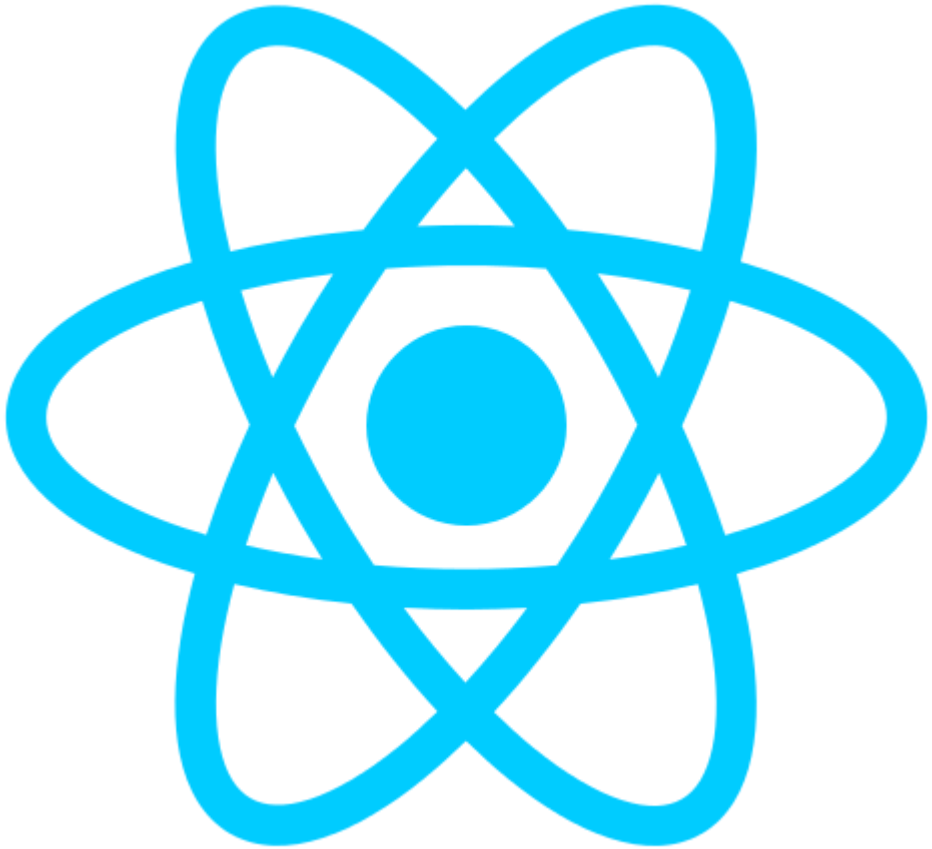
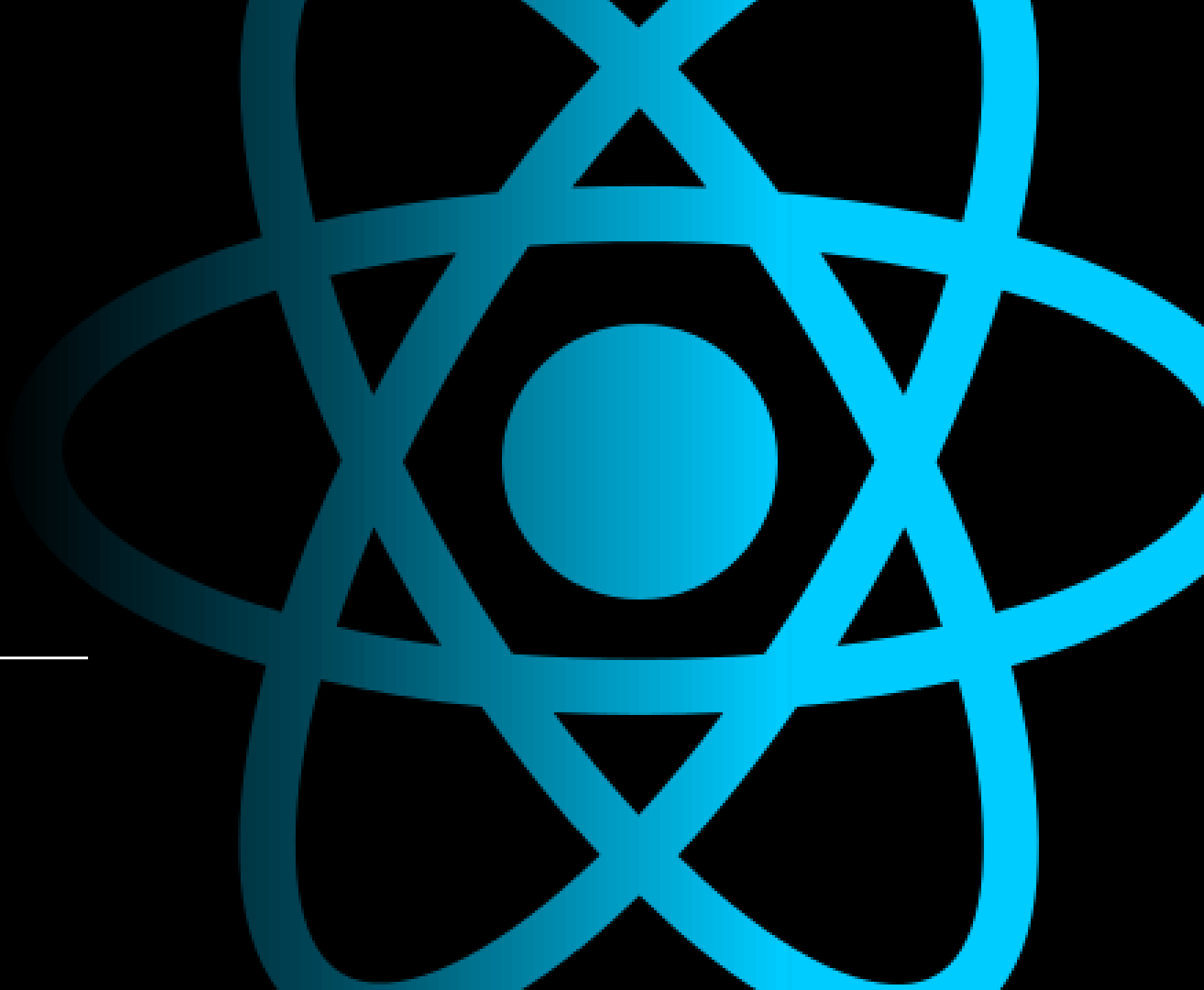


