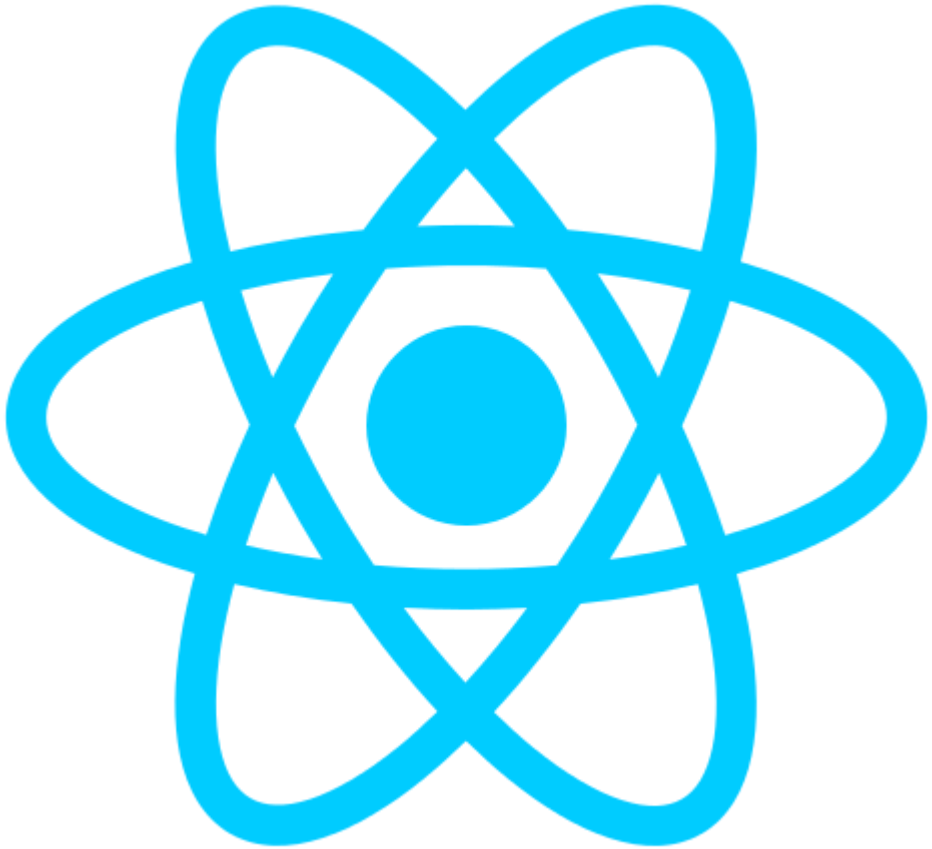
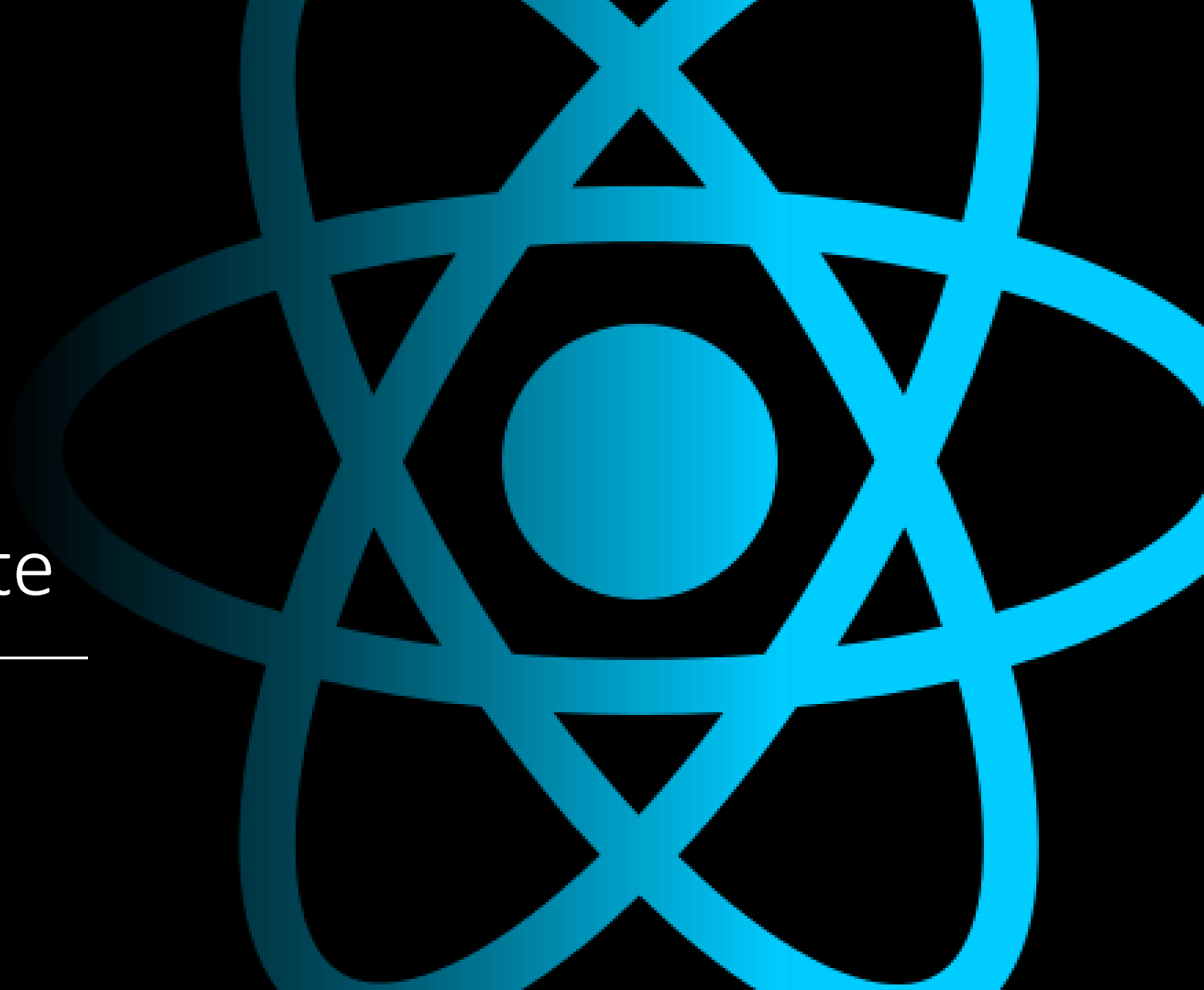


