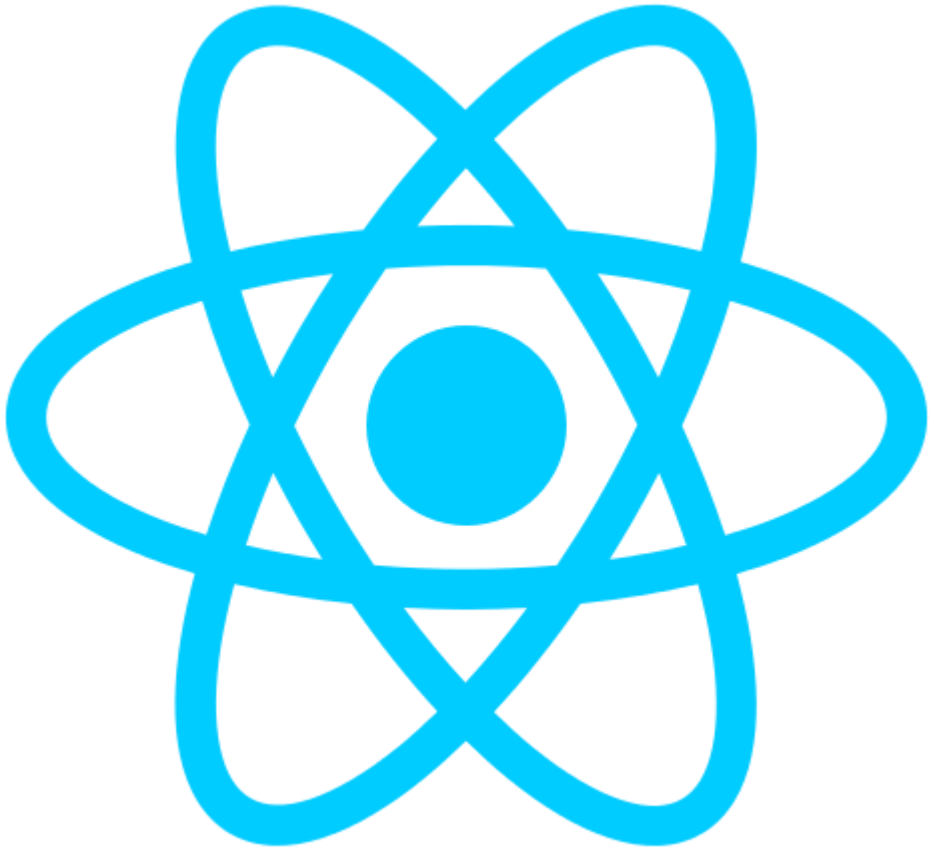