1. Czym jest komponent
2. Komponent jako funkcja
3. Komponent jako klasa
4. Renderowanie komponentu
5. Props
6. Kompozycja
7. Pure function a props
8. Klucze i komponenty

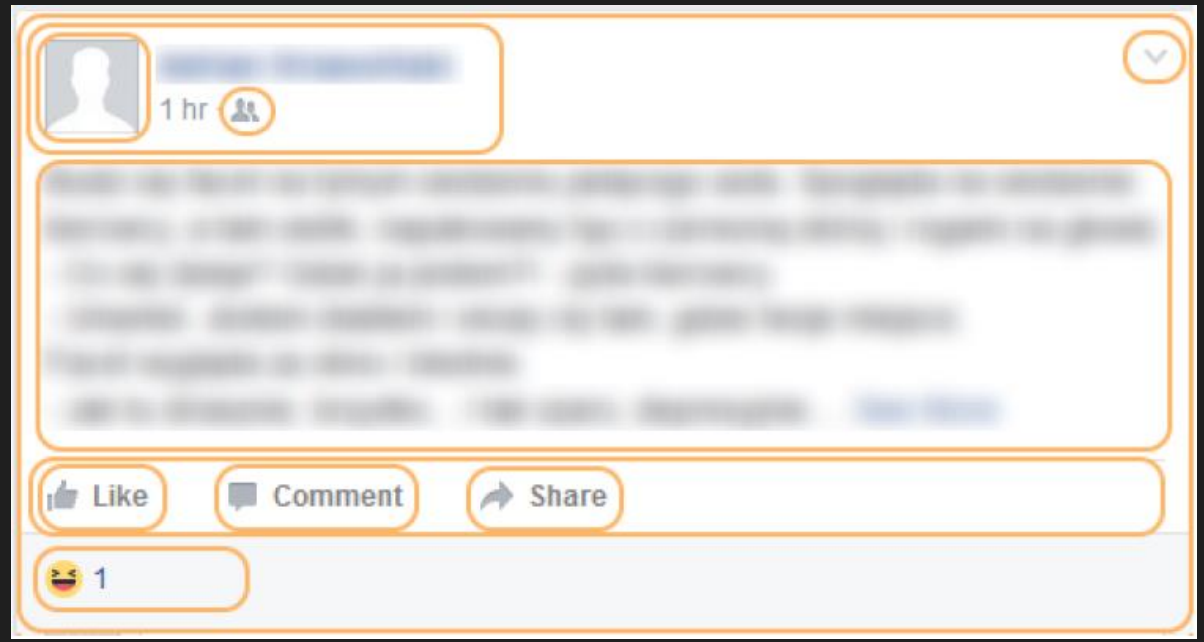
Czym jest komponent



Komponent to pojedynczy element interfejsu usera odpowiedzialny za konkretną część aplikacji. Składa się z elementów i/lub innych komponentów.

Można myśleć o komponentach jak o swoich tagach HTML.

Po prawej mamy standardowy komponent postu na facebooku. Na pomarańczowo zostały zaznaczone kolejne komponenty