1. Wstęp do state
2. State klasowo
3. Cykl życia komponentu

Wstęp do state



State (z ang. "stan") jest miejscem do przechowywania **aktualnego wewnętrznego stanu danego komponentu**.

Posiadać go może każdy komponent.

W przeciwieństwie do props – **state może być modyfikowany przez komponent z jego wnętrza i do tego właśnie służy**.

Do czego służy `state`?

Będziemy go wykorzystywać **zawsze** kiedy będzie wymagana aktualizacja treści naszego komponentu na podstawie jakiejś akcji czy czynnika zewnętrznego, np:

- Zwiększenie licznika poprzez kliknięcie w `button` i wyświetlenie aktualnej wartości na stronie
- Wyświetlenie aktualnej godziny i aktualizowanie jej co sekundę
- Ukrywanie czy pokazywanie części komponentu

props

- Służy do przekazywania dzieciom danych właściwości
- Idzie z góry na dół (z rodzica do dziecka)
- Nie może być modyfikowany z wewnątrz komponentu, który go otrzymał (niemutowalny)
- Jest jak parametry funkcji

state

- Służy do przechowywania własnego stanu komponentu
- Może być modyfikowany z wewnątrz (mutowalny)
- Jest jak zmienne tymczasowe w funkcji*

* Jest to określenie przybliżone, gdyż funkcje zwykle nie przechowują swojego stanu i nie ma dokładnego porównania 1:1.