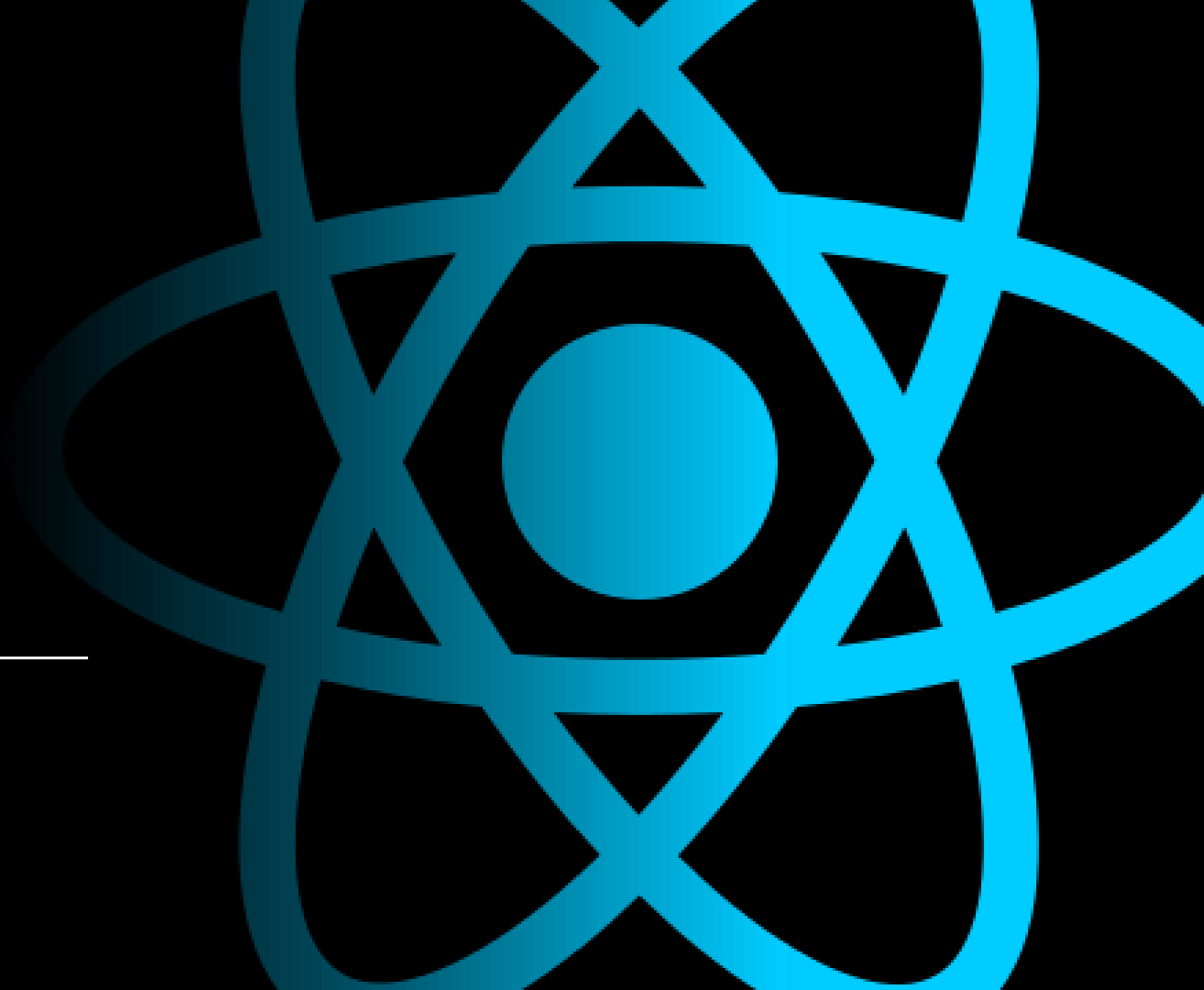


