Zaawansowany Przewodnik po React.js

Kacper Renkel

August 11, 2024

Contents

1 Wprowadzenie

React to biblioteka JavaScript służaca do tworzenia dynamicznych interfejsów użytkownika. W tym dokumencie omówimy zaawansowane aspekty Reacta, w tym szczegółowy opis hooków, serwisy, interceptory, zarzadzanie stanem i routing.

2 Komponenty

Komponenty sa podstawowymi elementami Reacta. Możemy je tworzyć zarówno jako funkcjonalne komponenty, jak i komponenty klasowe.

2.1 Funkcyjne komponenty

Funkcyjne komponenty sa prostymi funkcjami, które przyjmuja `props` i zwracaja elementy Reacta.

```
import React from 'react';

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

export default Welcome;
```

Listing 1: Prosty funkcjonalny komponent

2.2 Komponenty klasowe

Komponenty klasowe oferuja bardziej rozbudowana funkcjonalność, taka jak zarzadzanie stanem.

```
import React, { Component } from 'react';

class Welcome extends Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}

export default Welcome;
```

Listing 2: Komponent klasowy

3 Hooki w React

Hooki to funkcje, które pozwalaja na "zaczepienie" sie w wewnetrzne mechanizmy Reacta, takie jak stan, cykl życia komponentu, itp.

3.1 useState

`useState` to hook, który umożliwia zarzadzanie stanem w funkcjonalnych komponentach.

```
import React, { useState } from 'react';

function Counter() {
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>Click me</
         button>
    </div>
  );
}

export default Counter;
```

Listing 3: Przykład użycia `useState`

3.2 useEffect

`useEffect` pozwala na wykonywanie efektów ubocznych, takich jak pobieranie danych z API, po każdym renderowaniu komponentu.

```
import React, { useState, useEffect } from 'react';

function DataFetcher({ url }) {
  const [data, setData] = useState(null);

  useEffect(() => {
    fetch(url)
      .then(response => response.json())
      .then(data => setData(data));
  }, [url]);

  return (
    <div>
      <pre>{JSON.stringify(data, null, 2)}</pre>
    </div>
  );
}

export default DataFetcher;
```

Listing 4: Przykład użycia `useEffect`

3.3 useContext

`useContext` pozwala na korzystanie z kontekstu Reacta w komponentach funkcyjnych.

```
import React, { useContext } from 'react';

const ThemeContext = React.createContext('light');

function ThemedButton() {
  const theme = useContext(ThemeContext);
  return <button className={theme}>I am styled by theme
     context!</button>;
}

export default ThemedButton;
```

Listing 5: Przykład użycia `useContext`

3.4 useReducer

`useReducer` to zaawansowany hook do zarzadzania stanem komponentu, podobny do `useState`, ale umożliwiajacy bardziej złożona logike aktualizacji stanu.

```
import React, { useReducer } from 'react';

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return { count: state.count + 1 };
    case 'decrement':
      return { count: state.count - 1 };
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, { count: 0 })
     ;

  return (
    <div>
      <p>Count: {state.count}</p>
      <button onClick={() => dispatch({ type: 'increment' })
         }>+</button>
      <button onClick={() => dispatch({ type: 'decrement' })
         }>-</button>
    </div>
  );
}
```

```
export default Counter;
```

Listing 6: Przykład użycia `useReducer`

3.5 useRef

`useRef` pozwala na tworzenie referencji do elementów DOM lub przechowywanie zmiennych, które nie powoduja ponownego renderowania komponentu przy zmianie wartości.

```
import React, { useRef } from 'react';

function FocusableInput() {
  const inputRef = useRef(null);

  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus the input</button>
    </div>
  );
}

export default FocusableInput;
```

Listing 7: Przykład użycia `useRef`

3.6 useMemo

`useMemo` pozwala na memoizacje wartości, która zostanie ponownie obliczona tylko wtedy, gdy zależności sie zmienia, co jest przydatne do optymalizacji wydajności.

```
import React, { useMemo } from 'react';

function ExpensiveCalculationComponent({ num }) {
  const calculation = useMemo(() => {
    return expensiveCalculation(num);
  }, [num]);

  return <div>Result: {calculation}</div>;
}

function expensiveCalculation(num) {
  // skomplikowana operacja
  return num * 2;
```

```
}

export default ExpensiveCalculationComponent;
```