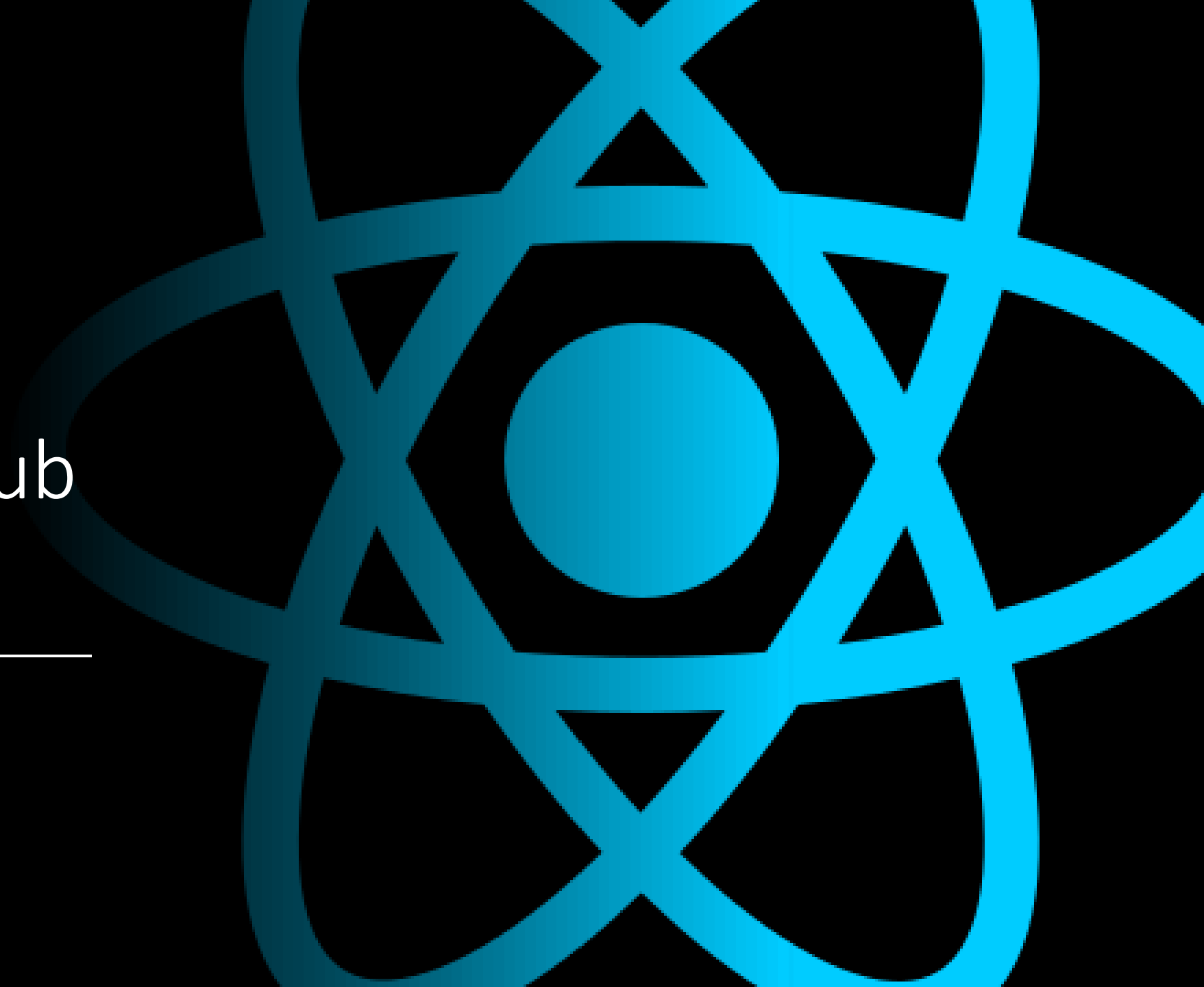


Ogólnie można ująć działanie komponentu w dwóch punktach:

1. Komponent (opcjonalnie) przyjmuje jakieś dane - atrybuty (props o których mowa później)
2. Zwraca nam Reactowe elementy lub inne komponenty, które chcemy wyświetlić

Aplikacje budowane z pomocą React składają się z komponentów i elementów. Element jest najmniejszą jednostką do budowania.

Komponent jako funkcja lub klasa



Pomimo iż zapis funkcyjny jest nowszy, bardziej elastyczny i wspierany to poznać należy oba zapisy. Jest to spowodowane tym że duża część projektów oraz bibliotek została zbudowana na komponentach klasowych co oznacza że czasem jest niezbędne użycie komponentu klasowego.

Ważne jest też poznanie wszystkich aspektów klasowych aby lepiej zrozumieć aspekty tak zwanych hooków w komponentach funkcyjnych.

Komponent funkcyjny możemy zapisać na kilka sposobów lecz w połączeniu z TypeScript najwygodniejszy jest taki:

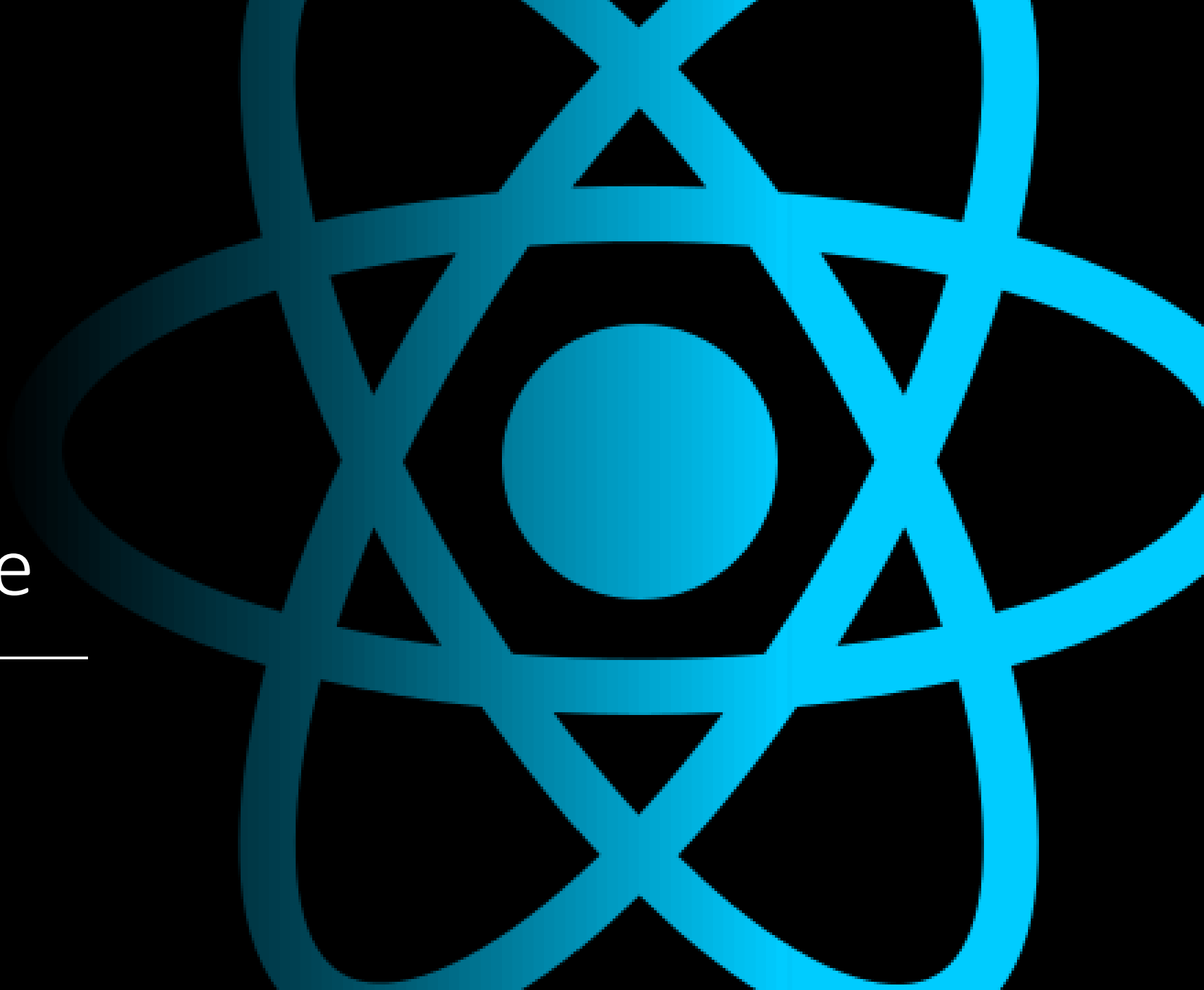
```tsx
import React, { FC } from 'react';
...
interface ITextContentProps {
    nameFromProps: string;
}

export const TextContent: FC<ITextContentProps> = props => {
    return (
        <div></div>
    )
}
```

