10.1 10.1 Teoria

Step 1

Formularze internetowe są nadal podobne do formularzy papierowych - stanowią ustrukturyzowany sposób przyjmowania i zapisywania danych wejściowych. Na papierze zapisujesz informacje za pomocą długopisu lub ołówka. W przypadku formularzy przeglądarki do przechwytywania informacji używamy klawiatury, myszy i plików na komputerze. Istnieje wiele elementów formularza, które, takie jak między innymi dane wejściowe, zaznaczanie czy obszar tekstowy. Większość aplikacji internetowych w mniejszym lub większym stopniu obejmują formularze. Formularze czasami objęte są złą reputację, ponieważ są trudne w obróbce. Być może właśnie z tego powodu wiele frameworków zaimplementowało „magiczne" podejście do formularzy, które ma na celu zmniejszenie obciążenia programisty. React nie stosuje magicznego podejścia, ale może ułatwić pracę z formularzami.

Nie ma standardowego sposobu tworzenia formularzy w ramach frontendu. W niektórych strukturach i bibliotekach można skonfigurować model formularza, który jest aktualizowany, gdy użytkownik zmienia wartości formularza, i ma wbudowane specjalne metody wykrywania, gdy formularz jest w różnych stanach. Inni wdrażają różne paradygmaty i techniki, jeśli chodzi o formularze. Łączy ich to, że mają nieco inne formy. Czasami „łatwiejsze w użyciu" podejście może przesłaniać podstawowe mechanizmy i logikę. To nie zawsze jest złe - czasami nie potrzebujemy wglądu w wewnętrzne działanie struktury. Ale musimy mieć wystarczające zrozumienie, aby wesprzeć model mentalny, który pozwoli nam na stworzenie łatwego do utrzymania kodu i naprawienia błędy, gdy się pojawią. Model obsługujący formularze w React jest bardziej tym, co już wiemy o tych formularzach. Nie ma specjalnego zestawu interfejsów API do użycia - formularze to po prostu więcej tego, co widzieliśmy do tej pory w React: komponenty! Do tworzenia formularzy używamy komponentów, stanów i właściwości. React umożliwia interakcję ze zdarzeniami, tak jak w zwykłym JavaScript przeglądarki. Pozwala słuchać regularnych zdarzeń, takich jak kliknięcia, przewijanie i inne, i reagować na nie. Będziemy korzystać z tych zdarzeń podczas pracy z formularzami. Należy mieć pod uwagę fakt, że istnieje wiele niespójności między różnymi przeglądarkami, zwłaszcza jeśli chodzi o zdarzenia. React wykonuje również dużo pracy, aby abstrahować od tych różnic w implementacjach przeglądarek. To korzyść, której nie poświęca się zbyt wiele uwagi, ale może być niesamowitą pomocą. Brak konieczności martwienia się tak bardzo o różnice między przeglądarkami pozwala skupić się bardziej na innych obszary aplikacji i generalnie powoduje, że programiści są szczęśliwsi. W wyniku interakcji użytkownika w przeglądarce może wystąpić wiele różnych zdarzeń - w tym ruchy myszą, pisanie na klawiaturze, kliknięcia i nie tylko. Martwimy się kilkoma tego typu wydarzeniami, szczególnie w przypadku naszych aplikacji. Dla naszych celów, ty chcesz słuchać przy użyciu dwóch głównych programów obsługi zdarzeń - `onChange` i `onClick.` Zdarzenie `onChange` jest uruchamiane, gdy zmienia się element wejściowy. Możemy uzyskać dostęp do nowego wartość elementu formularza za pomocą `event.target.value`. Zdarzenie onClick jest uruchamiane po kliknięciu elementu. Będziemy tego używać, aby wiedzieć, kiedy użytkownik chce wysłać dane z formularza na serwer.

Bycie programistą oznacza zbieranie dużej ilości informacji od użytkowników za pomocą formularzy. Oznacza to, że będziemy budować wiele komponentów formularzy za pomocą React. Wszystkie elementy formularza HTML, które są dostępne dla DOM, są również dostępne jako elementy React, co oznacza, że możemy je wyrenderować za pomocą JSX:

```
<form>
  <input type="text" placeholder="color title..." required />
  <input type="color" required />
  <button>ADD</button>
</form>
```

Formularza ma trzy elementy podrzędne: dwa elementy wejściowe (`input`) i przycisk (`button`). Pierwszym elementem wejściowym jest tekst, który zostanie użyty do zebrania wartości tytułu dla nowych kolorów. Drugi element wejściowy to wejście koloru HTML, które pozwoli użytkownikom wybrać kolor z koła kolorów. Będziemy używać podstawowego sprawdzania poprawności formularza HTML, więc oznaczyliśmy oba dane wejściowe jako wymagane. Przycisk `ADD` posłuży do dodania nowego koloru.

