

LABORATORIUM CYBERBEZPIECZEŃSTWO

**Data wykonania
ćwiczenia:**

07.10.2024

Rok studiów:

4

Semestr:

7

Grupa studencka:

2

Grupa laboratoryjna:

2B

Ćwiczenie nr.

1

Temat: Systemy bezpieczeństwa oparte na hasłach

Osoby wykonujące ćwiczenia:

1. Patryk Pawełek

Katedra Informatyki i Automatyki

Sprawozdanie z realizacji zadania – Systemy bezpieczeństwa oparte na hasłach

1. Wprowadzenie

Hasła są podstawowym środkiem ochrony dostępu do systemów oraz informacji o charakterze poufnym. Aby skutecznie pełniły swoją rolę, wymagają odpowiedniego zarządzania, zgodnie z wytycznymi technicznymi i organizacyjnymi. Proces uwierzytelniania umożliwia weryfikację tożsamości użytkowników poprzez podanie poufnego hasła. Celem tego zadania było stworzenie systemu bezpieczeństwa zapewniającego efektywne zarządzanie użytkownikami, ich hasłami oraz zasadami ich tworzenia.

2. Cel zadania

Zadanie obejmowało stworzenie programu implementującego system bezpieczeństwa haseł z następującymi funkcjonalnościami:

1. Obsługa dwóch ról: administratora (ADMIN) oraz użytkownika.
2. Administrator posiada możliwość:
 - zmiany hasła,
 - dodawania, modyfikowania oraz usuwania użytkowników,
 - blokowania kont oraz włączania/wyłączania ograniczeń dotyczących haseł,
 - ustawiania ważności hasła oraz wymuszania jego zmiany po określonym czasie.
3. Zwykły użytkownik może zmieniać swoje hasło oraz wylogować się.
4. Program zawiera mechanizm logowania z weryfikacją poprawności identyfikatora i hasła.
5. Przy pierwszym logowaniu użytkownik jest proszony o zmianę hasła.
6. Program korzysta z bezpiecznego algorytmu do hashowania haseł (bcrypt).
7. System musi weryfikować spełnianie indywidualnych zasad dotyczących hasła, w moim przypadku są to znaki z kategorii: wielkie litery, małe litery, oraz znaki specjalne (sekcja 9).

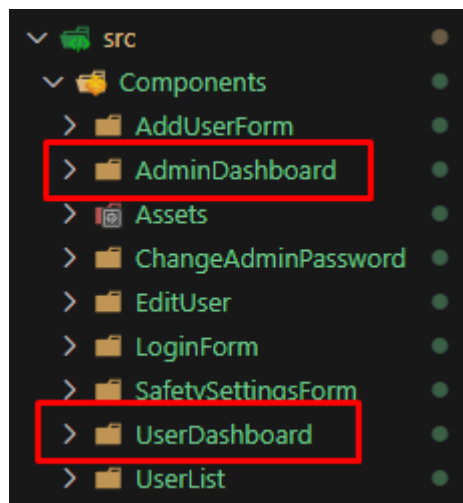
3. Implementacja programu

3.1. Struktura aplikacji

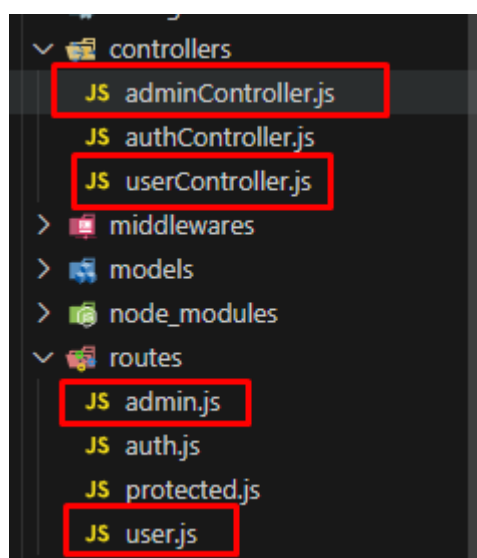
Aplikacja została stworzona z wykorzystaniem frameworka React, wykorzystana została baza danych MongoDB oraz serwer jest oparty na node oraz express. Program wykorzystuje bibliotekę **bcrypt** do bezpiecznego przechowywania haseł.

Obsługa dwóch ról: administratora (ADMIN) oraz użytkownika

Na frontendzie mamy do wykorzystania zarówno dashboard dla Admina oraz dla zwykłego Usera:

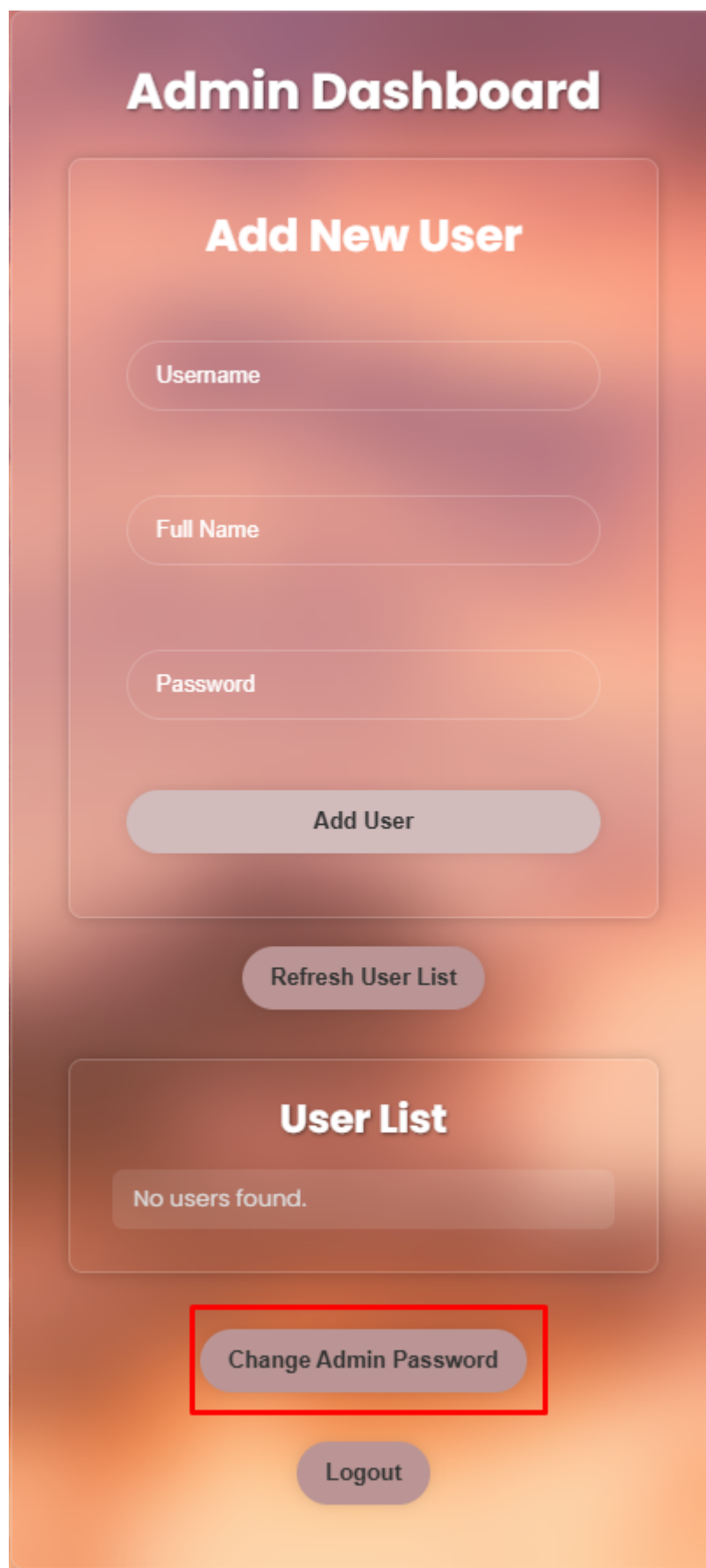


Jeśli chodzi o backend, to występują tam routes osobno dla admina oraz zwykłego usera, podobnie ma się sprawa w przypadku kontrolerów.



