Sposoby zarządzania stanem komponentów w aplikacji oraz komunikacja między komponentami w ReactJS.

1. Props drilling.

Props drilling polega na przesyłanie stanu w dół drzewa komponentów, za pomocą odpowiednich właściwości (props). Komponenty potomne odbierają te właściwości, a programista może się do nich odwołać wykorzystując **props.nazwaWłaściwości** .

Zaletą tego rozwiązania jest prostota. W praktyce props-drilling przypomina przekazywanie kaskadowo, odpowiednich wartości do przygotowanych wcześniej funkcji (tutaj komponentów). Dla niewielkich drzew komponentów jest to rozwiązanie o czytelnej architekturze i małej zawiłości wyprodukowanego kodu.

Z props-drilling'u należy zdecydowanie zrezygnować w sytuacji, gdy spodziewamy się przekazywania danych, przez duże ilości warstw drzewa komponentów, wynoszenia stanu w górę (callback do rodzica), bądź wielu komponentów potomnych.

Poniższy przykład prezentuje przykładowe wykorzystanie props-drillingu.

Komponent TrafficLight jest najwyżej w hierarchii drzewa komponentów i przechowuje on stan aktualnie wybranego światła. Poprzez "propsy" przekazywane są funkcje zmieniające wybrany kolor oraz cyfra opisująca wybrany kolor.

```
class TrafficLight extends React.Component {
constructor (props) {
  super (props);
  this.state = {
    // 0 - red, 1 - orange, 2- green
    currentColorNumber: 0,
  };
  this.incrementColor = this.incrementColor.bind (this);
  this.decrementColor = this.decrementColor.bind (this);
}

incrementColor () {
  if (this.state.currentColorNumber + 1 < 3)
    this.setState (state => ({
      currentColorNumber: state.currentColorNumber + 1,
    }));
}

decrementColor () {
  if (this.state.currentColorNumber > 0)
    this.setState (state => ({
      currentColorNumber: state.currentColorNumber - 1,
    }));
}

render () {
  return (
    <div>
      <Light color={this.state.currentColorNumber} />
      <LightsController
        increment={this.incrementColor}
        decrement={this.decrementColor}
      />
    </div>
  );
}
}
```

Komponent Light odpowiada za zapalenie odpowiedniego światła na podstawie przekazanego przez props stanu.

```
const Light = props => {
  return (
    <div>
      {props.color === 0 &&
        <div><a style={{padding: '4px', backgroundColor: 'red'}} /></div>}
      {props.color === 1 &&
        <div><a style={{padding: '4px', backgroundColor: 'orange'}} /></div>}
      {props.color === 2 &&
        <div><a style={{padding: '4px', backgroundColor: 'green'}} /></div>}
    </div>
  );
};
```