1. Wstęp do JSX
2. Wyrażenia
3. Atrybuty elementów
4. Zagnieżdżanie dzieci
5. Listy i klucze

# Wstęp do JSX



JSX jest rozszerzeniem języka JavaScript pozwalającym używać tagów przypominających tagi HTML wewnątrz plików JavaScriptowych.

Tag JSX nie jest ani stringiem, ani czystym kodem HTML. Składnia ta powstała tylko po to by ułatwić pisanie i zwiększyć czytelność kodu. Jest zalecana do użycia z React. Po skompilowaniu do czystego JS jest to zwyczajny kod korzystający z React.

Tagi JSX muszę spełniać poniższe warunki:

- Muszą być poprawnymi tagami. HTML dopuszcza nie wpisywanie cudzysłowu dla wartości atrybutu. JSX nie dopuszcza takiej możliwości
- Muszą być poprawnie zamknięte.

```
<img src=logo.png />

<img src="logo.png" />
```

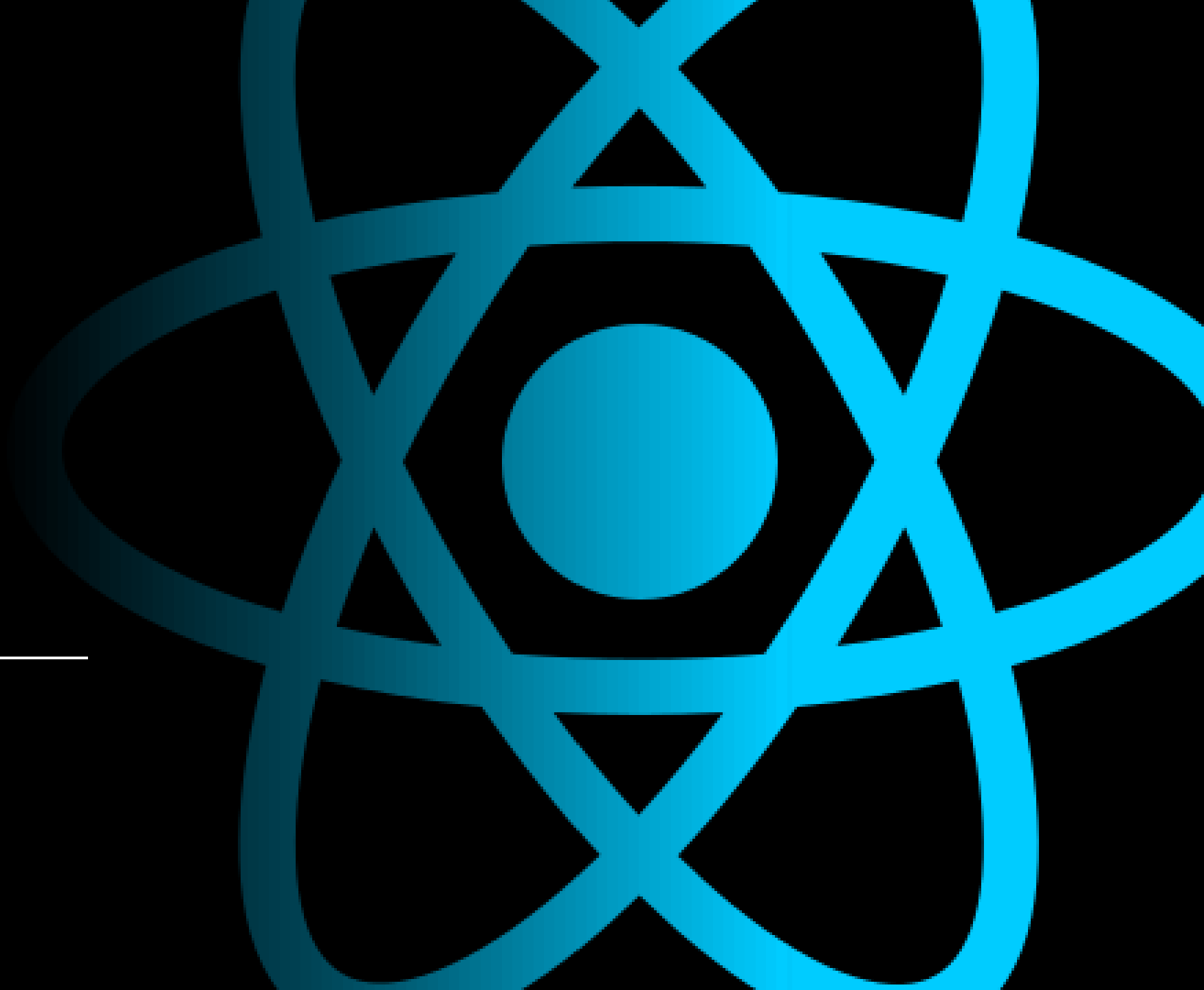
W trakcie kompilacji nasz kod zostanie zmieniony w odpowiednie wywołania metody React.createElement().

```
<span id="test">Hello, World</span>
```

```
React.createElement(
    "span",
    { id: "test" },
    "Hello, World"
);
```

```
{
    type: 'span',
        props: {
        id: 'test',
            children: 'Hello, World!'
    }
}
```

# Wyrażenia



Największą potęgą **JSX** jest możliwość zagnieżdżania wyrażeń, czyli dowolnego kodu JS, który coś zwraca.

Wystarczy umieścić wyrażenie w dowolnym miejscu elementu i otoczyć je klamrami.

```
<span id="test">{2 + 2}</span>
```

Dzięki takiemu podejściu w wyrażeniach możemy umieszczać w zasadzie dowolne dane do których mamy dostęp

```
const name = 'Matesz';
const count = name.length;
const person = {
    name: 'Mateusz',
    age: 32
}
```

```
<span>Twoje imię to {name} i ma {count} znaków</span>

<span>Twoje imię to {person.name} i masz {person.age} lat</span>
```

Możemy również użyć funkcji jeśli coś zwracają oczywiście. Dodatkowo możemy używać funkcji wbudowanych w native JS

```
function greeting(firstName: string) {
    return `Witaj ${firstName}`;
}
const name = 'Mateusz';
```

```
<span>{greeting(name)}</span>

<span>{name.toUpperCase()}</span>
```

