Zaznaczam, że poniższe notatki były robione przez osobę nie będącą fanem frontendu i nie pracującą w tym obszarze za wiele. Moje doświadczenie z Reactem to raptem pare godzin. A nóż komuś się to przyda.

Zawartość:

Hooki

Szybkie zerknięcie w dokumentację Reacta nie sprawiło, że wypatrzyłem tam jakiejś konkretnej definicji mechanizmu „hook”. Skupiają się tam raczej na tym co te hooki robią a nie na tym czym są pod spodem.

Mi się hooki kojarzą ze wstrzykiwaniem do funkcji (takiej skompilowanej, której kodu nie znamy) swojego kodu w celu podjęcia próby jakiegoś debugowania. Asembler i te sprawy ☺

Przyjmijmy, że hooki w React to mechanizm, którego zadaniem jest rozszerzenie możliwości zwykłych funkcji (komponentów funkcyjnych) o pewne rzeczy, które posiadają klasy (komponenty klasowe) z pudełka. Przykładem jest tutaj stanowość. Stan instancji klasy może się zmieniać a funkcji nie. Więcej przykładów podam przy opisie konkretnych hooków.

useState()

```
3   const Main = (props) => {
4       const [counter, setCounter] = useState(0);
5
6       console.log("Wykonuje ciało funkcji!!!");
7
8       return(
9           <div style={{margin:"5px", padding: "5px"}}>
10              <span>{counter}</span>
11              <br />
12              <button onClick={() => {setCounter(counter+1)}}>+++</button>
13          </div>
14      )
15  }
```

- hook useState() pozwala definiować zmienne stanowe
- zmiana zmiennej stanowej automatycznie odświeża komponent na ekranie
- w nawiasie kwadratowym [] podajemy nazwę zmiennej oraz nazwę funkcji, która będzie ją modyfikowała (konwencja jak na obrazku [zmienna, set zmienna])
- w parametrze useState() podajemy domyślną wartość
- w linii 12 widać małą lambdę, która inkrementuje za pomocą setCounter zmienną counter co w konsekwencji powoduje odświeżenie komponentu
- czasami nazwa zmiennej stanowej jest nieistotna bo się do niej nigdzie nie odwołujemy – stosuje się wtedy zapis :
    - const [_, setFlag] = useState(true);
    - to tylko przykład
    - pod _ pewnie padała by nazwa flag, ale dzięki powyższemu zapisowi node nie krzyczy w logach o nieużywanej zmiennej

useEffect()

```
3   const Main = (props) => {
4       const [counter, setCounter] = useState(0);
5
6       //some logic
7       console.log("Wykonuje ciało funkcji!!!");
8
9       useEffect(()=>{
10          //some async logic
11          console.log("Nastąpiło odświeżenie komponentu!!!");
12      })
13
14      return(
15          <div style={{margin:"5px", padding: "5px"}}>
16              <span>{counter}</span>
17              <br />
18              <button onClick={() => {setCounter(counter+1)}}>+++</button>
19          </div>
20      )
21  }
```

- hook ten służy do pisania tak zwanych side-effectów
- jeżeli jakaś logika może być wykonana już po wyrenderowaniu komponentu powinnismy umieszczać ją w useEffect
- przykładem może być jakaś komunikacja z API czy ustawianie setInterval()
- każde odświeżenie stanu komponentu -> ponowne wygenerowanie komponentu -> uruchomienie kodu useEffect

useRef()

```
const Main = (props) => {
    const myInput = useRef(null);

    const save = () => {
        let v = myInput.current.value; //read value
        console.log(`Saved ${v}`);

        //set focus
        myInput.current.focus();
    }

    return(
        <div>
            <input ref={myInput} type="text"></input>
            <button onClick={save}>Save</button>
        </div>
    )
}
```

- hook ten służy do tworzenia uchwytów na obiekty HTML
- w przykładzie powyżej zdefiniowaliśmy pusty uchwyt myInput
- następnie tworząc nasz input docelowy przypisaliśmy go do uchwytu

- taki uchwyt w polu **current** przechowuje bieżący stan daneg obiektu – w tym przypadku inputa
- na obrazku widać jak można odnieść się do gettera/settera **value**
- oraz jak ustawić obiektowi focus()
- **jest to całkiem spoko opcja, która pozwala pozbyć się z kodu ciągłych odnośników w stylu document.getElementBy**

useCallback()

```
function factory() {
    return (a, b) => a + b;
}

const sum1 = factory();
const sum2 = factory();

sum1(1, 2); // => 3
sum2(1, 2); // => 3

sum1 === sum2; // => false
sum1 === sum1; // => true
```