Możliwości Administratora:

a) Zmiana hasła:

A mockup of an Admin Dashboard with a warm orange-to-red gradient background. The dashboard is divided into several sections. At the top is the 'Admin Dashboard' title. Below it is a 'Add New User' section containing three input fields for 'Username', 'Full Name', and 'Password', followed by an 'Add User' button. Below this is a 'Refresh User List' button. The next section is 'User List', which currently displays 'No users found.' in a light gray box. At the bottom, there is a 'Change Admin Password' button highlighted with a red rectangular border, and a 'Logout' button below it.

Admin Dashboard

Add New User

Add UserRefresh User List

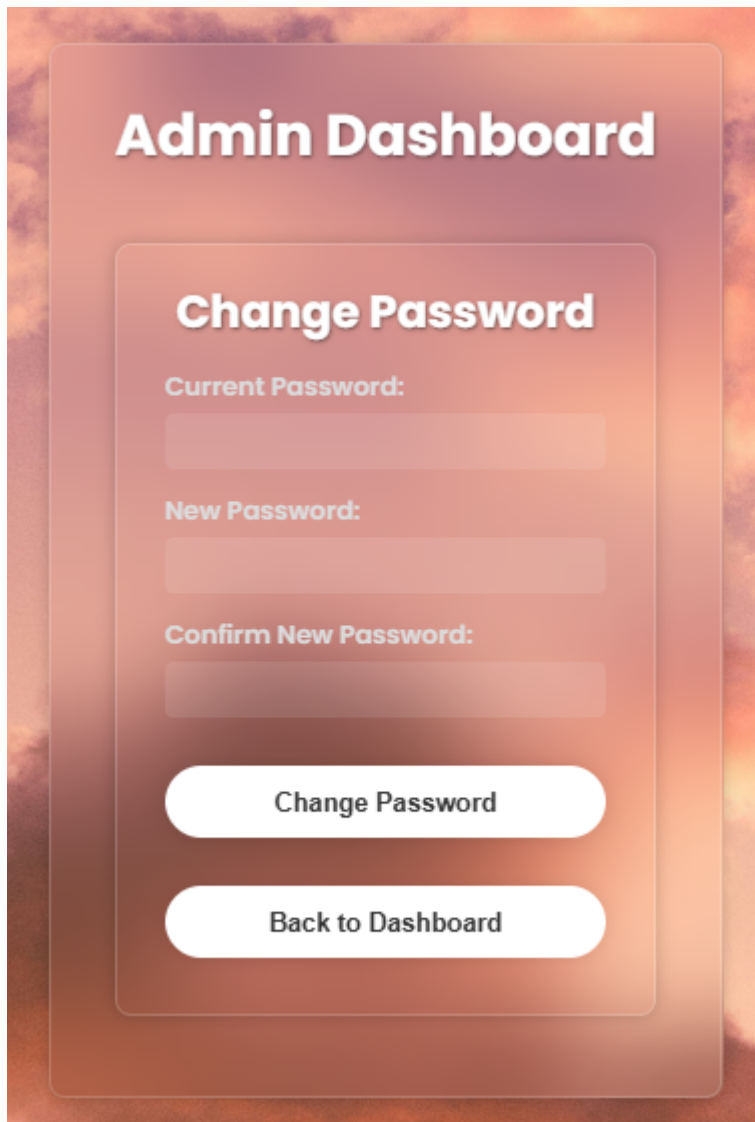
User List

No users found.

Change Admin Password

Logout

W AdminDashboard mamy możliwość zmiany hasła administratora, po kliknięciu na przycisk przenosi nas do innego komponentu gdzie możemy zmienić hasło.

A screenshot of a web form titled "Admin Dashboard" with a subtitle "Change Password". The form is set against a background of a sunset sky. It contains three input fields for "Current Password:", "New Password:", and "Confirm New Password:". Below the fields are two buttons: "Change Password" and "Back to Dashboard".

Admin Dashboard

Change Password

Current Password:

New Password:

Confirm New Password:

Change Password

Back to Dashboard

Poniżej znajduje się część kodu odpowiadająca za zmianę hasła administratora

```
5  const ChangePasswordForm = ({ onPasswordChange, onSuccess, onBack }) => {
6    const [currentPassword, setCurrentPassword] = useState('');
7    const [newPassword, setNewPassword] = useState('');
8    const [confirmPassword, setConfirmPassword] = useState('');
9    const [error, setError] = useState('');
10
11    const handleSubmit = async (e) => {
12      e.preventDefault();
13      setError('');
14
15      if (newPassword !== confirmPassword) {
16        setError('New passwords do not match.');
```

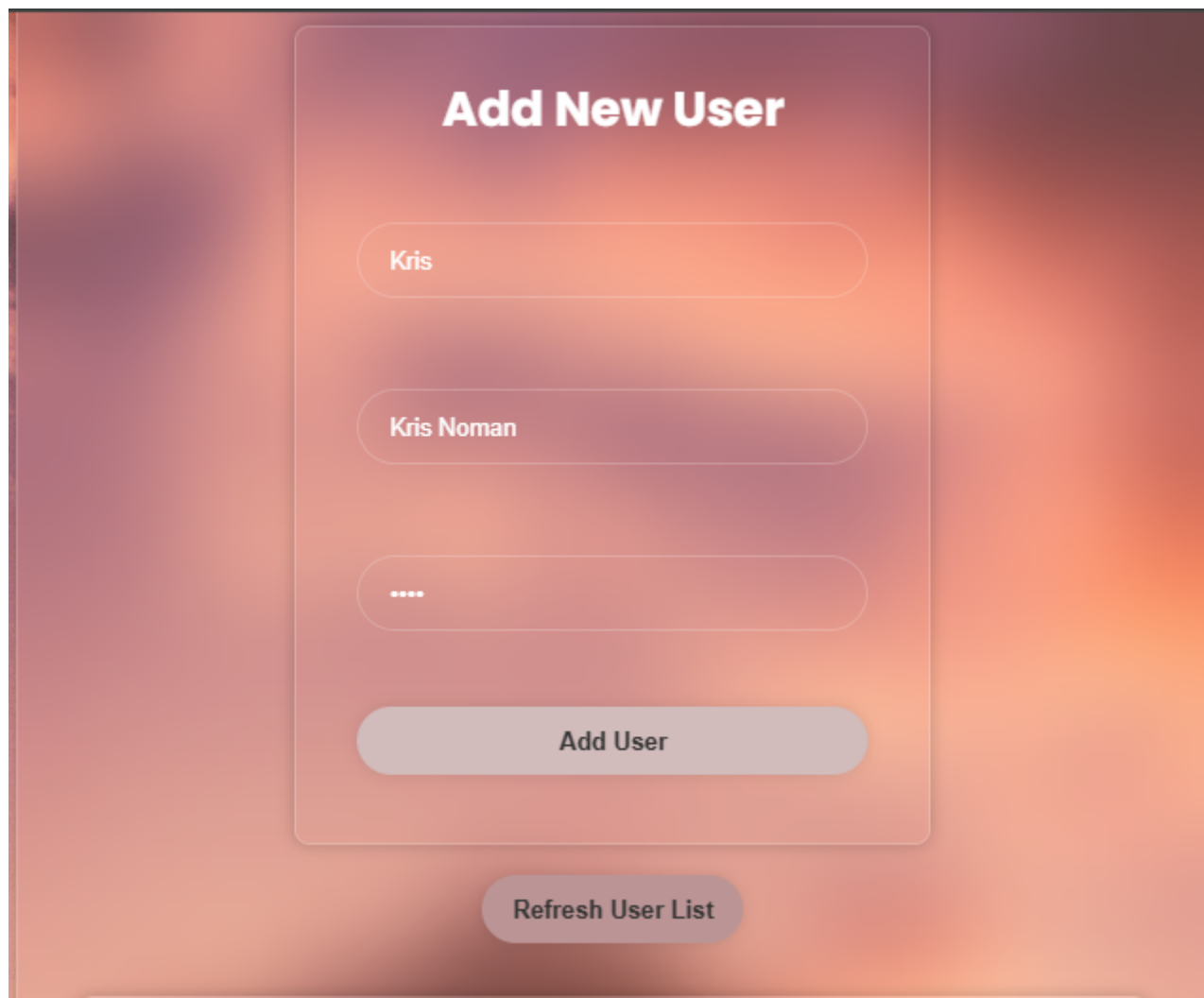
```
17      return;
18    }
19
20    try {
21      const token = localStorage.getItem('token');
22      await axios.put('http://localhost:5000/api/admin/change-password', {
23        currentPassword,
24        newPassword,
25      }, {
26        headers: {
27          'Authorization': `Bearer ${token}`,
28        },
29      });
30
31      onSuccess();
32      onPasswordChange();
33    } catch (err) {
34      setError(err.response?.data?.message || 'Error changing password');
```

```
35    }
36  };
37
38
39  return (
40    <form onSubmit={handleSubmit} className="change-password-wrapper">
41      <h2>Change Password</h2>
42      {error && <p style={{ color: 'red' }}>{error}</p>}
43      <div>
44        <label>Current Password:</label>
45        <input
46          type="password"
47          value={currentPassword}
48          onChange={(e) => setCurrentPassword(e.target.value)}
49          required
50        />
51      </div>
```