Kiedy przychodzi czas na zbudowanie komponentu formularza w React, mamy do dyspozycji kilka wzorców. Jeden z tych wzorców obejmuje bezpośredni dostęp do węzła DOM za pomocą elementu React o nazwie `ref` (odnośnik). W React `ref` to obiekt przechowujący wartości przez cały okres istnienia komponentu. Istnieje kilka przypadków użycia, które wymagają użycia `ref`. Przyjrzymy się teraz, w jaki sposób możemy uzyskać bezpośredni dostęp do węzła DOM za pomocą `ref`. React dostarcza nam hook `useRef`, którego możemy użyć do stworzenia `ref`. Użyjemy tego hooka podczas tworzenia komponentu `AddColorForm`:

```
//AddColorForm.js

import React, { useRef } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
    const txtTitle = useRef();
    const hexColor = useRef();

    const submit = e => { ... }

    return (...)
}
```

Po pierwsze, podczas tworzenia tego komponentu utworzymy również dwa odniesienia za pomocą hooka `useRef`. Odnośnik (`ref`) `txtTitle` zostanie użyty do odniesienia się do danych wejściowych, które dodaliśmy do formularza w celu zebrania tytułu koloru. Odnośnik `hexColor` zostanie użyty do uzyskania dostępu do szesnastkowych wartości kolorów z danych wejściowych koloru HTML. Możemy ustawić wartości dla tych referencji bezpośrednio w JSX za pomocą właściwości ref:

```
//AddColorForm.js

import React, { useRef } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
    const txtTitle = useRef();
    const hexColor = useRef();

    const submit = e => { ... }

    return (
        <form onSubmit={submit}>
            <input ref={txtTitle} type="text" placeholder="color title..." required />
            <input ref={hexColor} type="color" required />
            <button>ADD</button>
        </form>
    );
}
```

Tutaj ustawiamy wartość `txtTitle` i `hexColor`, dodając atrybut ref do tych elementów wejściowych w JSX. Tworzy to bieżące pole w naszym obiekcie `ref`, które bezpośrednio odwołuje się do elementu DOM. To daje nam dostęp do elementu DOM, co oznacza, że możemy uchwycić jego wartość. Gdy użytkownik prześle ten formularz, klikając przycisk ADD, wywołamy funkcję przesyłania:

```
//AddColorForm.js

import React, { useRef } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
    const txtTitle = useRef();
    const hexColor = useRef();

    const submit = e => {
        e.preventDefault();
        const title = txtTitle.current.value;
        const color = hexColor.current.value;
        console.log(title);
        console.log(color);
        onNewColor(title, color);
        txtTitle.current.value = "";
        hexColor.current.value = "";
    };

    return (
        <form onSubmit={submit}>
            <input ref={txtTitle} type="text" placeholder="color title..." required />
            <input ref={hexColor} type="color" required />
            <button>ADD</button>
        </form>
    );
}
```

Kiedy przesyłamy formularze HTML, domyślnie wysyłają one żądanie POST do bieżącego adresu URL z wartościami elementów formularza przechowywanych w treści. Tutaj nie chcemy tego robić. Dlatego pierwszą linią kodu w funkcji wysyłania jest `e.preventDefault ()`, która uniemożliwia przeglądarce próbę wysłania formularza z żądaniem POST. Następnie przechwytujemy bieżące wartości dla każdego z naszych elementów formularza, używając ich odnośników. Wartości te są następnie przekazywane do elementu nadrzędnego tego składnika za pośrednictwem właściwości funkcji `onNewColor`. Zarówno tytuł, jak i wartość szesnastkowa nowego koloru są przekazywane jako argumenty funkcji. Na koniec resetujemy atrybut `value` dla obu danych wejściowych, aby wyczyścić dane i przygotować formularz do zebrania innego koloru. Modyfikujemy bezpośrednio atrybut wartości węzłów DOM, ustawiając je jako równe „" pustym ciągom. To jest kod obowiązkowy. `AddColorForm` jest teraz tym, co nazywamy **niekontrolowanym komponentem**, ponieważ używa DOM do zapisywania wartości formularza. Czasami użycie niekontrolowanego komponentu może pomóc w rozwiązaniu problemów. Na przykład możemy chcieć udostępnić formularz i jego wartości poza React. Jednak kontrolowany komponent jest lepszym podejściem.

Step 2

W **kontrolowanym komponencie** wartościami formularza zarządza React, a nie DOM. Nie wymagają od nas używania odnośników. Nie wymagają od nas pisania kodu imperatywnego. Podczas pracy z kontrolowanym komponentem znacznie łatwiej jest dodawać funkcje, takie jak solidne sprawdzanie poprawności formularzy. Zmodyfikujmy `AddColorForm`, dając mu kontrolę nad stanem formularza:

```
//AddColorForm.js

import React, { useState } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
    const [title, setTitle] = useState("");
    const [color, setColor] = useState("#000000");

    const submit = e => { ... };

    return ( ... );
}
```