- tutaj zaczynają się trochę czary z tego co widzę bo wchodzimy w optymalizację
- każda funkcja to obiekt w JS
- funkcja fabryki zwraca funkcję, która jest obiektem
- zatem sum1 oraz sum2 zawierają obiekt funkcji, który działa tak samo (funkcja działa tak samo bo to ta sama funkcja)
- jednak to **nie są te same obiekty** co można rozumieć: w pamięci są dwie instancje funkcji dodającej dwie liczby zwracanej z fabryki
- co daje nam useCallback() – pozwala on cachować funkcje tak by nie następowało zbędne generowanie ich kopii (wielu obiektów)
- jak sama nazwa wskazuje useCallback zostało stworzone z myślą o funkcjach używanych jako callbacki

```
const ListView = (props) => {
    let items = []

    for(let i = 0;i<10;i++){
        items.push(<div key={i} onClick={() =>props.click(i)}>Element {i}</div>)
    }

    return(
        <div>
            {items}
        </div>
    )
}
```

- zbudowaliśmy komponent generujący n elementową listę
- każdy element można kliknąć co powoduje wywołanie funkcji otrzymanej przez props z rodzica
- nic nadzwyczajnego tutaj nie ma, ale

```
const Main = (props) => {
    const clickItem = useCallback((item) => {
        console.log(`Kliknięto element ${item}`);
    }, [props.data]);

    return(
        <div>
            <ListView click={clickItem}></ListView>
        </div>
    )
}
```

- komponent rodzic
- dzięki opakowaniu funkcji callback w hook useCallback jest ona cachowana do momentu, aż jedna z wymienionych w nawiasie kwadratowym wartości się nie zmieni
- daje nam to mniej więcej tyle, że odświeżenie komponentu Main nie koniecznie musi skutkować odświeżeniem całego komponentu ListView oraz listy tam się znajdującej
- zmiana props.data wymusi dopiero odświeżenie wszystkiego
- tak wiem zagmatwana sprawa
- czemu się tak dzieje
- generując ponownie Main, generujemy ponownie jego metody
- tak więc ListView w parametrze dostaje nową referencje na obiekt metody clickItem co w konsekwencji wymusza jego odświeżenie
- dzięki cachowaniu ListView dostanie tą samą referencje i nie odświeży się

useMemo()

```
const calculate = (iteration) => {
    let value = 0;
    for(let i = 0;i<iteration;i++) value += 1;
    return value;
}
```

- wyobraźmy sobie, że w naszym komponencie używamy wyniku funkcji, która jest bardzo kosztowna

```
const Main = (props) => {
    const [count, setCount] = useState(100000);
    const [data, setData] = useState([]);

    const result = calculate(count);

    return(
        <div>
            <span>{result} for {count} iteration</span>
            <button onClick={() => setCount(count+100000)}>+100000</button>
            <button onClick={() => setCount(count-100000)}>-100000</button>
        </div>
    )
}
```

- komponent wywołujący funkcję
- mamy tu dwie zmienne stanowe
- zmiana którejkolwiek powoduje przeliczanie funkcji calculate na nowo – co nie zawsze ma sens gdyż zmienna data nijak się ma do wyniku obliczeń

```
84  const Main = (props) => {
85      const [count, setCount] = useState(100000);
86      const [data, setData] = useState([]);
87
88      const result = useMemo(() => calculate(count), [count]);
89
90      return(
91          <div>
92              <span>{result} for {count} iteration</span>
93              <button onClick={() => setCount(count+100000)}>+100000</button>
94              <button onClick={() => setCount(count-100000)}>-100000</button>
95          </div>
96      )
97  }
```

- dzięki zastosowaniu hooka useMemo wartość zwracana z calculate jest cachowana do momentu aż jedna z wartości w nawiasie kwadratowym się zmienni
- możemy zmieniać stan **data** i odświeżać komponent – wartość result nie będzie przeliczana

Hooki useCallback i useMemo służą do **memoizowania** wartości.

userContext()

- na powyższym obrazku widzimy pewne drzewo komponentów zagnieżdżonych jeden w drugim
- najbardziej zewnętrzny komponent przechowuje jakąś wartość, która jest następnie przekazywana przez wszystkie poziomy do najbardziej wewnętrznego komponentu
- takie podejście może być uciążliwe zwłaszcza jeżeli sieć zagnieżdżeń się rozszerza lub kurczy w wyniku dodania/usunięcia komponentu

- (1) tworzymy kontekst (w językach niskiego poziomu była by to jakaś struktura/klasa z globalnymi danymi), który jest globalny dla wszystkich komponentów zagnieżdżonych w providerze (2)
- teraz nie musimy przekazywać przez **props/atrybuty** konkretnej wartości do najbardziej zagnieżdżonego widgetu
- wystarczy w tym docelowym komponencie odwołać się do globalnego kontekstu (globlana stała w pliku) i odczytać wartość
- oczywiście wartość może mieć strukturę JSON a nie tylko pojedynczego stringa jak tutaj