Natomiast tak wygląda kod od strony serwera:

```
3
4 // Change Admin Password
5 exports.changeAdminPassword = async (req, res) => {
6   const { currentPassword, newPassword } = req.body;
7   const adminId = req.user.id;
8
9   try {
10    const admin = await User.findById(adminId);
11    if (!admin) return res.status(404).json({ message: "Admin not found" });
12
13    const isMatch = await admin.matchPassword(currentPassword);
14    if (!isMatch)
15      return res.status(400).json({ message: "Current password is incorrect" });
16
17    const salt = await bcrypt.genSalt(10);
18    admin.password = await bcrypt.hash(newPassword, salt);
19
20    await admin.save();
21    res.json({ message: "Password updated successfully" });
22  } catch (error) {
23    console.error("Error in changeAdminPassword:", error);
24    res.status(500).json({ message: "Server error" });
25  }
26 };
27
```

b) Dodawanie, modyfikowanie, blokowanie oraz usuwanie użytkowników:

A screenshot of a mobile application interface for adding a new user. The background is a blurred orange and red gradient. A semi-transparent white rounded rectangle is centered on the screen. At the top of this rectangle, the text "Add New User" is written in bold black font. Below the title, there are three input fields, each with a light blue border and rounded corners. The first field contains the text "Kris". The second field contains the text "Kris Noman". The third field contains four dots "....", indicating a password. Below these fields is a light blue rounded button with the text "Add User" in bold black font. At the bottom of the screen, centered, is a light blue rounded button with the text "Refresh User List" in bold black font.

Add New User

Kris

Kris Noman

....

Add User

Refresh User List

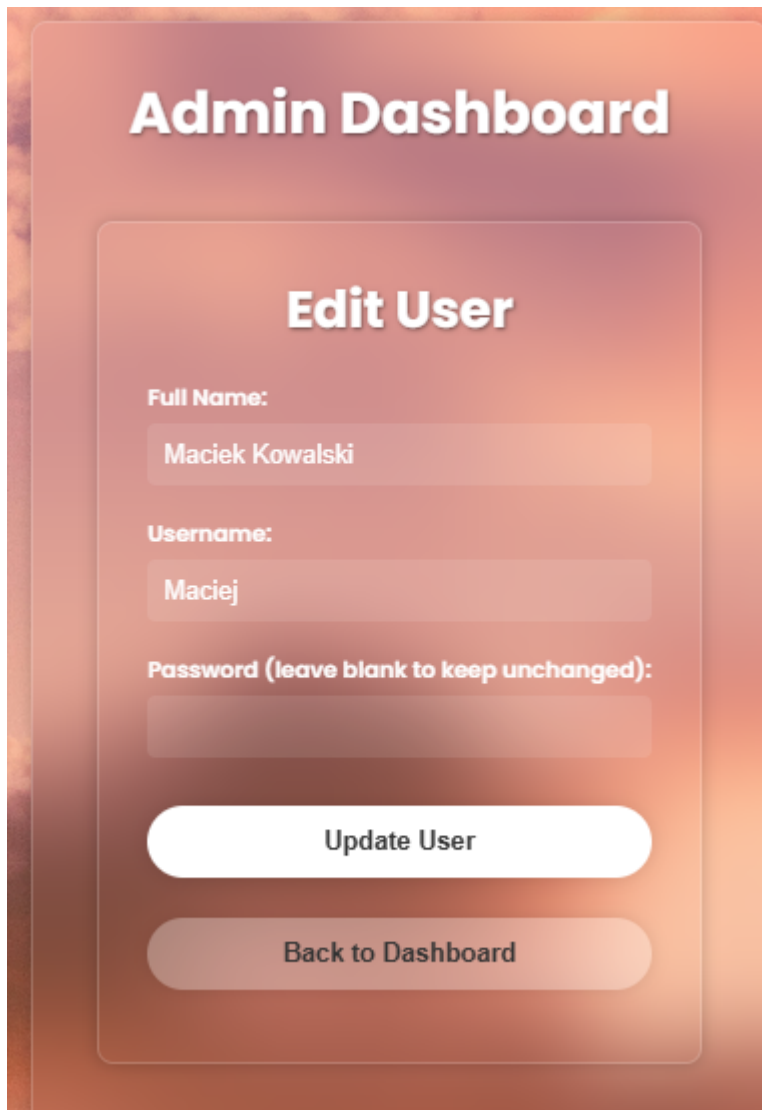
Formularz do dodawania nowych użytkowników.


```
51 // Add New User
52 exports.addNewUser = async (req, res) => {
53   const { username, fullName, password } = req.body;
54
55   try {
56
57     const existingUser = await User.findOne({ username });
58     if (existingUser) {
59       return res.status(400).json({ message: "Username already exists" });
60     }
61
62     const hashedPassword = await bcrypt.hash(password, 10);
63
64     const newUser = new User({
65       username,
66       fullName,
67       password: hashedPassword,
68     });
69
70     await newUser.save();
71
72     res.status(201).json({ message: "User added successfully" });
73   } catch (error) {
74     console.error("Error in addNewUser:", error);
75     res.status(500).json({ message: "Server error" });
76   }
77 };
```

Tak to wygląda od strony kodu na serwerze.



Po dodaniu kilku przykładowych użytkowników obok nich pojawiają się opcje takie jak Edit, Block, Delete oraz Safety Settings



The image shows a screenshot of an 'Admin Dashboard' with a focus on the 'Edit User' form. The dashboard has a dark orange background. The 'Edit User' form is a lighter orange rounded rectangle in the center. It contains three input fields: 'Full Name' with the value 'Maciek Kowalski', 'Username' with the value 'Maciej', and 'Password' which is empty. Below the fields are two buttons: 'Update User' (white with black text) and 'Back to Dashboard' (orange with black text).

