Zaawansowany Programista JS

Dzień 10

Plan

- Tworzenie wielu stron za pomocą React Router
- Kompozycja aplikacji Reactowej przy uzyciu stron
- Globalny stan przy uzyciu `React.Context`
- Poznanie `Firebase auth`
- Next steps: `Next.JS`

Tworzenie wielu stron za pomocą React Router

React Router to biblioteka, która umożliwia nawigację między widokami w aplikacji React.JS. Biblioteka ta dostarcza zestaw komponentów, które pozwalają na definiowanie ścieżek URL, zarządzanie historią przeglądania i tworzenie nawigacji między stronami.

React Router umożliwia tworzenie aplikacji z jednostronicowym interfejsem użytkownika (SPA), w których treść strony jest dynamicznie ładowana bez przeładowywania całej strony. Biblioteka ta działa w oparciu o deklaratywny sposób programowania, co oznacza, że można definiować ścieżki URL i przypisywać do nich komponenty w sposób podobny do tego, jak deklaruje się elementy interfejsu w React.JS.

React Router jest często używany w połączeniu z Create React App i Firebase, aby tworzyć zaawansowane aplikacje internetowe z interfejsem użytkownika opartym na jednej stronie.

Aby użyć React Router w Create React App, należy wykonać następujące kroki:

Zainstaluj React Router przy użyciu polecenia `npm install react-router-dom`.

1. W pliku `App.js` zaimportuj komponenty `BrowserRouter`, `Route` i `Switch` z biblioteki `React Router`.

```
import { BrowserRouter, Route, Switch } from 'react-router-dom';


function App() {
  return (
    <BrowserRouter>
      <div className="App">
      // treść aplikacji
```

```
      </div>
    </BrowserRouter>
  );
}
```

2. Definiuj ścieżki URL w komponentach Route. Wszystkie elementy wewnątrz Switch są renderowane tylko raz i tylko jeden z nich jest renderowany na raz, a elementy Route są renderowane tylko wtedy, gdy ścieżka URL pasuje.

```
function App() {
  return (
    <BrowserRouter>
      <div className="App">
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
        </Switch>
      </div>
    </BrowserRouter>
  );
}
```

3. Stwórz komponenty dla każdej strony w twojej aplikacji. Komponenty te będą przekazywane jako właściwości (props) do elementów Route. Na przykład:

```
function Home() {
  return (
    <div>
      <h1>Strona główna</h1>
      <p>Witaj na stronie głównej!</p>
    </div>
  );
}

function About() {
  return (
    <div>
      <h1>O nas</h1>
      <p>Tutaj dowiesz się więcej o naszej firmie.</p>
    </div>
  );
}

function Contact() {
  return (
    <div>
      <h1>Kontakt</h1>
      <p>Skontaktuj się z nami za pomocą formularza kontaktowego.</p>
    </div>
  );
}
```

4. Stwórz nawigację między stronami za pomocą elementu Link z biblioteki React Router. Na przykład:

```
import { Link } from 'react-router-dom';

function Navigation() {
  return (
    <nav>
      <ul>
        <li><Link to="/">Strona główna</Link></li>
        <li><Link to="/about">O nas</Link></li>
        <li><Link to="/contact">Kontakt</Link></li>
      </ul>
    </nav>
  );
}
```

5. Wyeksportuj swoje komponenty i skorzystaj z nich w swojej aplikacji. Na przykład:

```
function App() {
  return (
    <BrowserRouter>
      <div className="App">
        <Navigation />
        <Switch>
          <Route exact path="/" component={Home} />
          <Route path="/about" component={About} />
          <Route path="/contact" component={Contact} />
        </Switch>
      </div>
    </BrowserRouter>
  );
}

export default App;
```

Kompozycja aplikacji Reactowej przy uzyciu stron

Struktura folderów w Create React App zależy od preferencji programisty, ale zwykle zaleca się organizowanie projektu w następujący sposób, jeśli korzystamy z React Router:

1. w katalogu src tworzymy folder components, w którym umieszczamy wszystkie komponenty
2. w katalogu src tworzymy folder pages, w którym umieszczamy komponenty, które odpowiadają za poszczególne strony aplikacji
3. w katalogu src tworzymy plik App.js, który odpowiada za renderowanie aplikacji i zawiera główny routing
4. w katalogu src tworzymy plik index.js, który odpowiada za renderowanie komponentu App do drzewa DOM

Ponadto, w pliku App.js powinniśmy zaimportować komponenty BrowserRouter i Switch z react-router-dom. Dzięki temu będziemy mogli definiować ścieżki do naszych stron i renderować odpowiednie komponenty dla danego URL-a.

