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:

- zagnieżdżenia komponentów

- dynamiczne generowanie komponentów

- callbacki

Zagnieżdżanie/Zagnieżdżanie dynamiczne

```
//przykładowy komponent rodzic
const Parent = (props) => {
    return(
        <div>
            <Child numer="1"></Child>
            <Child numer="2"></Child>
        </div>
    );
}
```


```
//przykładowy komponent rodzic
const Parent = (props) => {
    return(
        <div>
            <Child numer="1"></Child>
            <Child numer="2"></Child>
        </div>
    );
}
```


- przygotowałem minimalistyczny komponent – rodzica, który agreguje dwie instancje komponentu – dziecka
- jak widać rodzic w swoim ciele zawiera dwie instancje komponentu Child
- każdemu z nich poprzez atrybut numer podaje liczbę będącą jakimś tam oznaczeniem danego dziecka

Co jeżeli chcielibyśmy by liczba dzieci była dynamiczna?

Przykład 1:

```
5   //przykładowy komponent rodzic
6   const Parent = (props) => {
7       const returnChildren = (count) => {
8           let children = [];
9
10          for(let i = 0;i<count;i++){
11              children.push(<Child numer={i}></Child>);
12          }
13
14          return children;
15      }
16
17      return(
18          <div>
19              {returnChildren(5)}
20          </div>
21      );
22  }
```

- w linii 7 definiujemy funkcję, która buduje tablicę komponentów (tablicę dzieci) i ją zwraca
- w linii 19 następuje wywołanie tejże funkcji i w to miejsce wrzucane wygenerowane dzieci

- powyższe rozwiązanie ma jednak pewien problem
- react nie umie zidentyfikować każdego komponentu wygenerowanego dynamicznie
- **dlatego należy każdemu dziecku nadać atrybut key, który będzie miał unikalną wartość na danej stronie**

```
5   //przykładowy komponent rodzic
6   const Parent = (props) => {
7       const returnChildren = (count) => {
8           let children = [];
9
10          for(let i = 0;i<count;i++){
11              children.push(<Child key={i} numer={i}></Child>);
12          }
13
14          return children;
15      }
16
17      return(
18          <div>
19              {returnChildren(5)}
20          </div>
21      );
22  }
```

- w tym przypadku zmienna i może być kluczem, bo dla każdego dziecka przyjmuje inną wartość
- jeżeli pobieramy coś z bazy danych to takim kluczem może być np. klucz główny
- z tego co wiem nie da się w dziecku odwołać do key dlatego zostawiłem atrybut numer

Przykład 2:

```
24  const Parent = (props) => {
25      let children = [];
26
27      for(let i = 0;i<5;i++){
28          children.push(<Child key={i} numer={i}></Child>);
29      }
30
31      return(
32          <div>
33              {children}
34          </div>
35      );
36  }
```

- należy pamiętać, że komponenty funkcyjne to funkcje, które wykonują się linia po linii

Przykład 3: (załóżmy, że chcemy wyświetlić coś bardziej zaawansowanego)

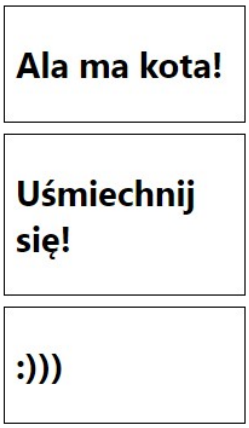
```jsx
//przykładowy komponent dziecko
const Child = (props) => {
    return(
        <div style={{border: "1px solid black", margin: "10px", padding: "10px", width: "200px"}}>
            <h1>{props.text}</h1>
        </div>
    );
}
```

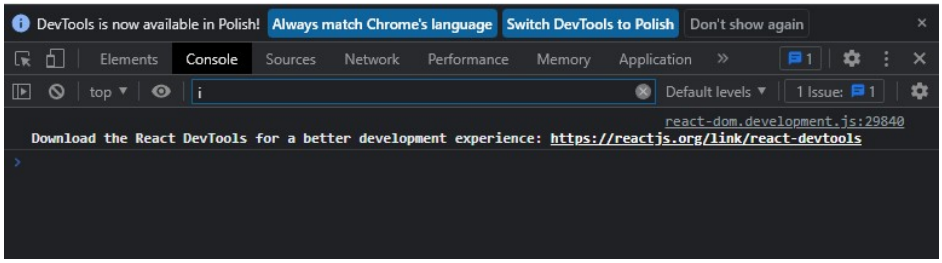
```jsx
const Parent = (props) => {
    let data = [
        {id: 78, text: "Ala ma kota!"},
        {id: 2345, text: "Uśmiechnij się!"},
        {id: 111, text: ":)))"},
    ];

    return(
        <div>
            {data.map((item)=><Child key={item.id} text={item.text}></Child>)}
        </div>
    );
}
```