Po pierwsze, zamiast używać odnośników, zapiszemy wartości tytułu i koloru za pomocą stanu React. Stworzymy zmienne dla tytułu i koloru. Dodatkowo zdefiniujemy funkcje, których można użyć do zmiany stanu: `setTitle` i `setColor`. Teraz, gdy komponent kontroluje wartości tytułu i koloru, możemy wyświetlić je wewnątrz elementów wejściowych formularza, ustawiając atrybutu `value`. Gdy ustawimy atrybut `value` elementu wejściowego, nie będziemy już mogli zmieniać go za pomocą formularza. Jedynym sposobem zmiany wartości w tym momencie byłaby zmiana zmiennej stanu za każdym razem, gdy użytkownik wpisze nowy znak w elemencie wejściowym. Dokładnie to zrobimy:

```
//AddColorForm.js

import React, { useState } from "react";

export default function AddColorForm({ onNewColor = f => f }) {
    const [title, setTitle] = useState("");
    const [color, setColor] = useState("#000000");

    const submit = e => { ... };

    return (
        <form onSubmit={submit}>
        <input
            value={title}
            onChange={event => setTitle(event.target.value)}
            type="text"
            placeholder="color title..."
            required
        />
        <input
            value={color}
            onChange={event => setColor(event.target.value)}
            type="color"
            required
        />
        <button>ADD</button>
    </form>
    );
}
```

Ten kontrolowany komponent ustawia teraz wartość obu elementów wejściowych przy użyciu tytułu i koloru ze stanu. Za każdym razem, gdy te elementy wywołują zdarzenie `onChange`, możemy uzyskać dostęp do nowej wartości za pomocą argumentu `event`. `event.target` jest odniesieniem do elementu DOM, więc możemy uzyskać bieżącą wartość tego elementu za pomocą `event.target.value`. Gdy zmieni się tytuł, wywołamy `setTitle`, aby zmienić wartość tytułu w stanie. Zmiana tej wartości spowoduje ponowne wyrenderowanie tego komponentu i możemy teraz wyświetlić nową wartość tytułu w elemencie wejściowym. Zmiana koloru działa dokładnie w ten sam sposób. Kiedy nadejdzie czas na przesłanie formularza, możemy po prostu przekazać wartości stanu dla tytułu i koloru do funkcji `onNewColor` jako argumenty podczas jej wywoływania. Funkcje `setTitle` i `setColor` mogą służyć do resetowania wartości po przekazaniu nowego koloru do komponentu nadrzędnego:

```
//AddColorForm.js

import React, { useState } from "react";

export default function AddColorForm({ onNewColor = f => f}) {
    const [title, setTitle] = useState("");
    const [color, setColor] = useState("#000000");

    const submit = e => {
        e.preventDefault();
        console.log(title);
        console.log(color);
        onNewColor(title, color);
        setTitle("");
        setColor("");
    };

    return (
        <form onSubmit={submit}>
        <input
            value={title}
            onChange={event => setTitle(event.target.value)}
            type="text"
            placeholder="color title..."
            required
        />
        <input
            value={color}
            onChange={event => setColor(event.target.value)}
            type="color"
            required
        />
        <button>ADD</button>
    </form>
    );
}
```

Nazywa się to kontrolowanym komponentem, ponieważ React kontroluje stan formularza. Warto zaznaczyć, że kontrolowane komponenty formularzy są często rerenderowane. Każdy nowy znak wpisany w polu tytułu powoduje ponowne wyrenderowanie AddColorForm. Użycie próbniku kolorów powoduje, że ten komponent odwraca się znacznie bardziej niż pole tytułu, ponieważ wartość koloru zmienia się wielokrotnie, gdy użytkownik przeciąga kursor myszy po kole kolorów. To jest w porządku - React jest zaprojektowany do obsługi tego typu obciążenia. Miejmy nadzieję, że świadomość, że kontrolowane komponenty są często renderowane, uniemożliwi dodanie długiego i kosztownego procesu do tego komponentu.

Step 3

Jeśli mamy duży formularz z wieloma elementami wejściowymi, możesz ulec pokusie skopiowania i wklejenia tych dwóch wierszy kodu:

```
value={title}
onChange={event => setTitle(event.target.value)}
```

Może się wydawać, że pracujemy szybciej, po prostu kopiując i wklejając te właściwości do każdego elementu formularza, jednocześnie modyfikując po drodze nazwy zmiennych. Jednak za każdym razem, gdy kopiujesz i wklejasz kod, powinieneś usłyszeć cichy dźwięk alarmu w głowie. Kopiowanie i wklejanie kodu sugeruje, że jest coś na tyle zbędnego, aby można było usunąć je z funkcji. Istnieje możliwość tworzenia własnych hooków. Na przykład moglibyśmy stworzyć własny hook `useInput`, w którym możemy oddzielić nadmiarowość związaną z tworzeniem kontrolowanych danych wejściowych formularza:

```
//hooks.js

import { useState } from "react";

export const useInput = initialValue => {
    const [value, setValue] = useState(initialValue);
    return [
        { value, onChange: e => setValue(e.target.value) },
        () => setValue(initialValue)
    ];
};
```