Admin Dashboard

Edit User

Full Name:

Username:

Password (leave blank to keep unchanged):

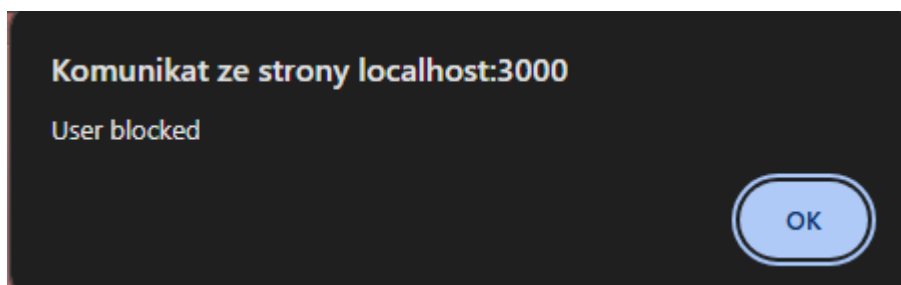
Update User

Back to Dashboard

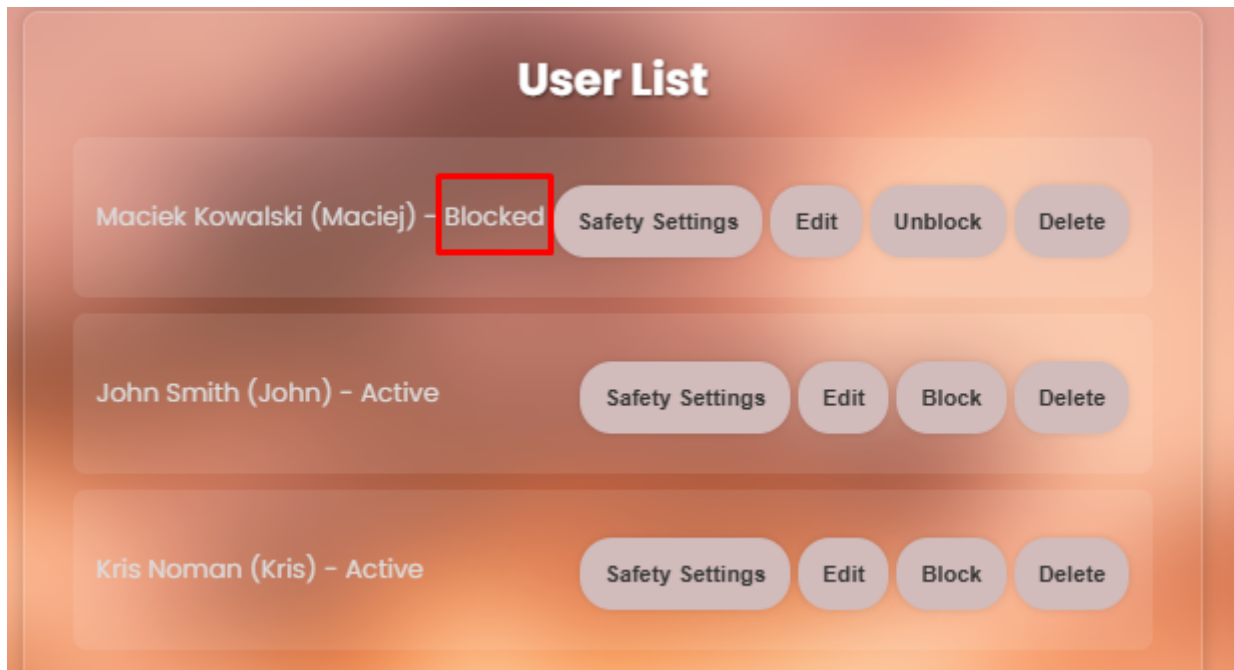
Po wybraniu opcji Edit mamy możliwość edytowania wszystkich pól użytkownika. Nazwa użytkownika, imię i nazwisko oraz hasło.

```
28 exports.modifyUserAccount = async (req, res) => {
29   const { fullName, username, password } = req.body;
30   const userId = req.params.id;
31
32   try {
33     const user = await User.findById(userId);
34     if (!user) return res.status(404).json({ message: "User not found" });
35
36     if (fullName) user.fullName = fullName;
37     if (username) user.username = username;
38     if (password) {
39       const salt = await bcrypt.genSalt(10);
40       user.password = await bcrypt.hash(password, salt);
41     }
42
43     await user.save();
44     res.json({ message: "User account updated successfully" });
45   } catch (error) {
46     console.error("Error in modifyUserAccount:", error);
47     res.status(500).json({ message: "Server error" });
48   }
49 };
```

Jeśli chodzi o edycję istniejących użytkowników tak wygląda kod od strony backendu.



Gdy klikniemy na przycisk Block wtedy pojawi się nam komunikat o zablokowaniu użytkownika, gdy klikniemy Ok zauważymy, że status użytkownika zmienił się na blocked, oraz użytkownik nie może się zalogować do swojego konta, i wyświetla mu się komunikat, że jego konto jest zablokowane, oraz powinien skontaktować się z administratorem.



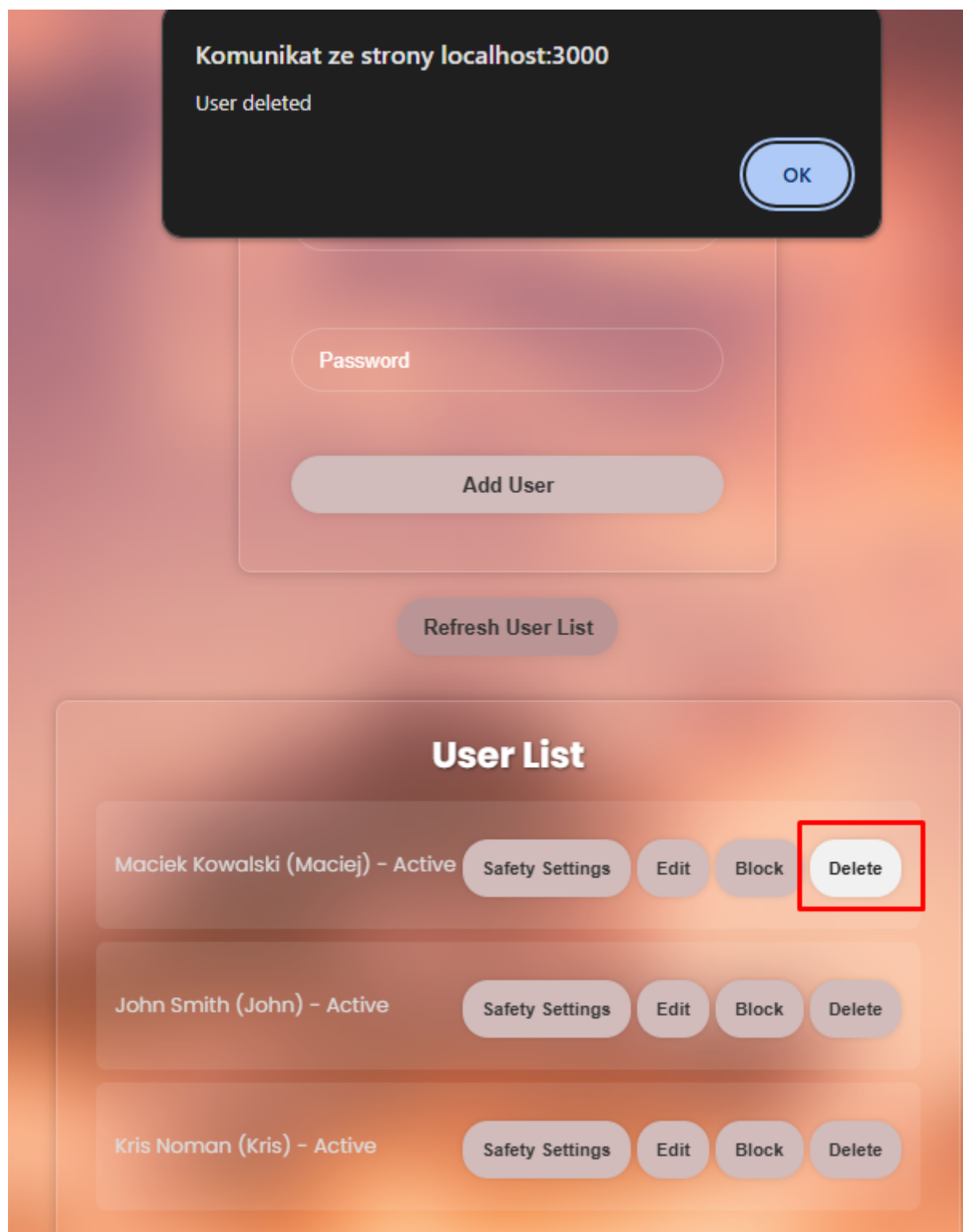
Blokowanie opiera się o 2 funkcje, jeśli chodzi o backend, jedna jest to blokowanie:

```
128 exports.blockUserAccount = async (req, res) => {
129   const userId = req.params.id;
130
131   try {
132     const user = await User.findById(userId);
133
134     if (!user) {
135       console.log("User not found:", userId);
136       return res.status(404).json({ message: "User not found" });
137     }
138
139     user.blocked = true;
140     await user.save();
141
142     const verifiedUser = await User.findById(userId);
143     if (!verifiedUser.blocked) {
144       console.error("Block operation failed - user not blocked");
145       return res.status(500).json({
146         message: "Failed to block user - please try again",
147         debug: { currentState: verifiedUser.blocked },
148       });
149     }
150   }
```

