trasy wymagające weryfikacji tokenem:
… app.get("/api/users/",tokenVerification)
  //…
  //POTEM trasy nie wymagające tokena (kolejność jest istotna!)
  app.use("/api/auth", authRoutes)
  app.use("/api/users", userRoutes) //tylko  metoda get wymaga tokena
```

Etap II Dodatkowe elementy w aplikacji klienckiej

1. Rozbuduj komponent Main o dodatkowy przycisk **Użytkownicy** (rysunek). Obsługę kliknięcia na ten przycisk ma realizować funkcja **handleGetUsers.** Dodaj też dodatkową zmienną stanu komponentu Main, która będzie przechowywała listę użytkowników pobraną z bazy danych np.
   const [dane, ustawDane] = useState([])

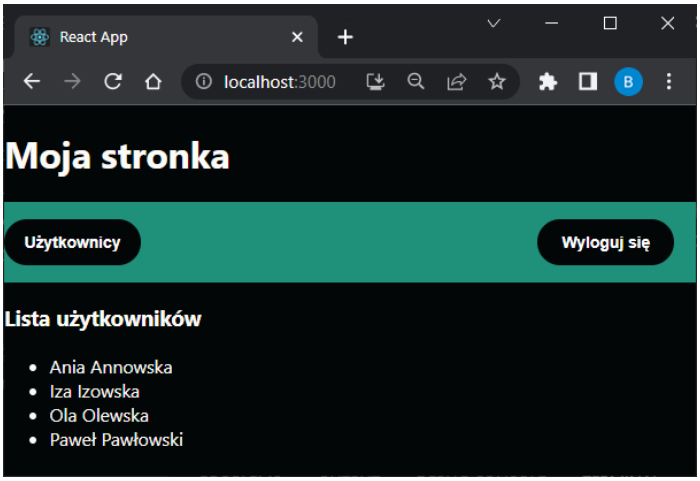


2. W komponencie Main zdefiniuj też funkcję **handleGetUsers**, która powinna wysłać asynchronicznie żądanie (z tokenem uwierzytelniającym jeśli jest) o dane wszystkich użytkowników zapisanych w bd. Kod funkcji **handleGetUsers** może wyglądać następująco:

```
const handleGetUsers = async (e) => {
      e.preventDefault()

……..//pobierz token z localStorage:
    const token = localStorage.getItem("token")
    //jeśli jest token w localStorage to:
    if (token) {
      try {
        //konfiguracja zapytania asynchronicznego z tokenem w nagłówku:
        const config = {
          method: 'get',
          url: 'http://localhost:8080/api/users',
          headers: { 'Content-Type': 'application/json', 'x-access-token': token }
        }
        //wysłanie żądania o dane:
        const { data: res } = await axios(config)
        //ustaw dane w komponencie za pomocą hooka useState na listę z danymi przesłanymi
        //z serwera – jeśli został poprawnie zweryfikowany token
        ustawDane(res.data) // `res.data` - zawiera sparsowane dane – listę
      } catch (error) {
        if (error.response && error.response.status >= 400 &&error.response.status <= 500)
        {
          localStorage.removeItem("token")
          window.location.reload()
        }
      }
    }
  }
```

3. W komponencie **Main** w bloku **return**, poniżej panelu z przyciskami wyświetl warunkowo listę użytkowników (jeśli kliknięto na przycisk).



W tym celu należy zdefiniować dodatkowe pliki komponentów **User.js** i **Users.js**, podobnie jak w przypadku zadania z listą produktów (komponent Product i Products) i zastosować polecenie (ewentualnie uzupełnione dodatkowymi warunkami sprawdzającymi stan komponentu):

{dane.length>0 ? <Users users={dane} /> : }

Komponent User **może być zdefiniowany jako:**

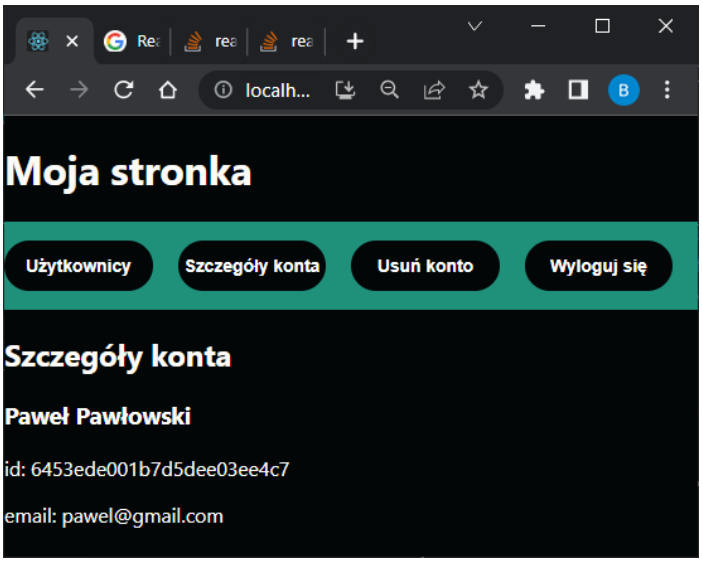
```
const User = (props) => {
   const user = props.user;
   return (  <li>  {user.firstName} {user.lastName}  </li> );
}
export default User
```

Komponent Users może wyglądać wtedy następująco:

```
import User from "./User"
function Users(props) {
   const users = props.users;
   return (
      <ul> {users.map((user) => <User key={user._id} value={user._id} user={user}/> )} </ul>  );
}
export default Users
```

Zadanie 7.3.

1. Dodaj kolejny przycisk umożliwiający wyświetlenie szczegółów konta aktualnie zalogowanego użytkownika.
2. Dodaj przycisk umożliwiający usunięcie konta aktualnie zalogowanego użytkownika, po wcześniejszym wyświetleniu okienka confirm z zapytaniem czy na pewno.
3. Wykorzystaj odpowiedź z serwera (response) i pokaż informacje przekazane w obiekcie **res.message** jako tytuł poprzedzający wyświetlaną treść (listę użytkowników lub szczegóły konta).



UWAGA! W każdym żądaniu przesyłanym od klienta (asynchronicznie za pomocą axios) wymagającym uwierzytelnienia użytkownika przesyłany jest token, który jest sprawdzany po stronie serwera (u nas za pomocą funkcji middleware **tokenVerification)**. W funkcji tej polecenie:

req.user = decodeduser

umieszcza odkodowany obiekt użytkownika (decodeduser) w obiekcie żądania (**req.user)**. Do obiektu **req** mają dostęp kolejne funkcje obsługujące to żądanie, dzięki czemu obiekt **user** można uzyskać za pomocą polecenia **req.user. N**ie trzeba więc przesyłać id w url w celu wykonania operacji pokazania szczegółów lub usunięcia konta aktualnie zalogowanego użytkownika. Id można pobrać po stronie serwera jako:

req.user._id

i mając id można już sięgnąć do bazy danych po informacje o tym użytkowniku lub usunąć jego konto korzystając z metod poznanych na laboratorium z MongoDB.