Nie ma wymogu aby komponent w ogóle korzystał z wewnętrznego stanu.

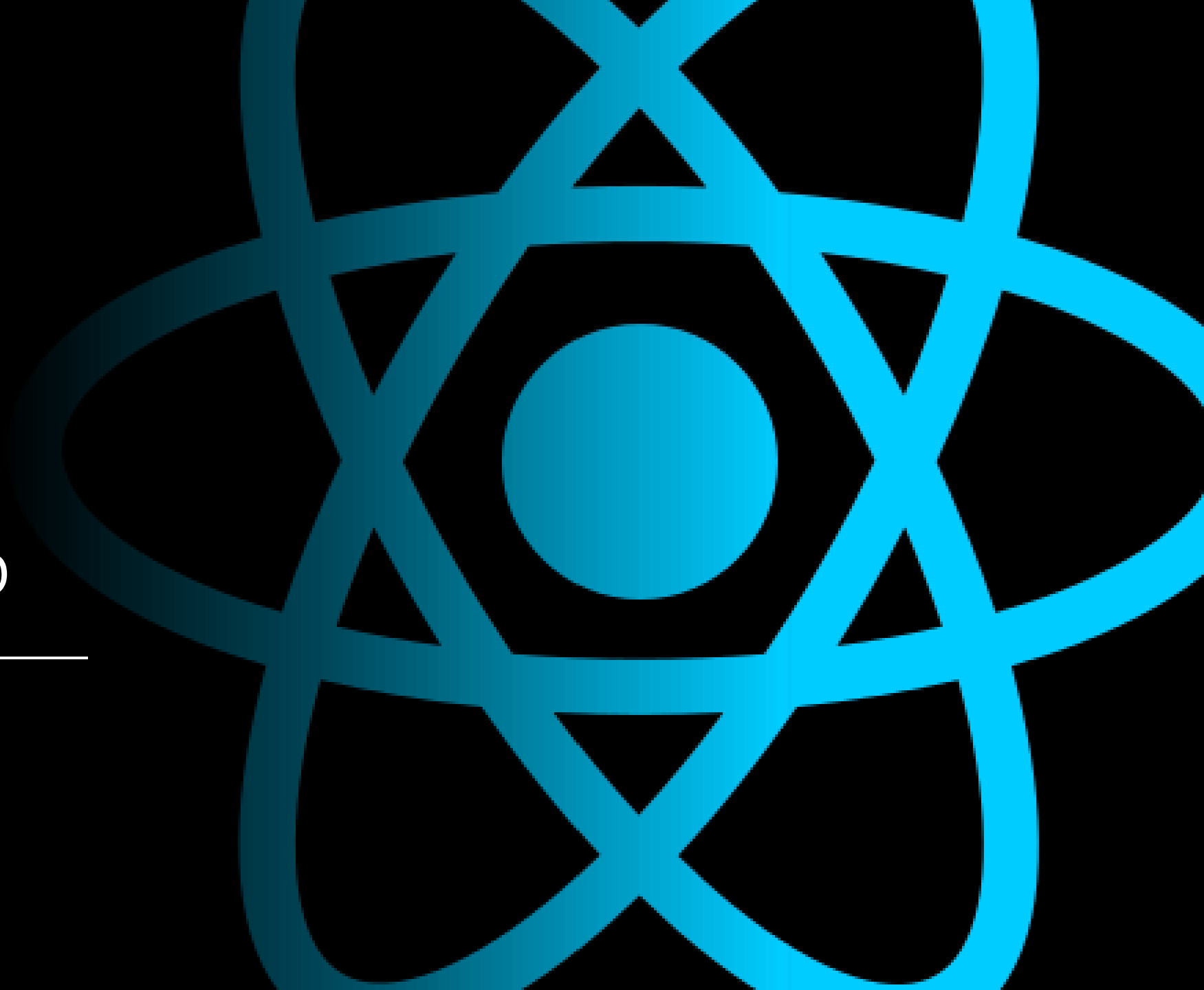
W praktyce zobaczysz, że nie każdy komponent wymaga posiadania takiego **state**.