To jest niestandardowy hook. Nie wymaga dużo kodu. Wewnątrz tego hook nadal używamy hook `useState` do tworzenia wartości stanu. Następnie zwracamy tablicę. Pierwsza wartość tablicy to obiekt, który zawiera te same właściwości, które kusiło nas, aby skopiować i wkleić: wartość ze stanu wraz z właściwością funkcji `onChange`, która zmienia tę wartość w stanie. Druga wartość w tablicy to funkcja, której można użyć ponownie w celu przywrócenia wartości początkowej. Możemy użyć naszego hooka wewnątrz `AddColorForm`:

```
//AddColorForm.js

import React from "react";
import { useInput } from "./hooks";

export default function AddColorForm({ onNewColor = f => f }) {
    const [titleProps, resetTitle] = useInput("");
    const [colorProps, resetColor] = useInput("#000000");

    const submit = event => { ... }

    return ( ... )
}
```

Hook `useState` jest zamknięty w utworzonym hook `useInput`. Możemy uzyskać właściwości zarówno tytułu, jak i koloru, używając destrukturyzacji z pierwszej wartości zwróconej tablicy. Druga wartość tej tablicy zawiera funkcję, której możemy użyć do przywrócenia właściwości `value` z powrotem do jej wartości początkowej, czyli pustego ciągu. `titleProps` i `colorProps` są gotowe do rozłożenia na odpowiadające im elementy wejściowe:

```
//AddColorForm.js

import React from "react";
import { useInput } from "./hooks";

export default function AddColorForm({ onNewColor = f => f }) {
    const [titleProps, resetTitle] = useInput("");
    const [colorProps, resetColor] = useInput("#000000");

    const submit = event => { ... }

    return (
        <form onSubmit={submit}>
            <input
                {...titleProps}
                type="text"
                placeholder="color title..."
                required
            />
            <input {...colorProps} type="color" required />
            <button>ADD</button>
        </form>
    );
}
```

Rozprzestrzenianie tych właściwości z naszego niestandardowego hooka jest znacznie przyjemniejsze niż ich wklejanie. Teraz zarówno tytuł, jak i dane wejściowe koloru otrzymują właściwości dotyczące ich wartości i zdarzeń `onChange`. Użyliśmy naszego hooka do tworzenia kontrolowanych danych wejściowych formularza bez martwienia się o szczegóły implementacji. Jedyną inną zmianą, jaką musimy wprowadzić, jest przesłanie tego formularza:

```
//AddColorForm.js

import React from "react";
import { useInput } from "./hooks";

export default function AddColorForm({ onNewColor = f => f }) {
    const [titleProps, resetTitle] = useInput("");
    const [colorProps, resetColor] = useInput("#000000");

    const submit = event => {
        event.preventDefault();
        console.log(titleProps.value, colorProps.value);
        onNewColor(titleProps.value, colorProps.value);
        resetTitle();
        resetColor();
    };

    return (
        <form onSubmit={submit}>
            <input
                {...titleProps}
                type="text"
                placeholder="color title..."
                required
            />
            <input {...colorProps} type="color" required />
            <button>ADD</button>
        </form>
    );
}
```

W ramach funkcji `submit` musimy upewnić się, że pobraliśmy wartość zarówno tytułu, jak i koloru z ich właściwości. Wreszcie, możemy użyć niestandardowych funkcji resetowania, które zostały zwrócone z podpięcia `useInput`. Hooki są przeznaczone do użytku wewnątrz komponentów React. Możemy komponować hooki w innych hookach, ponieważ ostatecznie dostosowany hook zostanie użyty wewnątrz komponentu. Zmiana stanu w tym zaczepie nadal powoduje, że `AddColorForm` ponownie wyświetla nowe wartości dla `titleProps` lub `colorProps`.

Step 4

Zarówno kontrolowany komponent formularza, jak i niekontrolowany komponent przekazują wartości tytułu i koloru do komponentu nadrzędnego za pośrednictwem funkcji `onNewColor`. Rodzica nie obchodzi, czy użyliśmy komponentu kontrolowanego czy niekontrolowanego; chce tylko wartości dla nowego koloru. Dodajmy `AddColorForm`, do komponentu aplikacji. Po wywołaniu właściwości `onNewColor` zapiszemy nowy kolor w stanie:

```
//App.js
import React, { useState } from "react";
import colorData from "./data/color-data.json";
import ColorList from "./components/ColorList.js";
import AddColorForm from "./components/AddColorForm";
import { v4 } from "uuid";

export default function App() {
    const [colors, setColors] = useState(colorData);
    return (
        <>
            <AddColorForm
                onNewColor={(title, color) => {
                    const newColors = [
                        ...colors,
                        {
                            id: v4(),
                            rating: 0,
                            title,
                            color
                        }
                    ];
                    setColors(newColors);
                }}
            />
            <ColorList
                colors={colors}
                onRateColor={(id, rating) => {
                    const newColors = colors.map(color =>
                        color.id === id ? { ...color, rating } : color
                    );
                    setColors(newColors);
                }}
                onRemoveColor={id => {
                    const newColors = colors.filter(color => color.id !== id);
                    setColors(newColors);
                }}
            />
       </>
);
}
```