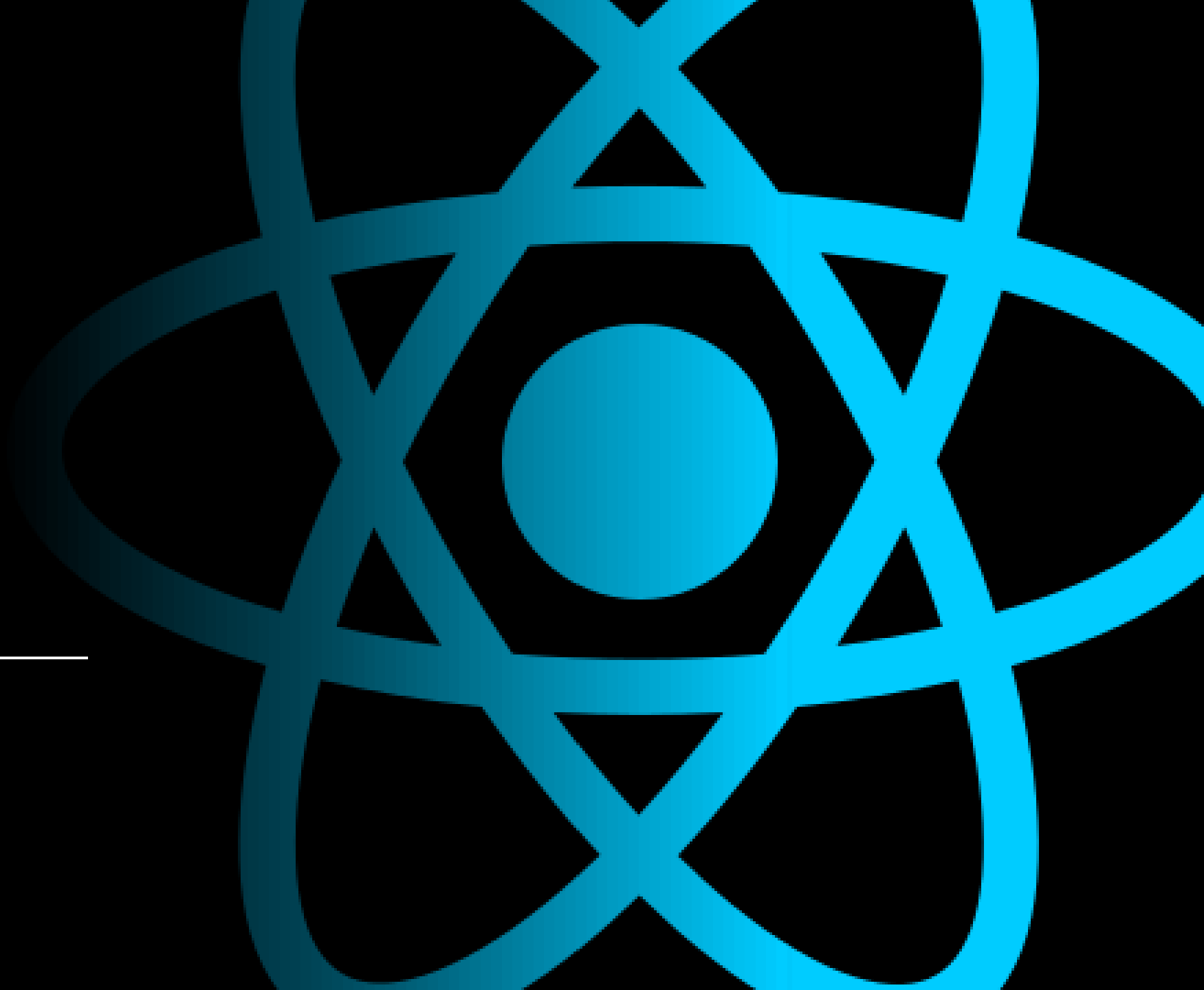
No i oczywiście możemy korzystać z wyrażeń warunkowych oraz map(), filter()

```
const array = ['Mateusz', 'Mariusz', 'Ilona'];
const name = 'Mateusz';
```

```
{array.map((elem: string) =>
    <span>{elem}</span>
)}

<span>{name.length > 0 ? 'Test1' : 'Test2'}</span>
```