Listing 8: Przykład użycia `useMemo`

3.7 useCallback

`useCallback` memoizuje funkcje, zapobiegajac ich ponownemu tworzeniu podczas każdego renderowania, co jest przydatne, gdy przekazujemy funkcje do komponentów zależnych.

```
import React, { useState, useCallback } from 'react';

function ParentComponent() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount(count + 1);
  }, [count]);

  return (
    <div>
      <p>Count: {count}</p>
      <ChildComponent onIncrement={increment} />
    </div>
  );
}

function ChildComponent({ onIncrement }) {
  return <button onClick={onIncrement}>Increment</button>;
}

export default ParentComponent;
```

Listing 9: Przykład użycia `useCallback`

4 Zaawansowane zarzadzanie stanem

Poza podstawowymi hookami, takimi jak `useState` i `useReducer`, React oferuje narzedzia takie jak Context API oraz biblioteki zewnetrzne, jak Redux, do zarzadzania globalnym stanem aplikacji.

4.1 Context API

Context API pozwala na przekazywanie danych przez drzewo komponentów bez konieczności recznego przekazywania `props` na każdym poziomie.

```
import React, { createContext, useContext, useState } from '
   react';
```

```
const CountContext = createContext();

function CounterProvider({ children }) {
  const [count, setCount] = useState(0);
  return (
    <CountContext.Provider value={{ count, setCount }}>
      {children}
    </CountContext.Provider>
  );
}

function CounterDisplay() {
  const { count } = useContext(CountContext);
  return <div>Count: {count}</div>;
}

function IncrementButton() {
  const { setCount } = useContext(CountContext);
  return <button onClick={() => setCount(count => count + 1)
     }>Increment</button>;
}

function App() {
  return (
    <CounterProvider>
      <CounterDisplay />
      <IncrementButton />
    </CounterProvider>
  );
}

export default App;
```

Listing 10: Przykład użycia Context API

4.2 Redux

Redux to popularna biblioteka do zarzadzania stanem aplikacji. Stan aplikacji jest przechowywany w jednym miejscu zwanym `store`, a zmiany w stanie sa realizowane przez `actions` i `reducers`.

5 Routing

Routing w React można zrealizować za pomoca biblioteki `react-router`, która pozwala na tworzenie nawigacji miedzy różnymi widokami aplikacji.

```
import React from 'react';
```

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

function Home() {
  return <h2>Home</h2>;
}

function About() {
  return <h2>About</h2>;
}

function App() {
  return (
    <Router>
      <Switch>
        <Route path="/about">
          <About />
        </Route>
        <Route path="/">
          <Home />
        </Route>
      </Switch>
    </Router>
  );
}

export default App;
```

Listing 11: Przykład użycia `react-router`

6 Serwisy

Serwisy w React moga być tworzone jako osobne moduły do zarzadzania logika aplikacji, szczególnie w zakresie interakcji z API.

```
class ApiService {
  static async fetchData(url) {
    const response = await fetch(url);
    if (!response.ok) {
      throw new Error('Failed to fetch data');
    }
    return response.json();
  }
}

export default ApiService;
```

Listing 12: Przykład serwisu API

7 Interceptors

Interceptors moga być używane do przechwytywania żadań lub odpowiedzi HTTP w celu dodania logiki przed lub po wysłaniu żadania.

```
import axios from 'axios';

axios.interceptors.request.use(request => {
  // Dodaj nag  wki , tokeny itp.
  return request;
});

axios.interceptors.response.use(response => {
  // Przetwarzaj odpowiedzi , b   dy itp.
  return response;
}, error => {
  // Obs uga b   d  w
  return Promise.reject(error);
});
```

Listing 13: Przykład użycia Interceptorów w Axios

8 Podsumowanie

W tym dokumencie omówiliśmy zaawansowane aspekty tworzenia aplikacji w React, takie jak użycie hooków, serwisów, interceptorów oraz zarzadzanie stanem. React oferuje elastyczne narzedzia do budowania złożonych interfejsów użytkownika, co czyni go jednym z najpopularniejszych wyborów wśród deweloperów front-endu.