Komponent klasowy jest rozszerzeniem klasy **Component** importowanej z React.

```tsx
import React, { Component } from 'react';

...
interface IHelloWorldProps {

}

...
interface IHelloWorldState {

}
...
export default class HelloWorld extends Component<IHelloWorldProps, IHelloWorldState> {
    render() {
        return <h1>Hello, World!</h1>;
    }
}
```

Każdy komponent Reactowy musi zwrócić jakąś wartość. Jeśli z jakiegoś powodu nie chcemy nic zwracać należy zwrócić null.

Renderowanie



Komponenty renderujemy dokładnie tak samo jak elementy. Należy tylko pamiętać o:

- Komponent musi znajdować się w odpowiednim, widocznym zakresie
- Wywołujemy go wpisując nazwę – tak jak ją zadeklarowaliśmy - z dużej litery
- Ma być poprawnym tagiem JSX – pamiętamy o zamknięciu i odpowiednich atrybutach.

Funkcyjnie

```
import React from "react";

const HelloWorld = () => {
  return <h1>Hello, World!</h1>;
}

ReactDOM.render(
  <HelloWorld />,
  document.getElementById("app")
);
```

Klasowo

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

class HelloWorld extends Component {
  render() {
    return <h1>Hello, World!</h1>;
  }
}

ReactDOM.render(
  <HelloWorld />,
  document.getElementById("app")
);
```

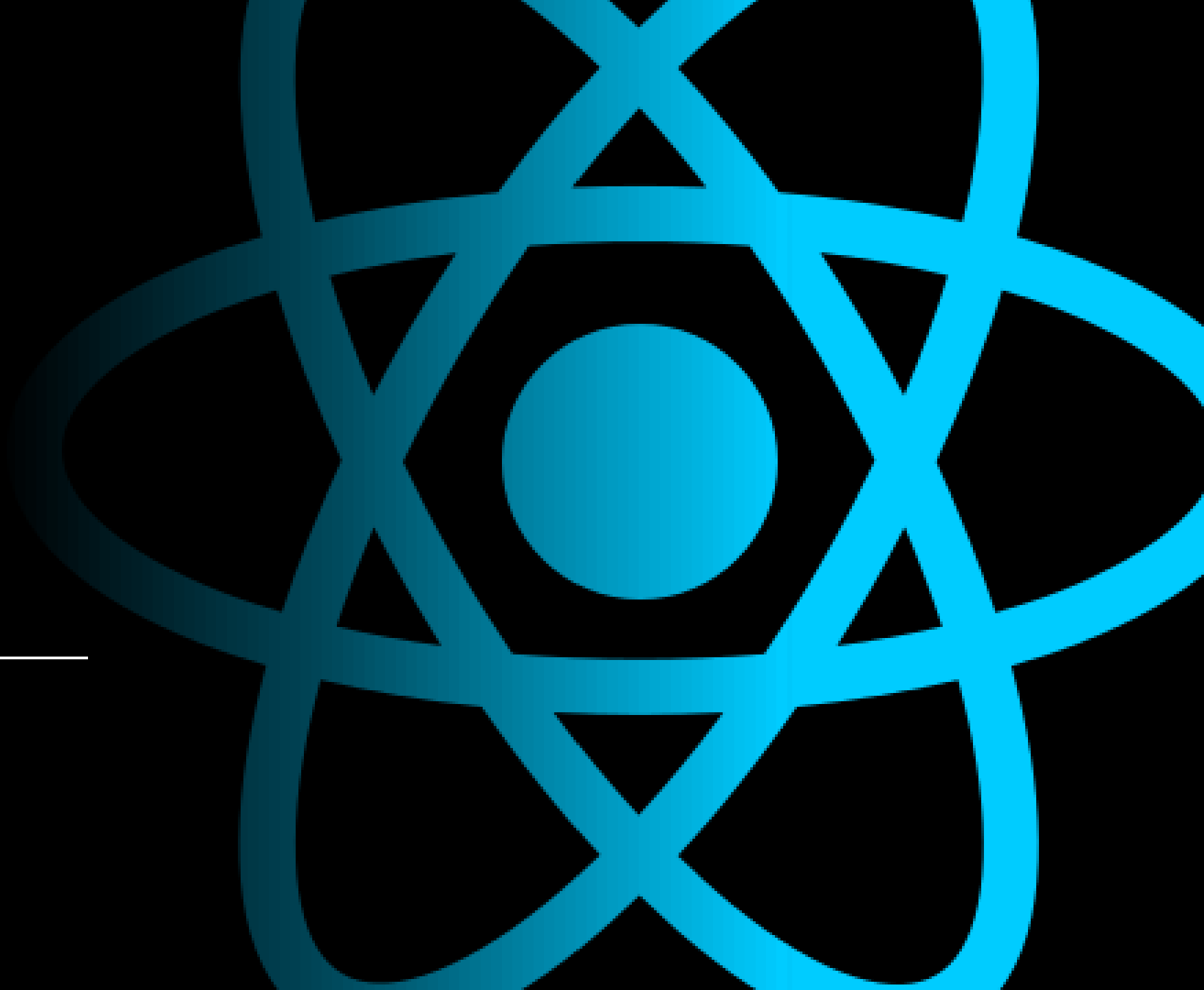
Funkcyjnie

```
import React from "react";

const Box = () => {
  return (
    <div className="box">
      <p>Krótki opis</p>
      <span>Autor, Data</span>
    </div>
  )
}

ReactDOM.render(
  <Box />,
  document.getElementById("app")
);
```

Klasowo

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

class Box extends Component {
  render() {
    return (
      <div className="box">
        <p>Krótki opis</p>
        <span>Autor, Data</span>
      </div>
    )
  }
}

ReactDOM.render(
  <Box />,
  document.getElementById("app")
);
```

Props



Wiemy już jak renderować koponenty, ale bardzo często chcemy aby to co w nich się znajduje było dynamiczne i zależne od czynników zewnętrznych, np. przesłanych parametrów.