Globalny stan przy uzyciu `React.Context`

React Context to mechanizm, który umożliwia przekazywanie danych pomiędzy komponentami w hierarchii drzewa komponentów bez konieczności przekazywania ich przez propsy. Pozwala to na uproszczenie struktury kodu oraz uniknięcie tzw. "prop drilling" - przekazywania propsów przez wiele poziomów komponentów.

React Context składa się z dwóch elementów: Provider i Consumer. Provider jest odpowiedzialny za udostępnianie danych, natomiast Consumer za ich odczytywanie.

Aby korzystać z React Context, należy utworzyć nowy kontekst za pomocą funkcji React.createContext(). Następnie, wewnątrz komponentu, w którym chcemy udostępnić dane, tworzymy Provider i przekazujemy mu te dane. Odczytywanie danych odbywa się poprzez utworzenie Consumera w komponencie, który potrzebuje tych danych.

Przykładowo, jeśli chcemy udostępnić informację o zalogowanym użytkowniku w naszej aplikacji, możemy stworzyć nowy kontekst za pomocą funkcji React.createContext(), a następnie wewnątrz komponentu App utworzyć Provider i przekazać informację o zalogowanym użytkowniku. W innych komponentach, które potrzebują tej informacji, możemy utworzyć Consumer i odczytać te dane.

Oto prosty przykład użycia React.Context w aplikacji stworzonej przy użyciu Create React App.

1. Najpierw definiujemy kontekst, w którym chcemy przechowywać informacje. W tym przypadku jest to kontekst użytkownika, który będzie przechowywał informacje o zalogowanym użytkowniku:

```
import { createContext } from 'react';

export const UserContext = createContext(null);
```

2. Następnie, w najwyższym komponencie aplikacji (np. w komponencie App), możemy przekazać wartości do kontekstu poprzez użycie UserContext.Provider:

```
import React, { useState } from 'react';
import { UserContext } from './UserContext';
import { Header } from './Header';
import { Main } from './Main';

function App() {
  const [user, setUser] = useState(null);

  return (
    <UserContext.Provider value={{ user, setUser }}>
      <div className="App">
        <Header />
        <Main />
      </div>
    </UserContext.Provider>
```

```
  );
}

export default App;
```

W powyższym przykładzie wartość kontekstu jest przekazywana poprzez `value={{ user, setUser }}`. W ten sposób każdy komponent, który jest "dzieckiem" `UserContext.Provider` może uzyskać dostęp do wartości kontekstu poprzez użycie `useContext`:

```
import React, { useContext } from 'react';
import { UserContext } from './UserContext';

export const Header = () => {
  const { user } = useContext(UserContext);

  return (
    <header>
    {user
      ? <h2>Welcome, {user}!</h2>
      : <h2>Please log in.</h2>
    }
    </header>
  );
};
```

W powyższym przykładzie, Header uzyskuje dostęp do wartości kontekstu UserContext poprzez użycie `useContext(UserContext)`. W ten sposób, jeśli wartość user jest ustawiona, wyświetlany jest komunikat powitalny, a w przeciwnym razie użytkownik jest proszony o zalogowanie się.

Poznanie `Firebase auth`

Firebase Authentication to usługa autoryzacji i uwierzytelniania użytkowników, która jest oferowana przez Firebase - platformę do budowania aplikacji mobilnych i webowych. Firebase Authentication umożliwia deweloperom łatwe i szybkie dodanie funkcjonalności uwierzytelniania do ich aplikacji, dzięki czemu użytkownicy mogą logować się i korzystać z aplikacji za pomocą różnych metod uwierzytelniania, takich jak adres e-mail i hasło, konta Google, Facebooka, Twittera, Apple ID itp. Firebase Authentication zapewnia również zabezpieczenie danych użytkowników poprzez zarządzanie dostępem do zasobów aplikacji w zależności od poziomu uprawnień użytkownika.