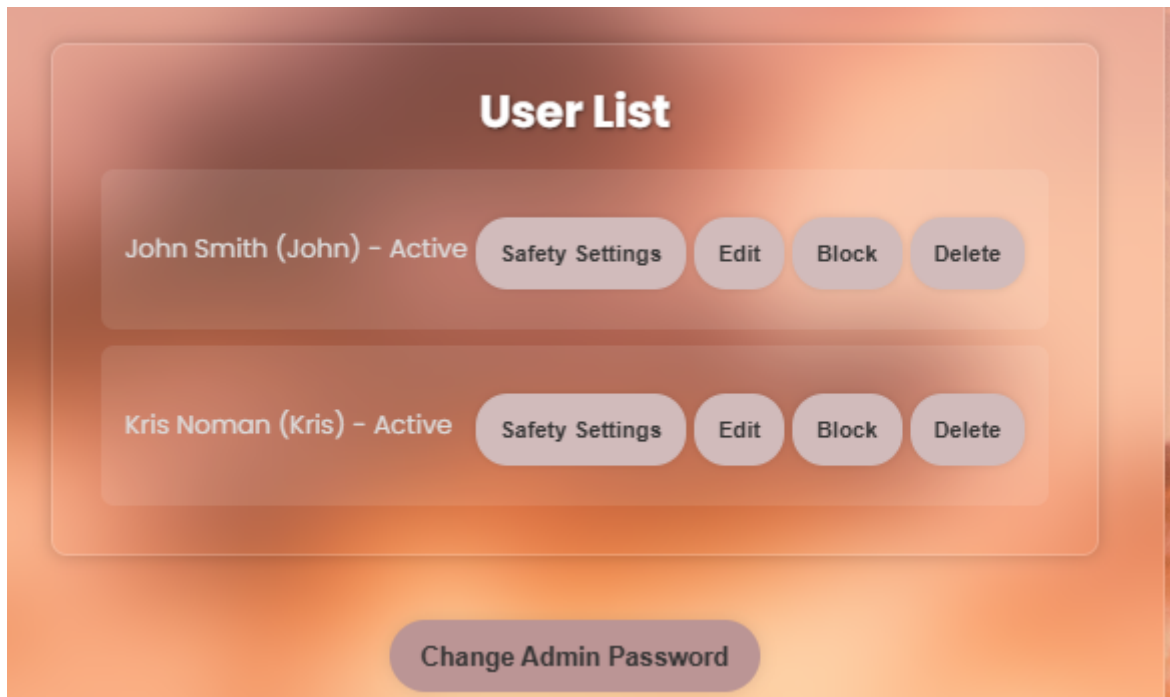
Natomiast następna to odblokowanie:

```
89 exports.unlockUserAccount = async (req, res) => {
90   const userId = req.params.id;
91
92   try {
93     const user = await User.findById(userId);
94
95     if (!user) {
96       console.log("User not found:", userId);
97       return res.status(404).json({ message: "User not found" });
98     }
99
100    user.blocked = false;
101    await user.save();
102
103    const verifiedUser = await User.findById(userId);
104
105    if (verifiedUser.blocked) {
106      console.error("Unblock operation failed - user still blocked");
107      return res.status(500).json({
108        message: "Failed to unblock user - please try again",
109        debug: { currentState: verifiedUser.blocked },
110      });
111    }
112
113    return res.json({
114      message: "User account unblocked",
115      user: verifiedUser,
116    });
117  } catch (error) {
118    console.error("Error in unlockUserAccount:", error);
119    return res.status(500).json({
120      message: "Server error",
121      error: error.message,
122    });
123  }
124 };
```

Kolejną rzeczą, którą admin jest w stanie robić z użytkownikami jest ich usuwanie.



Po kliknięciu przycisku delete pojawia się nam komunikat o usunięciu użytkownika, a gdy klikniemy "OK" pokaże nam się lista użytkowników już bez tego użytkownika, który został przez nas usunięty.

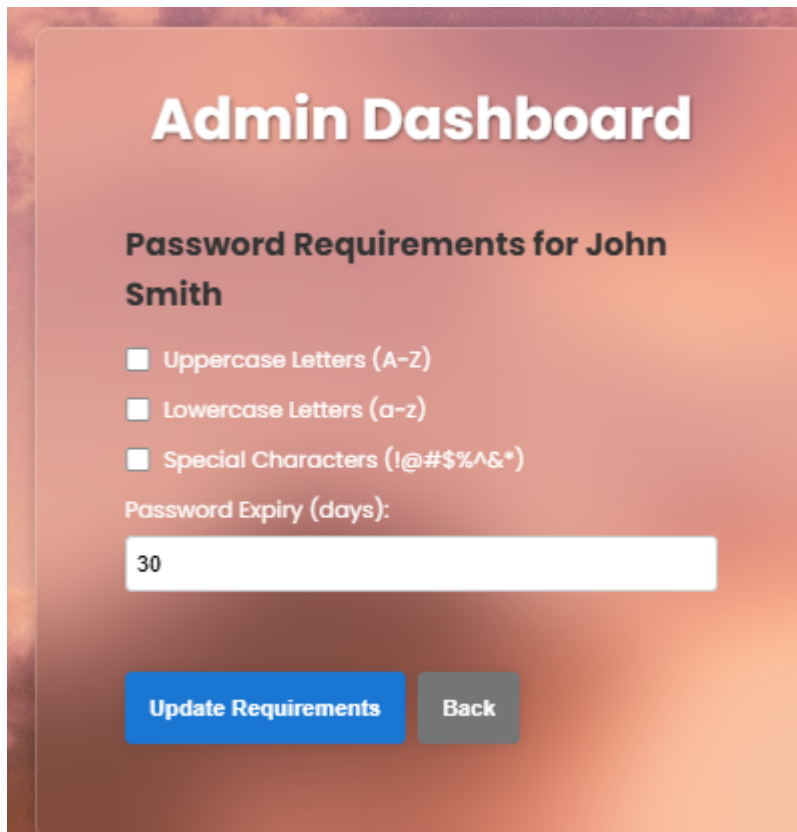


Usuwanie jest obsługiwane przez prostą funkcję `findByIdAndDelete`

```
162 // Delete User Account
163 exports.deleteUserAccount = async (req, res) => {
164   const userId = req.params.id;
165
166   try {
167     await User.findByIdAndDelete(userId);
168     res.json({ message: "User account deleted" });
169   } catch (error) {
170     console.error("Error in deleteUserAccount:", error);
171     res.status(500).json({ message: "Server error" });
172   }
173 };
174
```

c) Ustawianie ważności hasła oraz wymuszanie jego zmiany po określonym czasie.

Po kliknięciu na "Safety Settings", które jest dostępne obok każdego użytkownika otwiera nam się okno, w którym możemy zmienić wymagania dotyczące hasła (duże litery, małe litery oraz znaki specjalne), oraz dodatkowo czas po którym dane hasło wygasa.



The image shows a web interface titled "Admin Dashboard" for a user named John Smith. It contains three unchecked checkboxes for password requirements: Uppercase Letters (A-Z), Lowercase Letters (a-z), and Special Characters (!@#\$\$%^&*). Below these is a "Password Expiry (days):" label and a text input field containing the number "30". At the bottom are two buttons: "Update Requirements" in blue and "Back" in grey.