Komponenty ze stanem mogą korzystać z tych bez stanu i vice versa.

Komponenty, które korzystają ze state, nazywamy **komponentami ze stanem** (częściej spotkasz się z ang. nazwą: **stateful component**).

Komponenty bez stanu wewnętrznego nazywamy **komponentami bezstanowymi** (częściej spotkasz się z ang. nazwą: **stateless component**).

State klasowo



Aby móc korzystać ze stanu musimy najpierw go zadeklarować:

```
...
class GlobalDirectory extends React.Component<IGlobalDirectoryProps> {

    state = {

    }
```

Dostęp do state jest analogiczny jak w przypadku props

```
state = {
    counter: 0
}
```

```
render() {
    return (
        <p>{this.state.counter}</p>
    );
}
```

Jednak nasz przykład nie robi nic spektakularnego. Po prostu wyświetla jakieś wartości. Jak teraz zmienić wartość stanu tak aby nasz licznik `counter` zwiększał się o 1 za każdym razem kiedy klikniemy w przycisk?

Kluczową rzeczą jest aktualizacja wewnętrznego stanu (state) komponentu.

Na początku trochę teorii: każda aktualizacja `state` powoduje ponowne uruchomienie metody `render()`.

W praktyce oznacza to, że Twój komponent jest na nowo renderowany kiedy zostanie zmieniony jego stan wewnętrzny.

Jest to oczywiście zgodne z logiką – kiedy zmieniasz stan, chcesz odświeżyć widok.

`this.setState`

Dzięki dziedziczeniu po `Component`, w naszym komponencie możemy skorzystać z metody `setState`, która w swojej najprostszej wersji wygląda w ten sposób:

```
this.setState({
  key: value
});
```

Przyjmuje ona jako parametr obiekt z parami kluczy i wartości które chcemy zaktualizować w naszym stanie.

Przesyłamy w tym obiekcie tylko te klucze i wartości które poddajemy zmianie.

`this.setState`

Możemy użyć również zapisu funkcyjnego.

Musimy zwrócić w niej obiekt z nowym stanem. Użycie funkcji pozwala nam również skorzystać z najbardziej aktualnego stanu i propsów. Otrzymujemy te dwie rzeczy jako parametry funkcji:

```
this.setState((prevState, props) => {
  return {
    key: value
  }
});
```

Przejdźmy do naszego przykładu a następnie zobaczymy, jak dokładnie działa ta metoda.

Dodajmy do naszego przykładu komponentu `Counter` przycisk a także metodę która będzie wyświetlała w konsoli napis "Działa", po każdym kliknięciu.

```
class Counter extends Component {
  state = {
    counter: 0
  }

  handleClick = () => {
    console.log("Działa");
  }

  render(){
    return (
      <>
        <h2>Twoje kliknięcia: {this.state.counter}</h2>
        <button onClick={this.handleClick}>Kliknij!</button>
      </>
    );
  }
}
```

Skupimy się teraz na samej metodzie `handleClick`.

Reaguje ona już na kliknięcie w przycisk, dlatego wystarczy, że teraz uruchomimy w niej metodę `this.setState`.

Co musimy zmodyfikować w stanie, żeby poprawnie rozwiązać to zadanie? Klucz `counter` powinien się zwiększać. Dlatego:

- Wyciągniemy jego aktualną wartość ze `state`
- Dodamy do niego `1`
- Zapiszemy z powrotem w `state` naszej aplikacji

```
 handleClick = () => {
  console.log("Działa");

  // Wersja funkcyjna
  this.setState( (prevState) => {
    return {
      counter: prevState.counter + 1
    }
  });
}
```

Z poprzedniego przykładu wynika podstawowe działanie metody, z której możemy korzystać w komponencie - `this.setState()`.