Analogicznie jak w native JS funkcje mogą przyjmować argumenty.

W React mamy props

React po napotkaniu komponentu pobiera wszystkie jego atrybuty wpisane w nim i przekazuje je jako obiekt **props** do komponentu.

Sposób dostępu do props różni się w zależności od sposobu zapisu komponentu. Na szczęście przekazywanie props zawsze jest takie same.

```
<SetPropertiesIcon type={StakeDetailsType.Person} />

<FormattedMessage id="cluster.label.person" defaultMessage="Party" />
```

Funkcyjnie

```
const Greeting = (props) => {
  return <h1>Hi, {props.name}!</h1>;
}
```

Klasowo

```
class Greeting extends Component {
  render() {
    return <h1>Hi, {this.props.name}!</h1>;
  }
}
```

1. React napotyka na komponent
2. Budowany jest obiekt właściwości props
3. React znajduje definicję naszego komponentu
4. Przekazuje do niego zbudowany obiekt props

Oczywiście za pomocą props można przesyłać dowolne dane:

```
const Calc = (props) => {
  console.log(props.run); // true
  console.log(props.operation); // null

  return (
    <h1>{props.numberA + props.numberB}</h1>
  )
}
```

```
<Calc
  numberA={4}
  numberB={10}
  run={true}
  operation={null} />
```

```
const Alert = (props) => {
  return (
    <div>
      <h2>Alert!</h2>
      {props.message}
    </div>
  )
}

const user = "John";
```

```
<Alert message={`Hi, ${user}. You are logged in.`} />
```

Możemy przesłać nawet funkcje, co jest bardzo często robione:

```
const Message = (props) => {
  props.customFunction("Hello!"); // "Text from component: Hello!"

  return <h1>Hello!</h1>
}

const myFunc = (text) => {
  console.log(`Text from component: ${text}`);
}
```

```
<Message customFunction={myFunc} />
```

props.children jest specyficznym

propsem lecz bardzo łatwo nim

sterować

```
const App = () => {
  return (
    <Wrapper title="I am the wrapper">
      <Child body="Child component" />
    </Wrapper>
  );
};

const Wrapper = (props) => {
  return (
    <div className="wrapper">
      <h1>{props.title}</h1>
      {props.children}
    </div>
  );
};

const Child = (props) => {
  return <p>{props.body}</p>;
};
```

Destrukturyzacja często się przydaje przy przesyłaniu większych danych.

Komponent funkcyjny

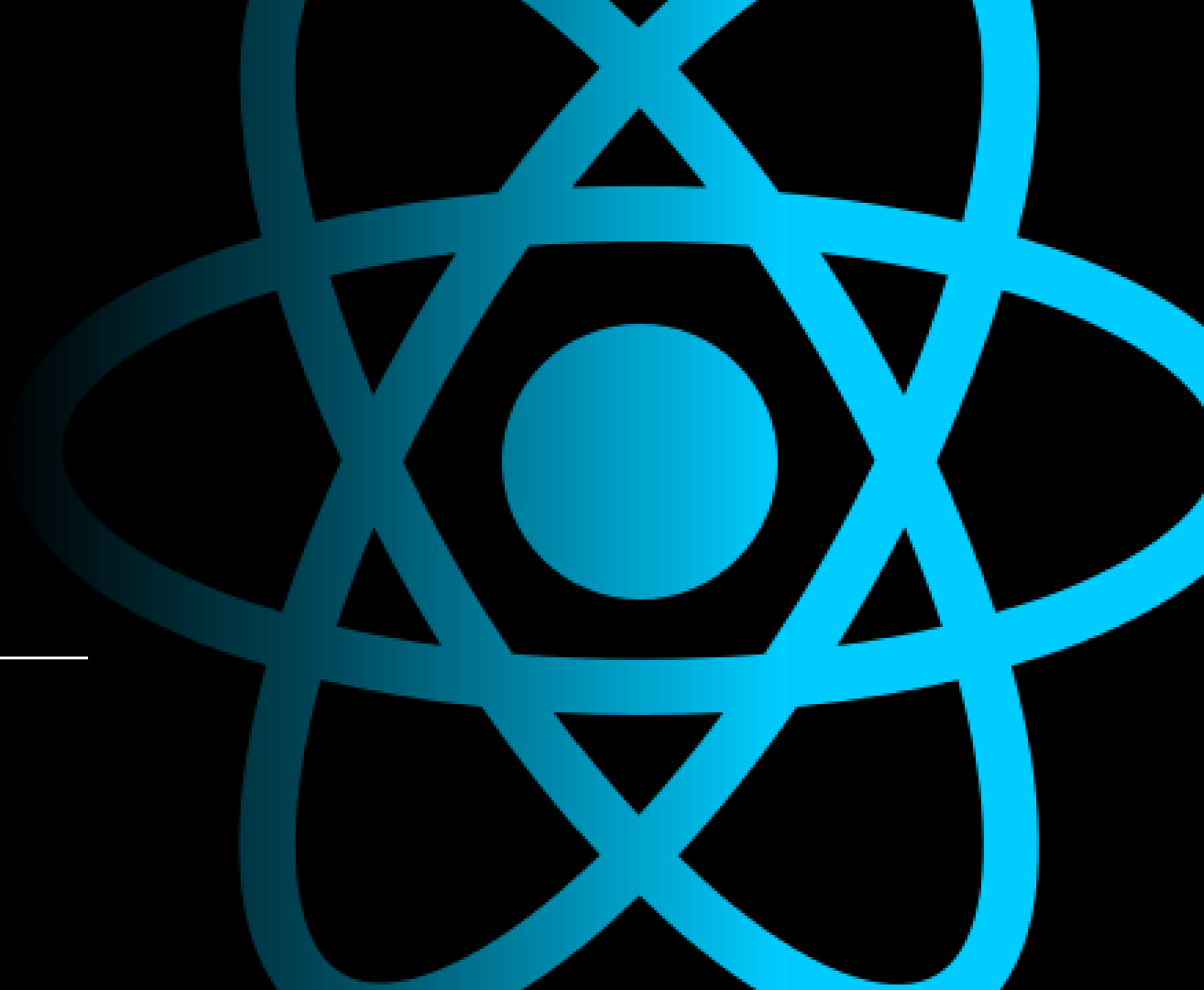
```
const Calc = (props) => {
  // Dokonujemy destrukturyzacji wewnątrz
  // funkcji odwołując się do props
  const {numberA, numberB} = props;
  return <h1>{numberA + numberB}</h1>
}
```

```
// Destrukturyzujemy obiekt już w
// momencie deklaracji parametrów
const Calc = ({numberA, numberB}) => {
  return <h1>{numberA + numberB}</h1>
}
```