Po dodaniu nowego koloru wywoływana jest właściwość `onNewColor`. Tytuł i wartość szesnastkowa nowego koloru są przekazywane do tej funkcji jako argumenty. Używamy tych argumentów, aby stworzyć nową tablicę kolorów. Najpierw przenosimy aktualne kolory ze stanu do nowej tablicy. Następnie dodajemy całkowicie nowy obiekt koloru, używając tytułu i wartości koloru. Dodatkowo ustawiliśmy ocenę nowego koloru na 0, ponieważ nie został jeszcze oceniony. Używamy również funkcji `v4` znajdującej się w pakiecie `uuid`, aby wygenerować nowy unikalny identyfikator koloru. Gdy już mamy tablicę kolorów, która zawiera nasz nowy kolor, zapisujemy go do stanu, wywołując `setColors`. Powoduje to, że składnik aplikacji wyrenderuje się z nową tablicą kolorów. Ta nowa tablica zostanie użyta do zaktualizowania interfejsu użytkownika. Nowy kolor zobaczymy u dołu listy. Dzięki tej zmianie zakończyliśmy pierwszą iterację organizatora kolorów. Użytkownicy mogą teraz dodawać nowe kolory do listy, usuwać kolory z listy i oceniać istniejący kolor na tej liście.

Link do projektu: https://drive.google.com/file/d/1ljUApeBh4PCJI9-iZ_5a3W4zv8rlBt81/view?usp=sharing (https://drive.google.com/file/d/1ljUApeBh4PCJI9-iZ_5a3W4zv8rlBt81/view?usp=sharing)

Step 5

Przechowywanie stanu w jednym miejscu w korzeniu drzewa DOM było ważnym wzorcem, który pomaga odnieść większy sukces we wczesnych wersjach React. Nauka przekazywania stanu zarówno w dół, jak i w górę drzewa komponentów za pomocą właściwości jest niezbędnym prawem do przejścia dla każdego programisty React - to coś, co każdy programista powinien wiedzieć, jak to zrobić. Jednak wraz z ewolucją Reacta i powiększaniem się drzew składowych, przestrzeganie tej zasady powoli stawało się bardziej nierealne. Wielu programistom trudno jest utrzymać stan w jednym miejscu w katalogu głównym drzewa komponentów dla złożonej

aplikacji. Przekazywanie stanu w dół i w górę drzewa przez dziesiątki komponentów jest żmudne i obarczone błędami. Elementy interfejsu użytkownika, nad którymi będziemy przeważnie pracować są złożone. Korzeń drzewa jest często bardzo daleko od liści. Wówczas dane, z których aplikacja zależy, są oddalone o wiele warstw od składników, które z nich korzystają. Każdy element musi otrzymać props, które przekazują tylko swoim dzieciom. Spowoduje to rozdęcie naszego kodu i utrudni skalowanie naszego interfejsu użytkownika. Przekazywanie danych o stanie przez każdy komponent jako props, aż do komponentu, który musi ich użyć, przypomina podróż pociągiem z San Francisco do DC. W pociągu przejedziesz przez wszystkie stany, ale nie wysiądziesz, dopóki nie dotrzesz do celu (prostszym i szybszym rozwiązaniem byłoby po prostu lot samolotem :-) - w naszej teorii nie musimy przechodzić przez każdy ze stanów aplikacji).

Do realizacji powyższego celu będziemy używać tzw. **kontekstu** w React. W React kontekst jest jak ustawienie odrzutowca dla danych. Możesz umieścić dane w kontekście Reacta, tworząc tzw. **dostawcę kontekstu**. Dostawca kontekstu to komponent Reacta, który można zawinąć wokół całego drzewa komponentów lub określonych sekcji drzewa komponentów. Można skojarzyć sobie, że dostawca kontekstu jest jak lotnisko odlotu, na którym dane są umieszczane na pokładzie samolotu. To także centrum lotnicze. Wszystkie loty odlatują z tego lotniska do różnych miejsc docelowych. Każde miejsce docelowe jest tzw. **konsumentem kontekstu**. Konsument kontekstu to komponent React, który pobiera dane z kontekstu. To jest lotnisko docelowe, na którym dane są "lądowane", obniżane i trafiają do pracy. Korzystanie z kontekstu nadal pozwala nam przechowywać dane stanu w jednym miejscu, ale nie wymaga od nas przekazywania tych danych przez kilka komponentów, które ich nie potrzebują.

Aby użyć kontekstu w React, musimy najpierw umieścić dane w dostawcy kontekstu i dodać tego dostawcę do naszego drzewa komponentów. React zawiera funkcję o nazwie `createContext`, której możemy użyć do stworzenia nowego obiektu kontekstu. Ten obiekt zawiera dwa komponenty: dostawcę kontekstu i konsumenta.