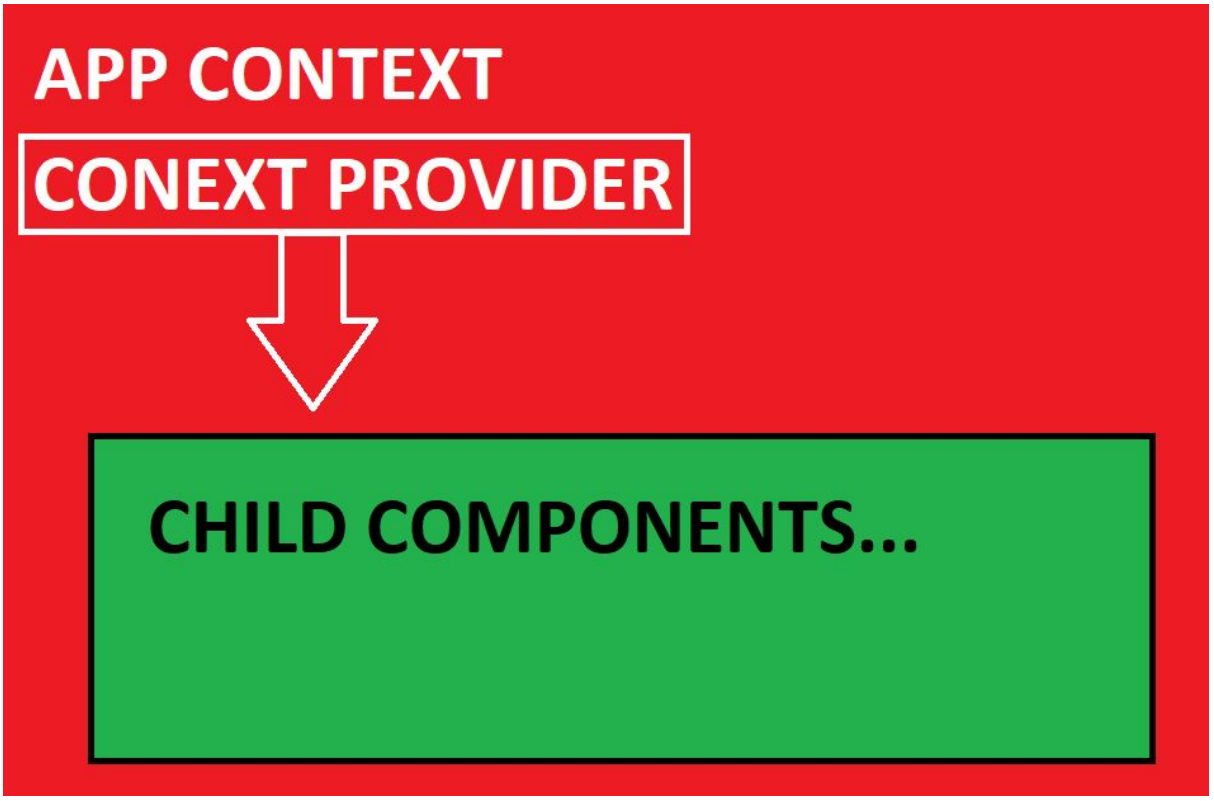
Natomiast komponent LightsController wywołuje metody zaimplementowane w klasie TrafficLight. Zbindowanie metod w komponencie TrafficLight pozwala na wykorzystanie ich, w celu zmiany stanu komponentu TrafficLight w innym komponencie. Jest to również przykład przekazywania funkcji przez props'y.

```
const LightsController = props => {
  return (
    <div>
      <button
        onClick={event => {
          event.preventDefault ();
          props.increment ();
        }}
      >
        INKREMENTUJ
      </button>
      <button
        onClick={event => {
          event.preventDefault ();
          props.decrement ();
        }}
      >
        DEKREMENTUJ
      </button>
    </div>
```

2. Context API.

Context API jest dostępne od wersji React v16.3.0. Jest to funkcjonalność wspierająca zarządzanie stanem komponentu oraz komunikację między komponentami.

Zamysł tej architektury przybliża poniższy schemat.



Elementy potomne danej funkcjonalności tworzone są w obszarze zasięgu jej kontekstu (kolor czerwony). Kontekst jest dostarczany do każdego potomka (Child Components kolor zielony) przez Context Provider. Przypomina to trochę korzystanie z globalnego magazynu (a nawet zmiennych globalnych), jednak kontekst można łatwo enkapsulować dostarczając go do wybranych komponentów potomnych.
Takie działanie nazywane jest owijaniem (ang. wrapping).

Kontekst tworzy się w sposób pokazany poniżej (createContext należy zaimportować z ‘react’).

```
const AppContext = createContext ();
```

Następnie należy stworzyć ContextProvider. Stan kontekstu (someData) po owinięciu jest dostępny w każdym elemencie potomnym.

```
const AppContextProvider = () => {
  // Stan w obszarze kontekstu
  const [someData, setSomeData] = useState ();

  return (
    <div>
      {/* Owijanie potomków przez Conext */}
      <AppContext.Provider>
        <ChildComponents />
      </AppContext.Provider>
    </div>
  );
};
```

Poniżej znajduje się przykład odbierania stanu w komponencie potomnym.

```
const ChildComponent = () => {
  const {someData, setSomeData} = useContext (AppContext);
};
```