Admin Dashboard

Password Requirements for John Smith

- ☐ Uppercase Letters (A-Z)
- ☐ Lowercase Letters (a-z)
- ☐ Special Characters (!@#\$\$%^&*)

Password Expiry (days):

[Update Requirements](#) [Back](#)

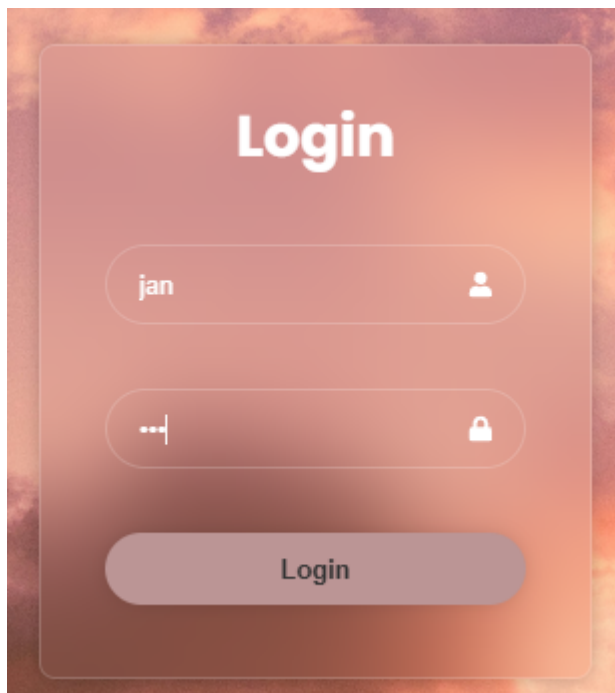
Jeśli chodzi o kod który to obsługuje:


```
212 // Set Safety Settings
213
214 exports.setSafetySettings = async (req, res) => {
215   const {
216     requireUpperCase,
217     requireSpecialChar,
218     requireLowerCase,
219     expiryDays,
220     isBlocked,
221   } = req.body;
222   const userId = req.params.id;
223
224   try {
225     const user = await User.findById(userId);
226     if (!user) return res.status(404).json({ message: "User not found" });
227
228     // Update the user's settings
229     user.requireLowerCase = requireLowerCase;
230     user.requireUpperCase = requireUpperCase;
231     user.requireSpecialChar = requireSpecialChar;
232
233     // Calculate expiry date
234     if (expiryDays) {
235       user.passwordExpiry = new Date(
236         Date.now() + expiryDays * 24 * 60 * 60 * 1000
237       );
238     }
239
240     await user.save();
241     res.json({ message: "Safety settings updated successfully" });
242   } catch (error) {
243     console.error("Error in setSafetySettings:", error);
244     res.status(500).json({ message: "Server error" });
245   }
246 };
247
```

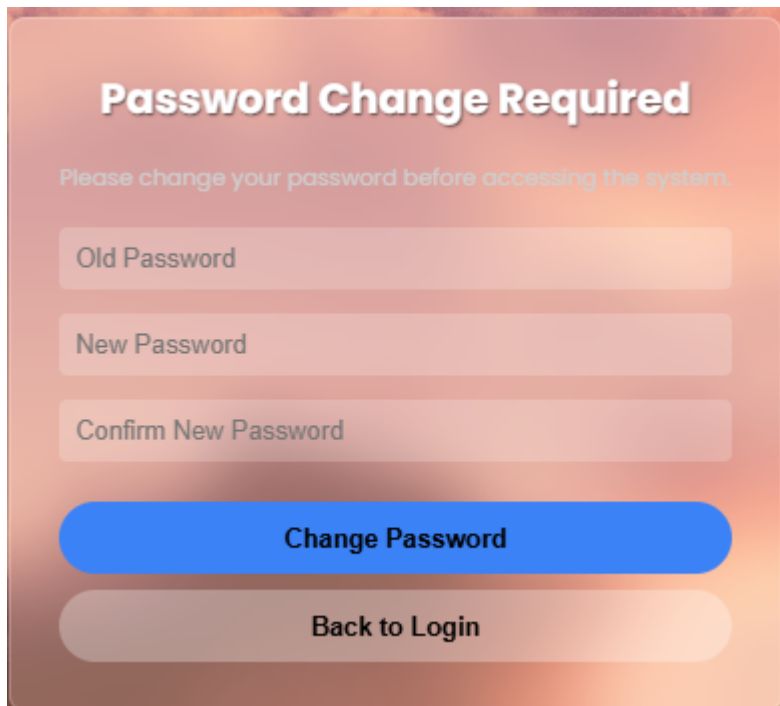
Tutaj natomiast mamy obsługę wygaszania ważności hasła:

```
192 // Set User Password Expiry
193 exports.setUserPasswordExpiry = async (req, res) => {
194   const { expiryDays } = req.body;
195   const userId = req.params.id;
196
197   try {
198     const user = await User.findById(userId);
199     if (!user) return res.status(404).json({ message: "User not found" });
200
201     user.passwordExpiry = new Date(
202       Date.now() + expiryDays * 24 * 60 * 60 * 1000
203     );
204     await user.save();
205     res.json({ message: `Password expiry set to ${expiryDays} days` });
206   } catch (error) {
207     console.error("Error in setUserPasswordExpiry:", error);
208     res.status(500).json({ message: "Server error" });
209   }
210 };
211
```

Możliwości Użytkownika:



Po wpisaniu loginu oraz hasła do naszej aplikacji przeniesiemy się do UserDashboard, w którym jedyną możliwością jest możliwość zmiany hasła.



Password Change Required

Please change your password before accessing the system.

Old Password

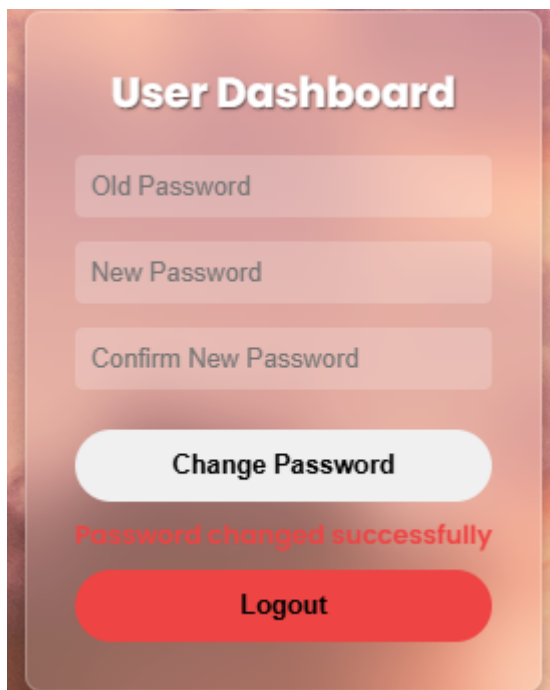
New Password

Confirm New Password

Change Password

Back to Login

Przy pierwszym logowaniu użytkownik jest zmuszony do zmiany obecnego hasła podanego przez administratora.