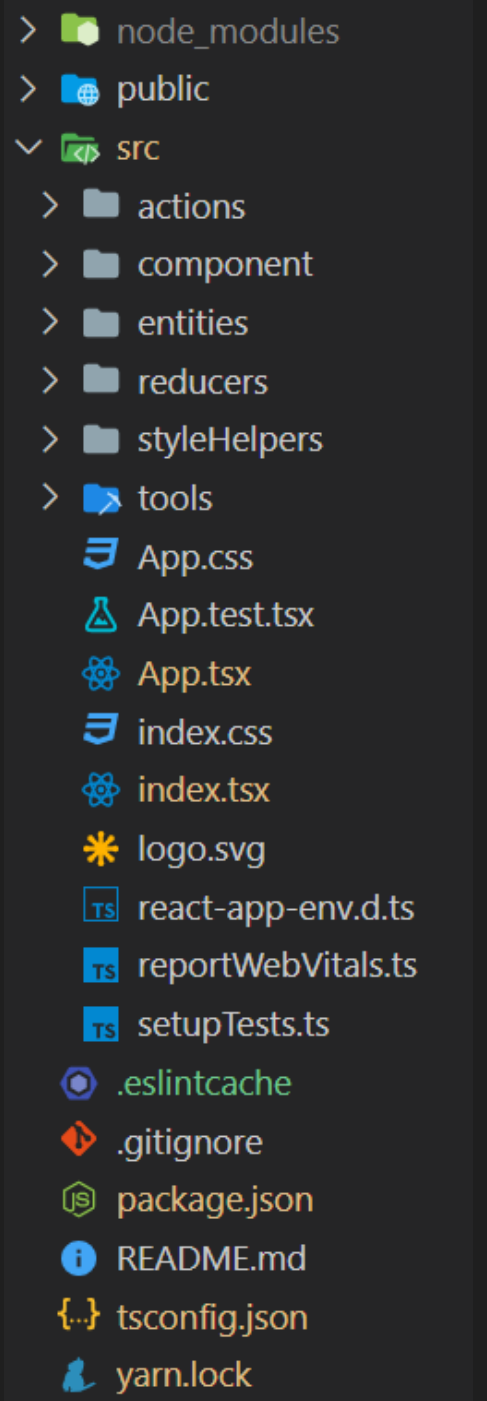
Komponent klasowy

```
class Calculator extends Component {
  render() {
    // Odwołujemy się do this.props
    const {numberA, numberB} = this.props;
    return <h1>{numberA + numberB}</h1>
  }
}
```