# Atrybuty w tagach



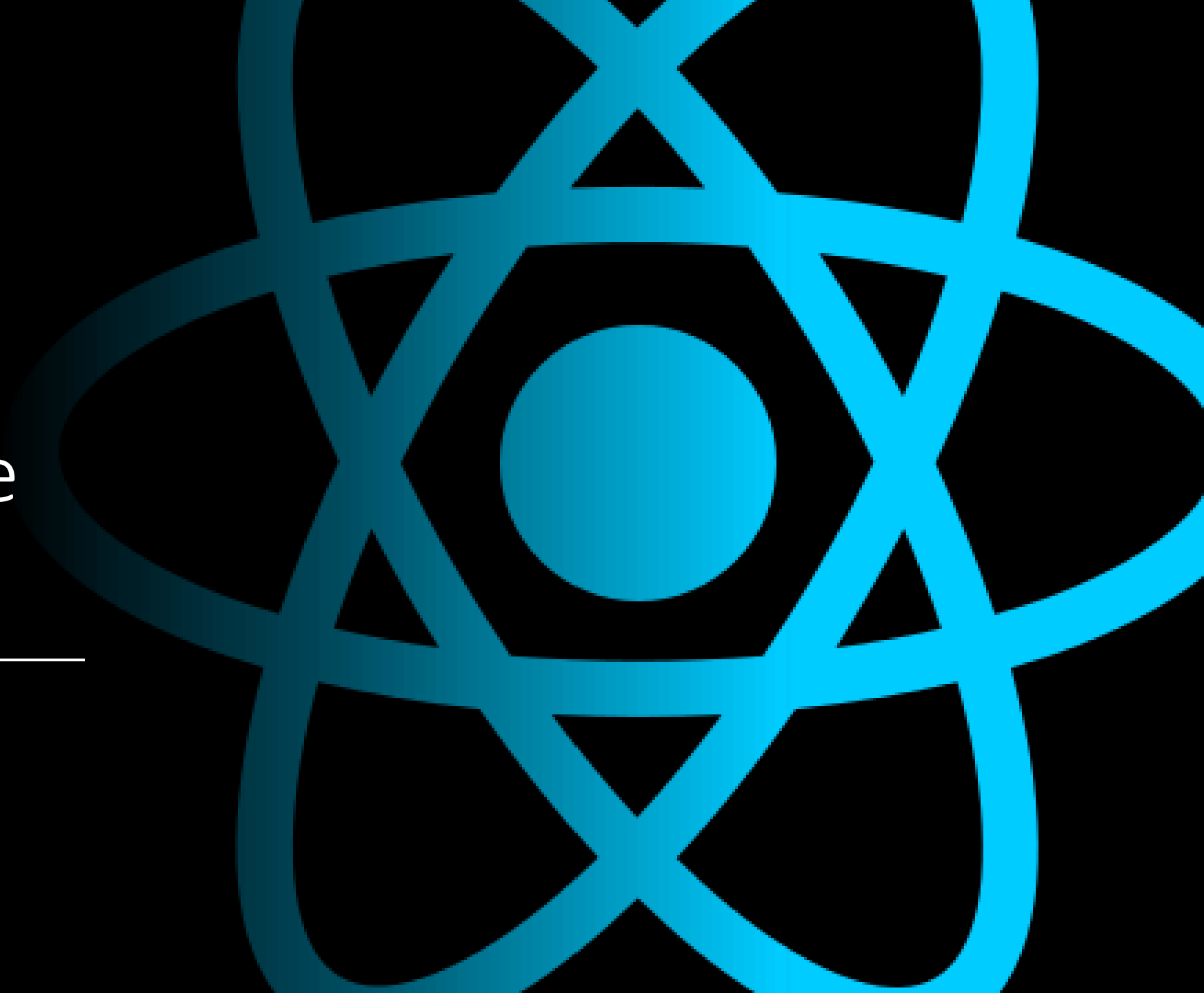
Tag JSX nie jest elementem drzewa DOM, tylko jego reprezentacją. Dlatego atrybuty tagu JSX nie zawsze są identyczne jak te w HTML. Nazewnictwo atrybutów w JSX jest takie samo jak w przypadku operacji w native JS na DOM-ie, z tym że przekształcamy pisowanie z myślnikami lub z małych liter na camelCase.

```
<div className="App">
    <header className="App-header">
        <img src={logo} className="App-logo" alt="logo" />
```

## W atrybutach możemy umieszczać również wyrażenia

```
<div contentEditable={ true }>Sample</div>
<input minLength={ 10 } />
<input value={ name } />
```

# Zagnieżdżanie dzieci



Należy zapamiętać, że w JSX zawsze **należy zwrócić jeden główny tag**, w środku może być zagnieżdżona ich dowolna liczba.