- zmienna data przechowuje jakieś dane np. pobrane z bazy danych lub jakiegoś API
- za pomocą funkcji map mapujemy każdy element listy na komponent Child (który minimalnie zmodyfikowałem dla tego przykładu)
- ponownie atrybut key jest tutaj mega istotny

Oczywiście zarówno zmienna children jak i data mogą być zmiennymi stanu. Takie podejście pozwoli dynamicznie manipulować ilością dzieci w trakcie działania aplikacji.





Callbacki

React to JavaScript i HTML, więc obowiązują tutaj standardowe eventy typu onClick. Wróćmy do jednego z poprzednich przykładów i dodajmy jakiś onClick.

```
5  const Parent = (props) => {
6      const print = () => {
7          console.log("Child has been clicked!");
8      }
9
10     return(
11         <div>
12             <span onClick={print}>Child #1</span><br />
13             <span onClick={print()}>Child #2</span>
14         </div>
15     );
16 }
```

- **na chwile jako dzieci użyłem zwykłych spanów**
- zrobiłem tak, ponieważ ten element (przyjmijmy, że jest to komponent) wie co to atrybut onClick oraz wie co przez niego przyjmuje
- span przez onClick przyjmuje referencję na funkcję, która ma zostać wywołana w momencie kliknięcia na nim

```
5  const Parent = (props) => {
6      const print = () => {
7          console.log("Child has been clicked!");
8      }
9
10     return(
11         <div>
12             <Child numer="1" onClick={print}></Child>
13             <Child numer="2" onClick={print()}></Child>
14         </div>
15     );
16 }
```

- w tym przypadku komponent Child nie wie nic o atrybucie onClick (jeszcze nie wie) więc klikanie nic nie da

Czym różni się zapis z linii 12 od tego z linii 13? Ten w linii 13 jest błędny. Powoduje on wywołanie funkcji print() w momencie budowania komponentu dlatego od razu w konsoli dostajemy nasz napis. Zapis z linii 12 jest poprawny, ponieważ przekazujemy do atrybutu onClick informację, którą funkcje należy uruchomić w momencie zdarzenia.

Jak przekazać parametr do takie funkcji skoro zapis z 13 linii jest zły??? Tak:

```
11             <div>
12                 <Child numer="1" onClick={() => print(1)}></Child>
13                 <Child numer="2" onClick={() => print(2)}></Child>
14             </div>
```

```
//przykładowy komponent dziecko
const Child = (props) => {
    return(
        <div onClick={props.onClick} style={{border: "1px solid black", margin: "10px", padd
            <h1>Child #{props.numer}</h1>
        </div>
    );
}
```

- kilka obrazków wyżej w atrybucie onClick podaliśmy do dziecka referencje na funkcję zdefiniowaną w rodzicu
- na tym obrazku widać jak przypisujemy te referencje do faktycznego onClick najbardziej zewnętrznego elementu w komponencie Child
- oczywiście props komponentu Child nie musi się nazywać onClick
- to my jesteśmy autorami komponentu – więc my decydujemy co i jak się nazywa

A tak to będzie wyglądało z parametrem:

```
const Parent = (props) => {
    const print = (num) => {
        console.log(`Child ${num} has been clicked!`);
    }

    return(
        <div>
            <Child numer="1" onClick={print}></Child>
            <Child numer="2" onClick={print}></Child>
        </div>
    );
}
```

- funkcja print przyjmuje parametr

```
//przykładowy komponent dziecko
const Child = (props) => {
    return(
        <div onClick={() => props.onClick(props.numer)} style={{borde
            <h1>Child #{props.numer}</h1>
        </div>
    );
}
```

- ale to w miejscu wywołania powinniśmy być tego świadomi
- to jest duży minus języków dynamicznie typowanych, że ja nie mam 100% pewności, że osoba używająca mojego komponentu poda w props.onClick referencję na funkcję a nie np. string
- bo błąd i tak poleci w moim kodzie