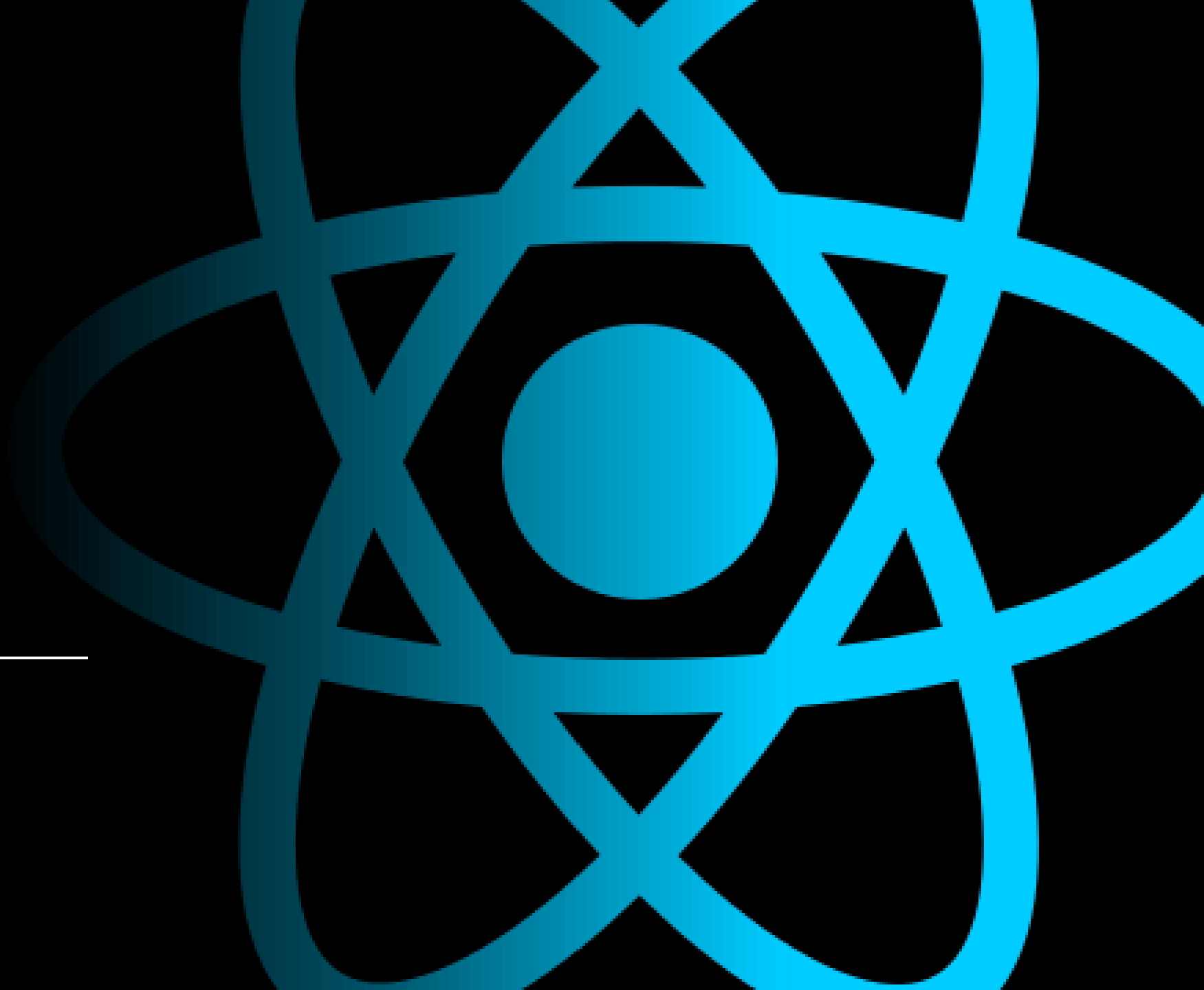
Kompozycja projektu



Jak wiecie – ilu programistów tyle kompozycji projektów. Jednak ja polecam taki układ folderów. Jest najczęściej używany wśród programistów React i najbardziej logiczny moim zdaniem.



Pure function



Pure function czyli funkcja, która zawsze zwraca te same dane gdy przyjmuje te same parametry oraz nie posiada skutków ubocznych.

Pierwsza część definicji odnosi się do działania funkcji – funkcja musi być przewidywalna. Jeżeli do funkcji przekażę jakieś argumenty to muszę być pewny że zawsze zwróci mi to samo przy tych samych parametrach

```
// Pure function
const add = (a:number, b: number) => a + b;
add(2, 5) // return 7
```

Druga część definicji odnosi się do środowiska, w jakim istnieje funkcja.

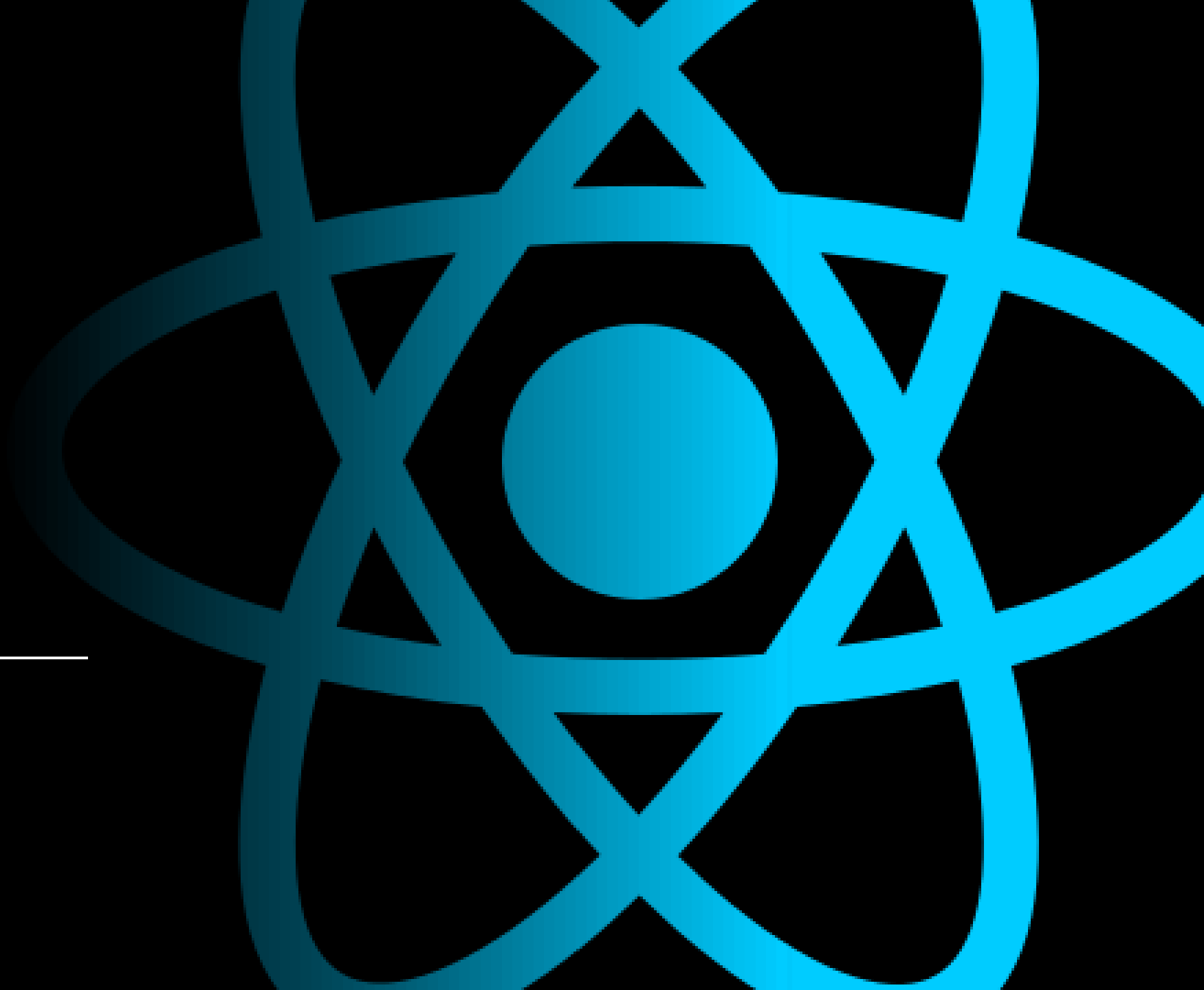
Funkcja czysta nigdy nie modyfikuje zmiennych w zakresach nadrzędnych.