Jeżeli potrzebujesz zwrócić większą ilość elementów – pamiętaj aby umieścić je w jakimś kontenerze, najczęściej w elemencie div.

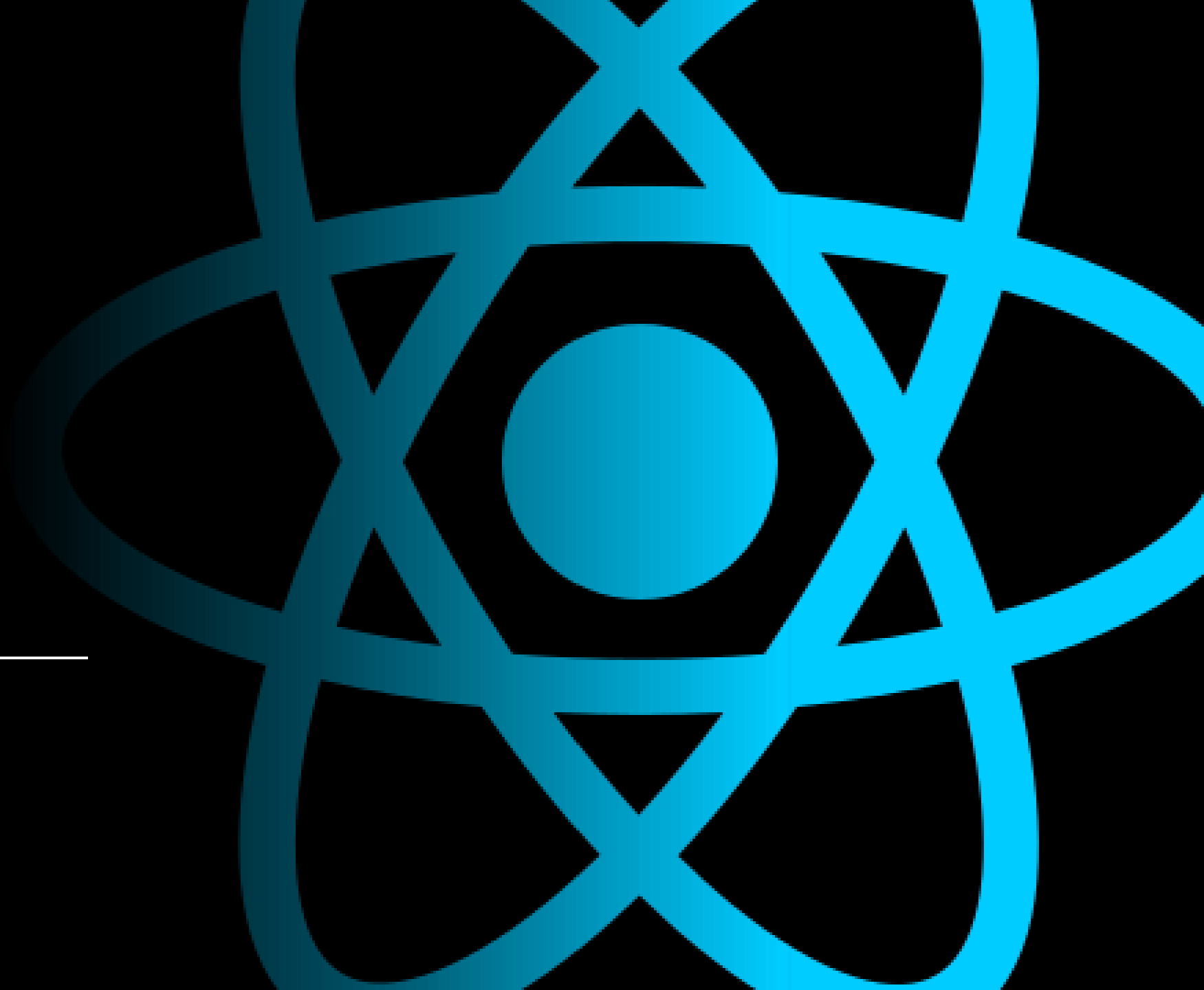
React od wersji 16.4 wprowadził obiekt, który nazwano Fragment. Może on zastąpić diva lub każdy inny element HTML. Mamy dwie metody zapisu tego elementu.

```
import React, {Fragment} from "react";
```

```
{array.map((elem: string) =>
    <Fragment>
        <span>{elem}</span>
        <span>Test</span>
    </Fragment>
)}
```

```
{array.map((elem: string) =>
    <>
        <span>{elem}</span>
        <span>Test</span>
    </>
)}
```