Umieśćmy domyślne kolory z pliku `color-data.json` w kontekście. Dodamy kontekst do pliku `index.js`, punktu wejścia naszej aplikacji:

```
//index.js
import React, { createContext } from 'react';
import colors from "./data/color-data";
import { render } from "react-dom";
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';

export const ColorContext = createContext();

render(
    <ColorContext.Provider value={{ colors }}>
        <App />
    </ColorContext.Provider>,
    document.getElementById("root")
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Korzystając z `createContext`, tworzymy nową instancję kontekstu `React`, którą nazwaliśmy `ColorContext`. Kontekst koloru zawiera dwa składniki: `ColorContext.Provider` i `ColorContext.Consumer`. Teraz należy użyć dostawcy, aby umieścić kolory w stanie. Dodamy dane do kontekstu, ustawiając właściwość `value` dostawcy. W tym scenariuszu dodamy obiekt zawierający kolory do kontekstu. Ponieważ zawarliśmy cały komponent aplikacji z dostawcą, wachlarz kolorów zostanie udostępniony każdemu konsumentowi kontekstu, który znajdzie w całym naszym drzewie komponentów. Należy zauważyć, że wyeksportowaliśmy również `ColorContext` z tej lokalizacji. Jest to konieczne, ponieważ będziemy musieli uzyskać dostęp do `ColorContext.Consumer,` gdy chcemy uzyskać kolory z kontekstu. Dostawca kontekstu nie zawsze musi opakowywać całą aplikację. Otaczanie określonych komponentów sekcji dostawcą kontekstu jest nie tylko w porządku, ale może również zwiększyć wydajność aplikacji. Dostawca zapewni tylko wartości kontekstu swoim elementom podrzędnym. Można używać wielu dostawców kontekstu. W rzeczywistości można już używać dostawców kontekstów w swojej aplikacji `React`, nawet o tym nie wiedząc. Wiele pakietów `npm` zaprojektowanych do pracy z Reactem używa kontekstu za kulisami.

Teraz, gdy podajemy wartość kolorów w kontekście, składnik aplikacji nie musi już utrzymywać stanu i przekazywać go swoim elementom podrzędnym jako `props`. Z komponentu aplikacji zrobiliśmy komponent „estakady". Dostawca jest rodzicem składnika aplikacji i będzie on zapewniał kolory w kontekście. `ColorList` jest elementem podrzędnym składnika aplikacji i może samodzielnie pobierać kolory. Aplikacja nie musi więc w ogóle dotykać kolorów, co jest świetne, ponieważ sam składnik aplikacji nie ma nic wspólnego z kolorami. Ta odpowiedzialność została przekazana niżej w dół drzewa. Możemy usunąć wiele linii kodu z komponentu aplikacji. Wystarczy wyrenderować `AddColorForm` i `ColorList` i nie musimy już martwić się o dane:

```
//App.js
import React from "react";
import ColorList from "./components/ColorList.js";
import AddColorForm from "./components/AddColorForm";

export default function App() {
    return (
        <>
            <AddColorForm />
            <ColorList />
        </>
    );
}
```

Dodanie hooków sprawia, że praca z kontekstem jest przyjemnością. Hook `useContext` służy do uzyskiwania wartości z kontekstu i uzyskuje te wartości, których potrzebujemy, z kontekstu `Consumer`. Składnik `ColorList` nie musi już pobierać tablicy kolorów na podstawie swoich właściwości. Może uzyskać do nich bezpośredni dostęp poprzez hook `useContext`:

```
//ColorList.js

import React, { useContext } from "react";
import { ColorContext } from "../index.js";
import Color from "./Color";

export default function ColorList() {
    const { colors } = useContext(ColorContext);
    if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;
    return (
        <div className="color-list">
            {
                colors.map(color => <Color key={color.id} {...color} />)
            }
        </div>
    );
}
```

W tym miejscu zmodyfikowaliśmy komponent `ColorList`, usuwając właściwość `colors = []`, ponieważ kolory są pobierane z kontekstu. Hook `useContext` wymaga, aby wystąpienie kontekstu uzyskało z niego wartości. `ColorContext` jest importowane z pliku `index.js`, w którym tworzymy kontekst i dodajemy dostawcę do naszego drzewa komponentów. `ColorList` może teraz konstruować interfejs użytkownika na podstawie danych dostarczonych w kontekście. Dostęp do `Consumer` uzyskuje się za pomocą hooka o nazwie `useContext`, co oznacza, że nie musimy już pracować bezpośrednio z komponentem konsumenta. ~~Przed hookami musielibyśmy uzyskać kolory z kontekstu za pomocą wzorca zwanego `render props` w `Consumer`. `render props` są przekazywane jako argumenty do funkcji potomnej. Poniższy przykład przedstawia sposób, w jaki użytkownik użyłby konsumenta do uzyskania kolorów z kontekstu:~~

```
//ColorList.js

import React, { useContext } from "react";
import { ColorContext } from "../index.js";
import Color from "./Color";

export default function ColorList() {
    return (
        <ColorContext.Consumer>
            {context => {
                if (!context.colors.length)
                    return <div>No Colors Listed. (Add a Color)</div>;
                return (
                    <div className="color-list">
                        {
                            context.colors.map(color =>
                                <Color key={color.id} {...color} />)
                        }
                    </div>
                );
            }}
        </ColorContext.Consumer>
    );
}
```

Step 6

Dostawca kontekstu może umieścić obiekt w kontekście, ale nie może samodzielnie mutować wartości w kontekście. Potrzebuje pomocy ze strony komponentu nadrzędnego. Sztuczka polega na utworzeniu składnika stanowego, który renderuje dostawcę kontekstu. Gdy stan składnika stanowego ulegnie zmianie, ponownie zwróci on dostawcę kontekstu z nowymi danymi kontekstowymi. Wszystkie elementy podrzędne dostawców kontekstu również zostaną ponownie wyrenderowane z nowymi danymi kontekstowymi. Składnikiem stanowym, który renderuje dostawcę kontekstu, jest nasz niestandardowy dostawca. To znaczy: to jest komponent, który zostanie użyty, gdy nadejdzie czas, aby połączyć naszą aplikację z dostawcą. W zupełnie nowym pliku stwórzmy komponent o nazwie `ColorProvider`:

```
//ColorProvider.js

import React, { createContext, useState } from "react";
import colorData from "../data/color-data.json";

const ColorContext = createContext();

export default function ColorProvider ({ children }) {
    const [colors, setColors] = useState(colorData);
    return (
        <ColorContext.Provider value={{ colors, setColors }}>
            {children}
        </ColorContext.Provider>
    );
};
```

`ColorProvider` to składnik, który renderuje `ColorContext.Provider`. W ramach tego komponentu utworzyliśmy zmienną stanu dla kolorów za pomocą hooka `useState`. Początkowe dane dotyczące kolorów są nadal wypełniane z `color-data.json`. Następnie `ColorProvider` dodaje kolory ze stanu do kontekstu przy użyciu właściwości `value` `ColorContext.Provider`. Wszystkie elementy podrzędne renderowane w `ColorProvider` zostaną opakowane przez `ColorContext.Provider` i będą miały dostęp do tablicy kolorów z kontekstu. Funkcja `setColors` jest również dodawana do kontekstu. Daje to konsumentom kontekstu możliwość zmiany wartości kolorów. Za każdym razem, gdy wywoływana jest metoda `setColors`, tablica kolorów ulegnie zmianie. Spowoduje to ponowne wyrenderowanie `ColorProvider`, a nasz interfejs użytkownika zaktualizuje się, aby wyświetlić nową tablicę kolorów. Dodanie `setColors` do kontekstu może nie być najlepszym pomysłem. Zachęca innych programistów do popełnienia błędów później podczas korzystania z niego. Istnieją tylko trzy opcje zmiany wartości tablicy kolorów: użytkownicy mogą dodawać kolory, usuwać kolory lub oceniać kolory. Lepszym pomysłem jest dodanie funkcji do każdej z tych operacji w kontekście. W ten sposób nie ujawnimy funkcji `setColors` konsumentom; ujawnimy funkcje tylko w przypadku zmian, które mogą je wprowadzać:

```
//ColorProvider.js

import React, { createContext, useState } from "react";
import colorData from "../data/color-data.json";
import { v4 } from "uuid";
const ColorContext = createContext();

export default function ColorProvider ({ children }) {
    const [colors, setColors] = useState(colorData);

    const addColor = (title, color) =>
        setColors([
            ...colors,
            {
                id: v4(),
                rating: 0,
                title,
                color
            }
        ]);

    const rateColor = (id, rating) =>
        setColors(
            colors.map(color => (color.id === id ? { ...color, rating } : color))
        );

    const removeColor = id => setColors(colors.filter(color => color.id !== id));

    return (
        <ColorContext.Provider value={{ colors, addColor, removeColor, rateColor }}>
            {children}
        </ColorContext.Provider>
    );
};
```

Wygląda lepiej. Dodana została funkcja do kontekstu dla wszystkich operacji, które można wykonać na tablicy `colors`. Teraz każdy komponent w naszym drzewie może korzystać z tych operacji i wprowadzać zmiany w kolorach za pomocą prostych funkcji, które możemy udokumentować.

Pozostała jeszcze jedna zmiana, którą możemy wprowadzić. Wprowadzenie hooków sprawiło, że nie musimy w ogóle ujawniać kontekstu komponentom konsumenckim. Kontekst może być mylący dla członków zespołu, którzy nie czytają tej książki. Możemy im wszystko znacznie ułatwić, opakowując kontekst w niestandardowy hook. Zamiast ujawniać instancję `ColorContext`, możemy utworzyć hook o nazwie `useColors`, który zwraca kolory z kontekstu:

```
import React, { createContext, useState, useContext } from "react";
import colorData from "../data/color-data.json";
import { v4 } from "uuid";
const ColorContext = createContext();
export const useColors = () => useContext(ColorContext);

export default function ColorProvider ({ children }) {
    const [colors, setColors] = useState(colorData);

    const addColor = (title, color) =>
        setColors([
            ...colors,
            {
                id: v4(),
                rating: 0,
                title,
                color
            }
        ]);

    const rateColor = (id, rating) =>
        setColors(
            colors.map(color => (color.id === id ? { ...color, rating } : color))
        );

    const removeColor = id => setColors(colors.filter(color => color.id !== id));

    return (
        <ColorContext.Provider value={{ colors, addColor, removeColor, rateColor }}>
            {children}
        </ColorContext.Provider>
    );
};
```

Ta prosta zmiana ma ogromny wpływ na architekturę. Wszystkie funkcje niezbędne do renderowania kolorów stanowych i pracy z nimi zawarliśmy w jednym module `JavaScript`. Kontekst jest zawarty w tym module, ale uwidoczniony przez hook. To działa, ponieważ zwracamy kontekst za pomocą hooka o nazwie `useContext`, który ma dostęp do obiektu `ColorContext` lokalnie w tym pliku. Teraz dołóżmy utworzony przez nas hook do naszej aplikacji. Najpierw musimy opakować składnik aplikacji `ColorProvider`. Możemy to zrobić w pliku `index.js`:

```
//index.js
import React, { createContext } from 'react';
import colors from "./data/color-data";
import { render } from "react-dom";
import './index.css';
import App from './App';
import ColorProvider from "./components/ColorProvider.js";
import reportWebVitals from './reportWebVitals';

export const ColorContext = createContext();

render(
    <ColorProvider>
        <App />
    </ColorProvider>,
    document.getElementById("root")
);

// If you want to start measuring performance in your app, pass a function
// to log results (for example: reportWebVitals(console.log))
// or send to an analytics endpoint. Learn more: https://bit.ly/CRA-vitals
reportWebVitals();
```

Teraz każdy komponent, który jest elementem podrzędnym aplikacji, może uzyskać kolory z hooka `useColors`. Składnik `ColorList` musi uzyskać dostęp do tablicy kolorów, aby renderować kolory na ekranie:

```
//ColorList.js

import React, { useContext } from "react";
import { ColorContext } from "../index.js";
import Color from "./Color";
import { useColors } from "./ColorProvider";

export default function ColorList() {
    const { colors } = useColors();
    if (!colors.length) return <div>No Colors Listed. (Add a Color)</div>;
    return (
        <div className="color-list">
            {
                colors.map(color => <Color key={color.id} {...color} />)
            }
        </div>
    );
}
```

Usunęliśmy wszelkie odniesienia do kontekstu z tego komponentu. Wszystko, czego potrzebuje, jest teraz dostarczane z naszego hooka. Komponent `Color` może użyć naszego hooka, aby uzyskać funkcje do bezpośredniego oceniania i usuwania kolorów:

```
//Color.js

import React from "react";
import StarRating from "./StarRating";
import { useColors } from "./ColorProvider";

export default function Color({ id, title, color, rating }) {
    const { rateColor, removeColor } = useColors();
    return (
        <section>
            <h1>{title}</h1>
            <button onClick={() => removeColor(id)}>X</button>
            <div style={{ height: 50, backgroundColor: color }} />
            <StarRating
                selectedStars={rating}
                onRate={rating => rateColor(id, rating)}
            />
        </section>
    );
}
```

Teraz komponent `Color` nie musi już przekazywać zdarzeń do rodzica za pomocą właściwości props. Ma dostęp do funkcji `rateColor` i `removeColor` w kontekście. Można je łatwo uzyskać dzięki hook `useColors`. To świetna zabawa, ale jeszcze nie skończyliśmy. `AddColorForm` może również skorzystać z hooka `useColors`:

```
//AddColorForm.js

import React from "react";
import { useInput } from "./hooks";
import { useColors } from "./ColorProvider";

export default function AddColorForm() {
    const [titleProps, resetTitle] = useInput("");
    const [colorProps, resetColor] = useInput("#000000");
    const { addColor } = useColors();

    const submit = e => {
        e.preventDefault();
        addColor(titleProps.value, colorProps.value);
        resetTitle();
        resetColor();
    };

    return (
        <form onSubmit={submit}>
            <input
                {...titleProps}
                type="text"
                placeholder="color title..."
                required
            />
            <input {...colorProps} type="color" required />
            <button>ADD</button>
        </form>
    );
}
```

Składnik `AddColorForm` może dodawać kolory bezpośrednio za pomocą funkcji `addColor`. Po dodaniu, ocenie lub usunięciu kolorów stan wartości kolorów w kontekście ulegnie zmianie. Gdy nastąpi ta zmiana, elementy podrzędne `ColorProvider` są ponownie renderowane z nowymi danymi kontekstowymi. Wszystko to dzieje się za pomocą prostego hooka. Hooki zapewniają programistom stymulację, której potrzebują, aby pozostać zmotywowanym i cieszyć się programowaniem frontendowym. Dzieje się tak przede wszystkim dlatego, że są one niesamowitym narzędziem do rozdzielania problemów. Teraz komponenty Reacta muszą zajmować się tylko renderowaniem innych komponentów Reacta i utrzymywaniem aktualności interfejsu użytkownika. Innymi słowy hooki mogą zajmować się logiką wymaganą do działania aplikacji. Zarówno interfejs użytkownika, jak i hooki można rozwijać osobno, osobno testować, a nawet oddzielnie wdrażać. To bardzo dobra wiadomość dla Reacta.

Link do projektu pod adresem: https://drive.google.com/file/d/1Xybf8-grRhTN_CqzbikK_uwBZhTqEPWa/view?usp=sharing (https://drive.google.com/file/d/1Xybf8-grRhTN_CqzbikK_uwBZhTqEPWa/view?usp=sharing)