User Dashboard

Old Password

New Password

Confirm New Password

Change Password

Password changed successfully

Logout

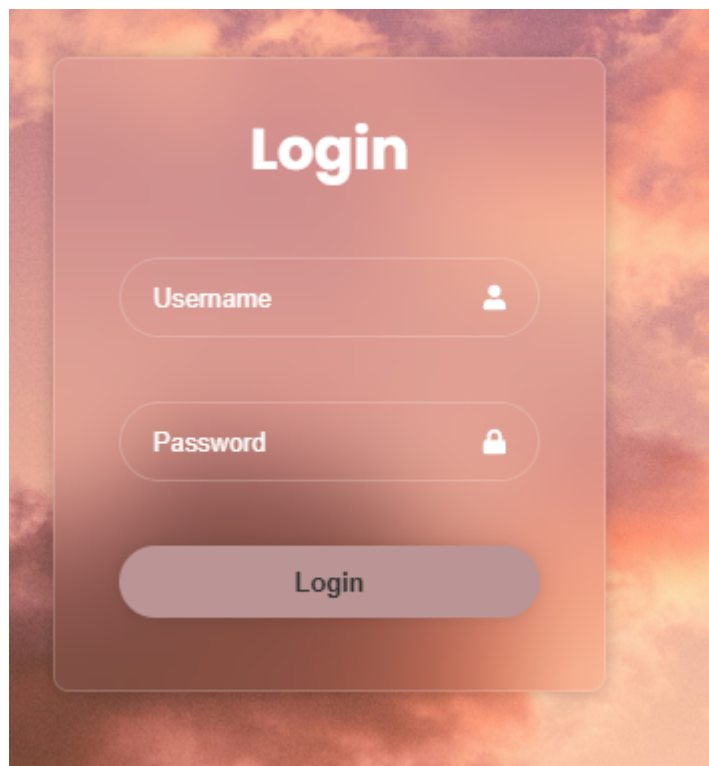
Tak wygląda UserDashboard

```
4 exports.changeUserPassword = async (req, res) => {
5   const { oldPassword, newPassword } = req.body;
6   const userId = req.user.id;
7
8   try {
9     const user = await User.findById(userId);
10    if (!user) {
11      return res.status(404).json({ message: "User not found" });
12    }
13
14    const isMatch = await user.matchPassword(oldPassword);
15    if (!isMatch) {
16      return res.status(400).json({ message: "Old password is incorrect" });
17    }
18
19    // Validate password requirements
20    const { requireUpperCase, requireLowerCase, requireSpecialChar } = user;
21    let passwordValid = true;
22    let errorMessage = "Password must contain:";
23
24    if (requireUpperCase && !/[A-Z]/.test(newPassword)) {
25      passwordValid = false;
26      errorMessage += " uppercase letter,";
27    }
28    if (requireLowerCase && !/[a-z]/.test(newPassword)) {
29      passwordValid = false;
30      errorMessage += " lowercase letter,";
31    }
32    if (requireSpecialChar && !/[!@#$$%^&*]/.test(newPassword)) {
33      passwordValid = false;
34      errorMessage += " special character,";
35    }
36
37    if (!passwordValid) {
38      return res.status(400).json({
39        message: errorMessage.slice(0, -1), // Remove comma
40      });
41    }
42
43    const salt = await bcrypt.genSalt(10);
44    user.password = await bcrypt.hash(newPassword, salt);
45    user.isFirstLogin = false; // Set first login to false after password change
46    await user.save();
47
48    res.json({ message: "Password changed successfully" });
49  } catch (error) {
50    console.error("Error in changeUserPassword:", error);
51    res.status(500).json({ message: "Server error" });
52  }
```

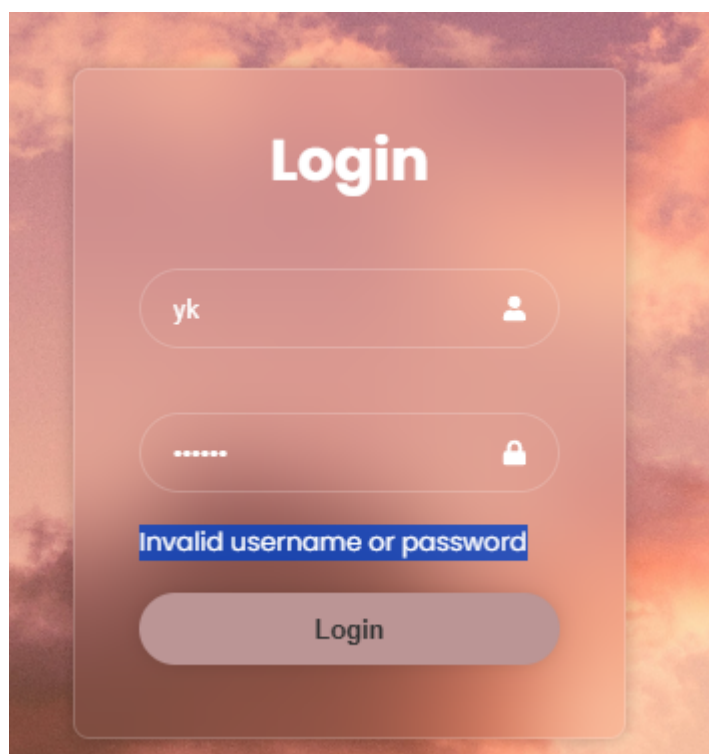
Powyżej znajduje się funkcja obsługująca zmianę hasła.

Program zawiera mechanizm logowania z weryfikacją poprawności identyfikatora i hasła.

Gdy uruchomimy naszą aplikację widzimy tylko ekran logowania:

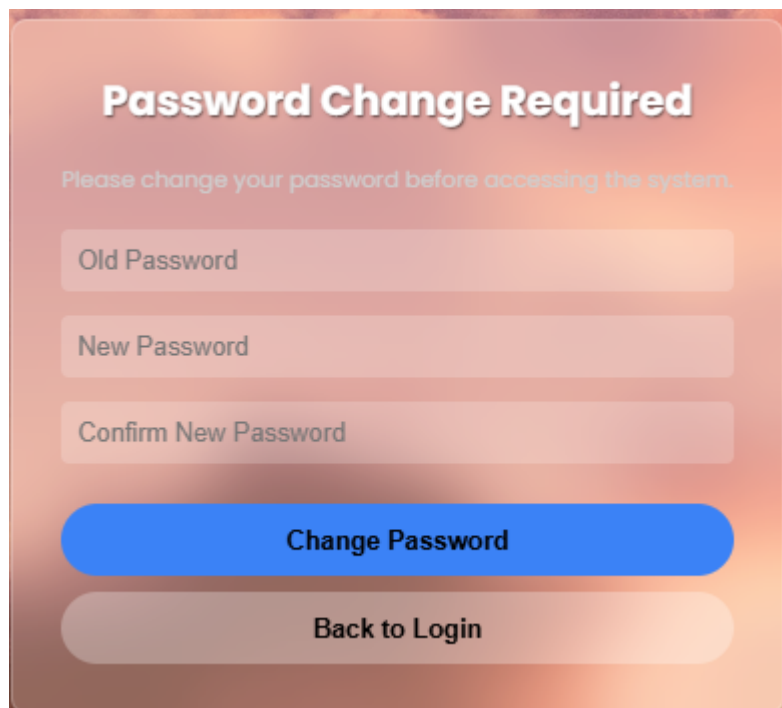


W przypadku podania złego hasła lub nazwy użytkownika program poinformuje nas o tym, że wpisaliśmy złe dane. Hasło jest zakryte, podczas wpisywania.



Przy pierwszym logowaniu użytkownik jest proszony o zmianę hasła.

Po wpisaniu loginu oraz hasła do naszej aplikacji przeniesiemy się do UserDashboard, w którym jedyną możliwością jest możliwość zmiany hasła.



Password Change Required

Please change your password before accessing the system.

Old Password

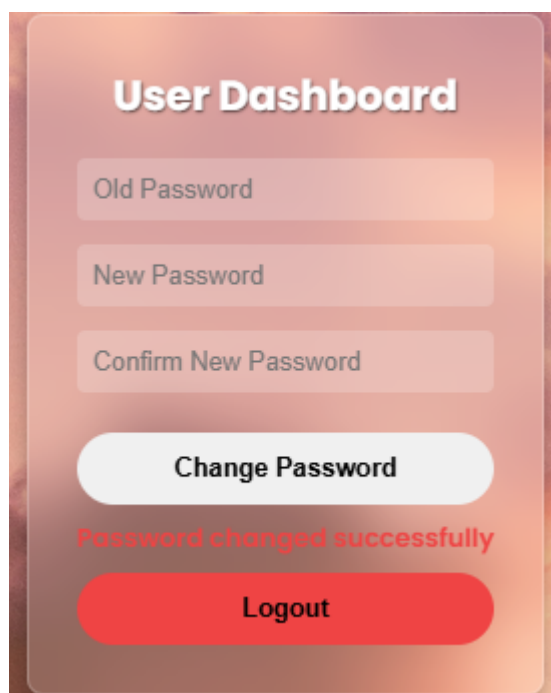
New Password

Confirm New Password

Change Password

Back to Login

Przy pierwszym logowaniu użytkownik jest zmuszony do zmiany obecnego hasła podanego przez administratora.