# Listy i klucze



Przy pomocy metody map() która jak wiesz zwraca nam nową tablicę, bardzo łatwo jest generować dynamicznie wiele elementów. Jednak jeśli zrobimy to w ten sposób w consoli dostaniemy błąd

```
▶Warning: Each child in a list should have a unique "key" prop.

Check the render method of `App`. See https://reactjs.org/link/warning-keys for more information.
    at App
```

```
{array.map((elem: string) =>
    <>
        <span>{elem}</span>
        <span>Test</span>
    </>
)}
```

Jak już wspomniałem React automatycznie porównuje zmiany w elementach, dzięki czemu w rzeczywistym DOM-ie są dokonywane wyłącznie niezbędne zmiany.

Niby proste ale…

Żeby zrealizować zadanie porównania zmian programiści Reacta zbudowali specjalny algorytm heaurystyczny, który jest naprawdę bardzo wydajny, ale opiera się na pewnych odgórnych założeniach.

Żeby zrozumieć jak bardzo jest wydajny ten algorytm najpierw zobaczymy jak działają algorytmy porównania zmian w drzewach.

Najwydajniejsze z nich potrafią mieć tak niską złożoność jak 0(n^3), czyli że liczba operacji porównania jakie należy wykonać to liczba elementów w drzewie do potęgi trzeciej.

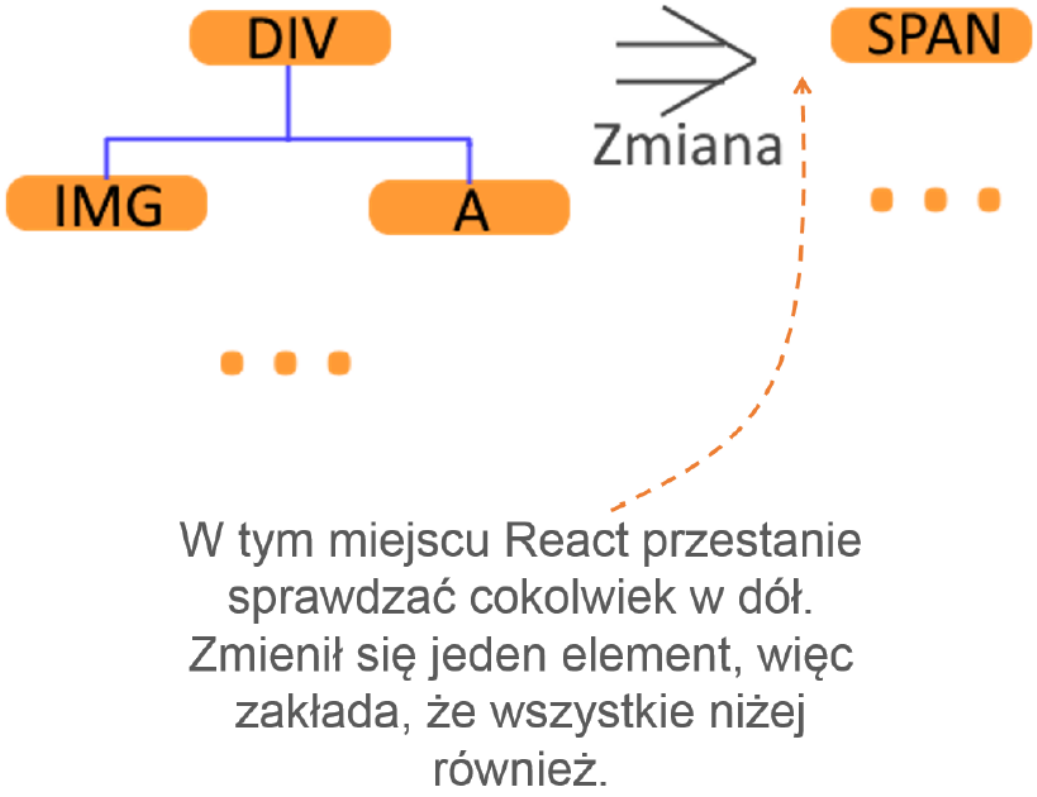
**Bardziej obrazowo:** jeśli na stronie mamy **1000 elementów** to należy wykonać **1 000 000 000** (miliard) porównać. Czyli bardzo dużo czasu na obliczenia.