Aby użyć Firebase Authentication i React.Context w Create React App, należy wykonać następujące kroki:

1. Zainstaluj bibliotekę firebase i react-firebase-hooks przy użyciu npm:

```
npm install firebase react-firebase-hooks
```

2. Zainicjuj Firebase w pliku `src/firebase.js`:

```
import firebase from "firebase/app";
import "firebase/auth";
```

```
const firebaseConfig = {
// wklej tutaj konfigurację Firebase
};

firebase.initializeApp(firebaseConfig);

export const auth = firebase.auth();
```

3. Utwórz plik `src/AuthProvider.js`, w którym będzie zawarta logika uwierzytelniania i kontekstu:

```
import React, { createContext, useEffect, useState } from "react";
import { auth } from "./firebase";
import { useAuthState } from "react-firebase-hooks/auth";

export const AuthContext = createContext();

export const AuthProvider = ({ children }) => {
  const [user, loading, error] = useAuthState(auth);
  const [authUser, setAuthUser] = useState(null);

  useEffect(() => {
    if (user) {
      setAuthUser({ uid: user.uid, email: user.email });
    } else {
      setAuthUser(null);
    }
  }, [user]);

  if (loading) {
    return <div>Loading...</div>;
  }

  if (error) {
    return <div>Error: {error}</div>;
  }

  return (
    <AuthContext.Provider value={authUser}>{children}</AuthContext.Provider>
  );
};
```

4. Zaimplementuj komponenty `SignIn` i `SignOut`:

```
import React, { useContext } from "react";
import { auth } from "./firebase";
import { AuthContext } from "./AuthProvider";

const SignIn = () => {
  const signInWithGoogle = () => {
    const provider = new firebase.auth.GoogleAuthProvider();
```

```
    auth.signInWithPopup(provider);
  };

  return (
    <div>
      <button onClick={signInWithGoogle}>Sign in with Google</button>
    </div>
  );
};

const SignOut = () => {
  const authUser = useContext(AuthContext);

  return (
    authUser && (
      <div>
        <p>Hello, {authUser.email}</p>
        <button onClick={() => auth.signOut()}>Sign Out</button>
      </div>
    )
  );
};

export { SignIn, SignOut };
```

5. Ostatecznie, zaimplementuj routing w App.js:

```
import React from "react";
import { BrowserRouter as Router, Switch, Route } from "react-router-dom";
import { AuthProvider } from "./AuthProvider";
import { SignIn, SignOut } from "./Auth";

function App() {
  return (
    <AuthProvider>
      <Router>
        <Switch>
          <Route exact path="/" component={SignIn} />
          <Route path="/home" component={SignOut} />
        </Switch>
      </Router>
    </AuthProvider>
  );
}

export default App;
```

Dzięki powyższym krokom, stworzono aplikację, w której można się zalogować i wylogować przy użyciu Firebase Authentication, a dane użytkownika są przechowywane w kontekście React.Context.

Next steps: `Next.JS`

Next.js to framework open source dla React, który umożliwia łatwe tworzenie aplikacji internetowych SSR (Server Side Rendering) i SSG (Static Site Generation). Next.js dodaje warstwę serwerową do aplikacji React, dzięki czemu umożliwia wyświetlanie zawartości na serwerze zamiast jedynie w przeglądarce klienta, co pozwala na szybsze ładowanie strony, lepsze SEO i zoptymalizowanie czasu wczytywania. Ponadto, Next.js posiada wiele narzędzi, takich jak automatyczne generowanie ścieżek i współpracę z plikami statycznymi, dzięki czemu ułatwia tworzenie kompletnych aplikacji internetowych. Next.js jest polecany dla projektów, które wymagają wysokiej wydajności i zoptymalizowania czasu ładowania, a także dla projektów, w których ważne jest dobre SEO i responsywność na różnych urządzeniach.

Jest to natomiast temat dość zaawansowany, dlatego jest polecany jako następny krok zaraz po zapoznaniu się z React.JS.