User Dashboard

Old Password

New Password

Confirm New Password

Change Password

Password changed successfully

Logout

Gdy zrobimy to poprawnie możemy przenieść się już do UserDashboard gdzie użytkownik może zmienić hasło.

Program korzysta z bezpiecznego algorytmu do hashowania haseł (bcrypt).

```
exports.changePassword = async (req, res) => {
  const { oldPassword, newPassword } = req.body;
  const userId = req.user.id; // Extracted from JWT token

  try {
    const user = await User.findById(userId);

    const isMatch = await user.matchPassword(oldPassword);
    if (!isMatch) {
      return res.status(400).json({ message: "Old password is incorrect" });
    }

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

    await user.save();
    res.json({ message: "Password updated successfully" });
  } catch (error) {
    res.status(500).json({ message: "Server error" });
  }
};
```

Używanie bcrypt.

security.users

4 DOCUMENTS 2 INDEXES

Documents Aggregations Schema Indexes Validation

Filter ⓘ ⓘ Type a query: { field: 'value' } or [Generate query](#) ⚡ Explain Reset Find </> Options ▶

ADD DATA EXPORT DATA UPDATE DELETE 1 - 4 of 4

```
{
  "_id": ObjectId('671f61bf7941873c6165dbb0'),
  "username": "John",
  "password": "$2a$10$FumqYZctF5WalMaUui0B7ecSg/h1Xn5gB13sl/CsyDHqQX6tzdhq",
  "role": "user",
  "fullName": "John Smith",
  "passwordExpiry": null,
  "blocked": false,
  "requireUpperCase": false,
  "requireLowerCase": false,
  "requireSpecialChar": false,
  "isFirstLogin": true,
  "__v": 0
}
```

```
{
  "_id": ObjectId('671f621b7941873c6165dbb8'),
  "username": "Kris",
  "password": "$2a$10$fg3g3fH.WRFGJuHvubQZseBYPshHZHbcHK6r2PzMPcVepyT3CVs7G",
  "role": "user",
  "fullName": "Kris Noman",
  "passwordExpiry": null,
  "blocked": false,
  "requireUpperCase": false,
  "requireLowerCase": false,
  "requireSpecialChar": false,
  "isFirstLogin": true
}
```

Po wejściu w bazę danych możemy zobaczyć, że dzięki temu algorytmowi nasze hasła użytkowników są bezpiecznie zaszyfrowane.

System musi weryfikować spełnianie indywidualnych zasad dotyczących hasła, w moim przypadku są to znaki z kategorii: wielkie litery, małe litery, oraz znaki specjalne (sekcja 9).

```
5  const SafetySettingsForm = ({ user, onBack, fetchUsers }) => {
6    const [requireUpperCase, setRequireUpperCase] = useState(false);
7    const [requireLowerCase, setRequireLowerCase] = useState(false);
8    const [requireSpecialChar, setRequireSpecialChar] = useState(false);
9    const [expiryDays, setExpiryDays] = useState(30);
10   const [error, setError] = useState('');
11
12   useEffect(() => {
13     if (user) {
14       setRequireUpperCase(user.requireUpperCase);
15       setRequireLowerCase(user.requireLowerCase);
16       setRequireSpecialChar(user.requireSpecialChar);
17
18       if (user.passwordExpiry) {
19         const expiryDate = new Date(user.passwordExpiry);
20         const today = new Date();
21         const remainingDays = Math.ceil((expiryDate - today) / (1000 * 60 * 60 * 24));
22         setExpiryDays(Math.max(remainingDays, 1));
23       }
24     }
25   }, [user]);
26
27   const handleSubmit = async (e) => {
28     e.preventDefault();
29     setError('');
30
31     if (!user?._id) {
32       setError('User ID not found. Please try again.');
```

Formularz SafetySettings został przedstawiony wcześniej, zarówno jego możliwości, jak i kod od strony backendu, w związku z tym, tutaj znajduje się kod do frontendu.


```
5  const SafetySettingsForm = ({ user, onBack, fetchUsers }) => {
27    const handleSubmit = async (e) => {
30      // ...
51    }
52  };
53
54    alert('Password requirements and expiry updated successfully');
55    await fetchUsers();
56    onBack();
57  } catch (err) {
58    setError(err.response?.data?.message || 'Error updating password requirements');
59  }
60 };
61
62  const handleExpiryChange = (e) => {
63    const value = parseInt(e.target.value) || 1; // Default to 1 to avoid zero or negative values
64    setExpiryDays(Math.max(1, value));
65  };
66
67  return (
68    <div className="safety-settings-container">
69      <h2>Password Requirements for {user?.fullName || 'User'}</h2>
70      <form onSubmit={handleSubmit}>
71        <div className="requirements-group">
72          <label>
73            <input
74              type="checkbox"
75              checked={requireUpperCase}
76              onChange={(e) => setRequireUpperCase(e.target.checked)}
77            />
78            Uppercase Letters (A-Z)
79          </label>
80
81          <label>
82            <input
83              type="checkbox"
84              checked={requireLowerCase}
85              onChange={(e) => setRequireLowerCase(e.target.checked)}
86            />
87            Lowercase Letters (a-z)
88          </label>
89
90          <label>
91            <input
92              type="checkbox"
93              checked={requireSpecialChar}
94              onChange={(e) => setRequireSpecialChar(e.target.checked)}
95            />
96            Special Characters (!@#$%^&*)
```

Część dalsza kodu.

3.2. Zarządzanie sesjami

W celu zapewnienia bezpieczeństwa i wygody użytkownika, aplikacja wykorzystuje mechanizmy zarządzania sesjami oparte na tokenach JWT, które są zapisywane w ciasteczkach. Administratorzy mają dostęp do panelu, w którym mogą zarządzać kontami użytkowników oraz polityką haseł. W ten sposób prezentuje się to od strony kodu:

```
33 // Generate a JWT token if the credentials are valid
34 const token = jwt.sign(
35   { id: user._id, role: user.role },
36   process.env.JWT_SECRET,
37   { expiresIn: "1h" }
38 );
39
40 // Return the token and role in the response
41 res.json({ token, role: user.role });
42 } catch (error) {
43   console.error("Login error:", error);
44   res.status(500).json({ message: "Server error" });
45 }
46 });
47
48 // Token verification route
49 router.get("/verify", async (req, res) => {
50   const token = req.headers.authorization?.split(" ")[1];
51   if (!token) {
52     return res.status(401).json({ message: "No token provided" });
53   }
54
55   try {
56     // Verify the token
57     const decoded = jwt.verify(token, process.env.JWT_SECRET);
58
59     // Find the user by ID extracted from the token
60     const user = await User.findById(decoded.id);
61     if (!user) {
62       return res.status(401).json({ message: "User not found" });
63     }
64
65     // Check if the user is blocked
66     if (user.blocked) {
67       return res.status(403).json({
68         message: "Your account is blocked, please contact the administrator",
69       });
70     }
71
72     // Return the user's role
73     res.json({ role: user.role });
74   } catch (error) {
75     console.error("Token verification error:", error);
76     res.status(401).json({ message: "Invalid token" });
77   }
78 }
```

4. Podsumowanie

Opracowany program spełnia wszystkie wymagania projektu. Umożliwia efektywne zarządzanie kontami użytkowników, zapewnia weryfikację haseł zgodnie z ustalonymi standardami bezpieczeństwa oraz obsługę różnych ról w systemie. Wdrożenie tego rozwiązania w środowisku produkcyjnym mogłoby znacząco zwiększyć poziom bezpieczeństwa danych, dzięki rygorystycznym zasadom uwierzytelniania użytkowników.