Algorytm Reacta ma złożoność 0(n) czyli liczba operacji porównania jest równa liczbie elementów w drzewie, czyli w naszym przypadku 1000 porównań (milion razy mniej).

Ten wzrost wydajności wynika właśnie z heaurystyki, czyli „wiedzy” silnika Reacta, w jaki sposób najczęściej postępując programiści. Dlatego powinniśmy jako programiści przestrzegać przynajmniej dwóch zasad:

**Elementy różnego typu produkują różne wyjście.**

W praktyce oznacza to, że jeżeli zmieniasz w jakimś miejscu drzewa DOM element na inny, to React zakłada, że zmienia się całe rozgałęzienie od tego elementu w dół.



A więc jeśli mamy:

```
<div>
    <div>
        <article>Lorem ipsum</article>
    </div>
</div>
```

I zamieniamy na:

```
<section>
    <div>
        <article>Lorem ipsum</article>
    </div>
</section>
```

To dla Reacta zmiana elementu głównego oznacza, że zmienia się wszystko w środku.

Stary article i div zostaje usunięty z DOM w całości. Następnie zostanie stworzony nowy div i article i wstawiony do DOM

**Programista może wskazać, które elementy mogą być stabilne pomiędzy zmianami za pomocą atrybutu key**

Brak takiego klucza może sprawić, że porównania będą bardzo niewydajne bo React uzna że zamiast aktualizacji jednego elementu trzeba zaktualizować wszystkie.

Mamy taki JSX:

```jsx
<ul>
    <li>Mateusz</li>
    <li>Jan</li>
    <li>Marcin</li>
</ul>
```

A następnie taki:

```jsx
<ul>
    <li>Mateusz</li>
    <li>Jan</li>
    <li>Marcin</li>
    <li>Józef</li>
</ul>
```

Taka zmiana przebiegnie poprawnie ze względu na to jak działa algorytm porównania dla elementów rodzeństwa.

**Lista jest przeszukiwana od góry do dołu i szukana jest zmiana.**

React poprawnie rozpozna że należy dodać jeden element na końcu listy.

Niestety ale to jedyny przypadek gdzie React poradzi sobie z listą.

Mamy taki JSX:

```
<ul>
    <li>Mateusz</li>
    <li>Jan</li>
    <li>Marcin</li>
</ul>
```

A następnie taki:

```
<ul>
    <li>Józef</li>
    <li>Mateusz</li>
    <li>Jan</li>
    <li>Marcin</li>
</ul>
```

W tej sytuacji React na samej górze rozpozna, że elementy są różne i wszystko co porówna w dół – również. React uzna więc, że należy usunąć wszystkie 2 stare elementy i wstawić wszystkie 3 nowe elementy. Bez sensu. Ale…

Możemy ręcznie podpowiedzieć Reactowi, które elementy listy są cały czas tymi samymi elementami

Do tego celu używamy do każdego powtarzającego się elementu/komponentu atrybut key.

Czyli mamy JSX:

```jsx
<ul>
    <li key="Mateusz">Mateusz</li>
    <li key="Jan">Jan</li>
    <li key="Marcin">Marcin</li>
</ul>
```

A następnie taki:

```jsx
<ul>
    <li key="Józef">Józef</li>
    <li key="Mateusz">Mateusz</li>
    <li key="Jan">Jan</li>
    <li key="Marcin">Marcin</li>
</ul>
```

Klucz **musi być unikalny w ramach tej jednej listy w której występuje**

**Najlepsze klucze to:**

Klucz musi być wartością jednoznacznie identyfikującą dany element. Zazwyczaj sprawa jest prosta i np. wyświetlając userów z bazy mamy ich id. Jeśli nie to użyj jakiejś innej unikalnej właściwości. Ostatecznie może być to kolejny index elementu.

```
{array.map((elem: string, index: number) =>
    <span key={index}>{elem}</span>
)}
```

**Pamiętaj o atrybucie key w elementach list.**

Klucze muszą być unikalne w ramach jednej listy.

Szukaj możliwych key następująco: id, unikalny element, indeks

Brak kluczy lub niestabilne klucze (losowe) mogą doprowadzić do znacznego obniżenia wydajności podczas renderowania list.