## 8.1. Czym jest Redux?

W dokumentacji bibliotekę tę opisano jako „przewidywalny kontener stanu dla aplikacji JavaScriptu". Redux jest zasadniczo globalnym obiektem stanu, który jest jedynym źródłem prawdy w aplikacji. Ten globalny obiekt stanu jest pobierany jako właściwości do komponentów React Native. Za każdym razem, gdy zmienią się dowolne dane w stanie Reduksa, cała aplikacja otrzymuje te nowe dane jako właściwości.

Redux upraszcza stan aplikacji, przenosząc ją w jedno miejsce zwane *magazynem* (*store*)[1]; znacznie ułatwia to wnioskowanie i rozumienie. Gdy będziemy potrzebować jakiejś wartości, będziemy dokładnie wiedzieli, gdzie jej szukać, oraz możemy oczekiwać, że ta sama wartość będzie dostępna i aktualna również w innych miejscach aplikacji.

Jak działa Redux? Wykorzystuje funkcjonalność React zwaną *kontekstem*, czyli mechanizm tworzenia stanu globalnego i zarządzania nim.

## 8.2. Użycie kontekstu do tworzenia stanu globalnego i zarządzania nim w aplikacji React

Kontekstem jest API Reacta, tworzący zmienne globalne, do których można uzyskać dostęp w dowolnym miejscu aplikacji, pod warunkiem że komponent pobierający kontekst jest potomkiem komponentu, który go utworzył. Normalnie można by to zrobić, przekazując właściwości w dół przez poszczególne poziomy struktury komponentów. Mając kontekst, nie trzeba używać właściwości. Można wykorzystać kontekst w dowolnym miejscu w aplikacji i uzyskać do niego dostęp bez przekazywania go przez poszczególne poziomy.

> **UWAGA** Chociaż kontekst łatwo zrozumieć oraz jest on używany w wielu bibliotekach open source, prawdopodobnie nie trzeba będzie go używać w każdej aplikacji, chyba że tworzymy bibliotekę open source lub nie można znaleźć innego rozwiązania problemu. Omawiamy to tutaj, aby można było w pełni zrozumieć, jak w rzeczywistości działa Redux.

Zobaczmy, jak utworzyć kontekst w prostej strukturze składającej się z trzech komponentów: `Parent`, `Child1` i `Child2`. Przykład ten pokazuje, jak z poziomu nadrzędnego przekazać motyw, co w razie potrzeby może umożliwić nadanie stylu całej aplikacji.

**Listing 8.1. Tworzenie kontekstu**

```
const ThemeContext = React.createContext()

class Parent extends Component {
  state = { themeValue: 'light' }
  toggleThemeValue = () => {
    const value = this.state.themeValue === 'dark' ? 'light' : 'dark'
    this.setState({ themeValue: value })
  }
```

**Utworzenie nowej zmiennej o nazwie ThemeContext**

**Utworzenie zmiennej stanu themeValue o wartości 'light'**

**Sprawdzenie bieżącej wartości themeValue i przełączenie jej na 'light' lub 'dark'**

[1] Obiekt zarządzający stanem aplikacji (przyp. tłumacza).

```
  render() {
    return (
      <ThemeContext.Provider
        value={{
          themeValue: this.state.themeValue,
          toggleThemeValue: this.toggleThemeValue
        }}
      >
  <View style={styles.container}>
          <Text>Hello World</Text>
        </View>
        <Child1 />
      </ThemeContext.Provider>
    );
  }
}

const Child1 = () => <Child2 />

const Child2 = () => (
  <ThemeContext.Consumer>
    {(val) => (
        <View style={[styles.container,
                      val.themeValue === 'dark' &&
                     { backgroundColor: 'black' }]}>
          <Text style={styles.text}>Hello from Component2</Text>
          <Text style={styles.text}
                onPress={val.toggleThemeValue}>
              Toggle Theme Value
          </Text>
        </View>
      )}
  </ThemeContext.Consumer>
)

const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  text: {
    fontSize: 22,
    color: '#666'
  }
})
```

**Dostarczenie kontekstu komponentom potomnym. Cokolwiek, co będzie opakowane w Provider, będzie dostępne dla komponentów potomnych w Consumer**

**Funkcja bezstanowa zwracająca komponent, która pokazuje, że nie przekazujemy właściwości między Parent a Child2**

**Funkcja bezstanowa, która zwraca komponent opakowany w ThemeContext.Consumer**

Funkcja bezstanowa `Child2` zwraca komponent opakowany w `ThemeContext.Consumer`. `ThemeContext.Consumer` wymaga funkcji jako swojego potomka. Ta funkcja przyjmuje argument zawierający to, co znajduje się w kontekście (w tym przypadku obiekt `val` zawierający dwie właściwości). Możemy teraz używać wartości kontekstu w komponencie.

Korzystając w React z Reduksa, możemy skorzystać z funkcji o nazwie `connect`, która zasadniczo pobiera fragmenty kontekstu i udostępnia je jako właściwości w komponencie. Zrozumienie kontekstu powinno znacznie ułatwić naukę Reduksa!