Przykład z prawej nie jest czystą funkcją ponieważ modyfikuje zmienną z innego

zakresu oraz zmienne przekazane nie są bezpośrednio do funkcji.

```
const a = 2;
const b = 5;
let result = 0;

const add = () => {
    result = a + b;
    return result;
};

add() // return 7
```

Pure function a props



Mówiliśmy już, że zasada działania komponantu opiera się o funkcję: przyjmuje parametry (props) i zwraca wynik (w postaci komponentów lub elementów)

Uściślę nieco definicję props:

Komponenty w stosunku do props mają zachowywać się jak czyste funkcje.

Wynika z tego bardzo ważna zasada: komponent nie może w żadnej chwili zmienić niczego, co otrzymał w swoich props.

Nie ma od tego odstępst!

Mówiliśmy już, że zasada działania komponantu opiera się o funkcję: przyjmuje parametry (props) i zwraca wynik (w postaci komponentów lub elementów)

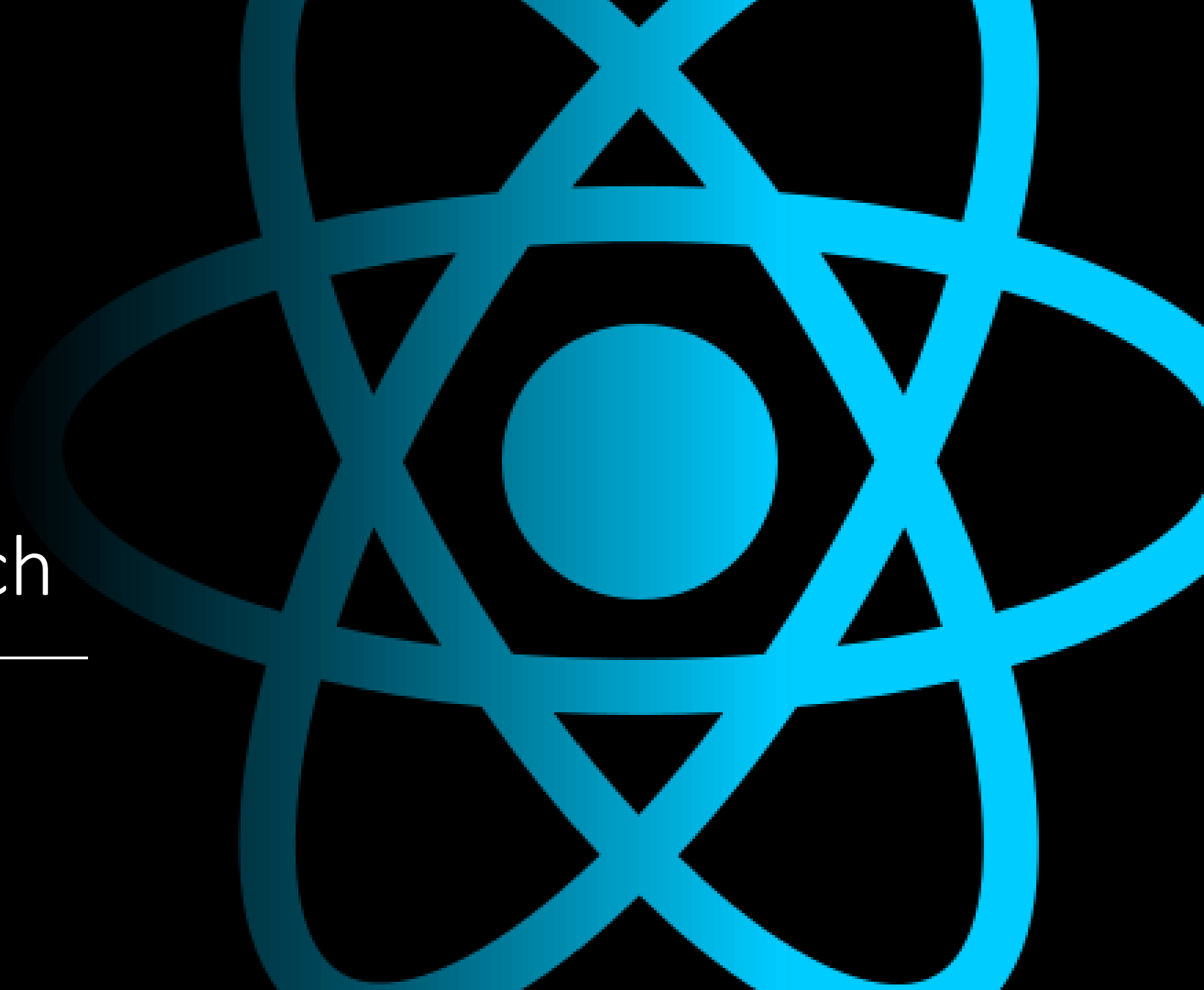
Uściślę nieco definicję props:

Komponenty w stosunku do props mają zachowywać się jak czyste funkcje.

Wynika z tego bardzo ważna zasada: komponent nie może w żadnej chwili zmienić niczego, co otrzymał w swoich props.

Nie ma od tego odstępst!

Key w komponentach



Atrybut key przekazywany do dowolnego komponentu nie jest widoczny w props tego komponentu.

Wynika to z tego, że atrybut key jest wykorzystywany wewnętrznie przez React a nie jest przekazywany do props.