Aktualizuje ona stan wewnętrzny aplikacji, co wymusza aktualizację wyglądu komponentu.

Jest kilka rzeczy, na które należy zwrócić uwagę podczas korzystania z tej metody.

Omówimy je na kolejnych slajdach.

Przechowywany stan nie musi być obiektem z jednym prostym elementem. `State` może być obiektem zawierającym wiele wartości – elementów dowolnego typu.

Spójrz na ten przykład (nie musisz się póki co przejmować wartościami):

```
this.state = {
  counter: 0,
  startedAt: new Date(),
  currentUser: this.props.user
};
```

Mamy taki oto kod do aktualizacji, taki jak w naszym poprzednim przykładzie:

```
this.setState( (prevState) => {
  return {
    counter: prevState.counter + 1
  }
});
```

React inteligentnie połączy stary stan z nowym i docelowo będzie on wyglądał tak:

```
{
  counter: 1,
  // Poniższe dane pozostaną bez zmian,
  // ponieważ ich nie aktualizowaliśmy
  startedAt: stara_wartosc,
  currentUser: stara_wartosc
}
```

Drugą bardzo ważną informacją jest to, że wywołania `this.setState()` mogą być asynchroniczne.

Oznacza to, że bezpośrednio po użyciu `this.setState()` nie masz jeszcze dostępu do nowego state.

Wynika to z optymalizacji, która znajduje się w React. Stara się ona zebrać w jednym przebiegu maksymalną ilość zmian stanu by na końcu dokonać jednej większej aktualizacji.

Oznacza to, że poniższy przykład nie zrobi tego, co mogłoby się wydawać:

```
this.setState( (prevState) => {
  return {
    counter: prevState.counter + 1
  }
});
console.log(`Kliknięcia: ${this.state.counter}`);
```

Kod ten nie wyświetli nowej, poprawnej liczby sekund.

Co zrobić w takiej sytuacji? Metoda `this.setState()` może przyjąć więcej niż jeden parametr:

```
this.setState( newState, callback )
```

Jako pierwszy argument przyjmuje standardowo nowy stan (czy funkcje która zwraca obiekt z nowym stanem), natomiast jako drugi funkcję do wykonania gdy stan zostanie faktycznie zmieniony.

Aby poprzednio przytoczony przykład zadziałał poprawnie, wystarczy lekko zmodyfikować nasz kod dodając callback – popatrz na kod poniżej.

Kod ten wyświetli nową, poprawną liczbę sekund.

```
this.setState((prevState) => {
  return {
    counter: prevState.counter + 1
  }
}, () => {
  console.log(`Kliknięcia: ${this.state.counter}`);
});
```

Na koniec bardzo ważna informacja.

Być może przeszedł Ci przez myśl niezbyt dobry pomysł bezpośredniej modyfikacji `state` wewnątrz swojego obiektu.

```
this.state.seconds = this.state.seconds + 1;
```

Może się to wydawać szybszą alternatywą, jednak tak nie jest - nie rób tak. React wykonuje w `this.setState()` szereg skomplikowanych operacji, które są potrzebne.

W każdej sytuacji poza inicjacją `state` zawsze używaj `this.setState()`.

Dotyczy to również modyfikacji tablic, które znajdują się w stanie.

```
this.state = {
  users: ["Beniamin", "Dariusz", "Anna"]
}
```

Aby dodać kolejną osobę do tablicy nie możemy tak po prostu skorzystać z metody `push()`. Nie wystarczy również przypisanie `this.state.users` do nowej zmiennej. To nadal jest modyfikacja bezpośrednia stanu, która jest niepoprawna:

```
const users = this.state.users;
users.push("Marek");

this.setState({
  users
});
```

W takiej sytuacji skorzystamy z metody rozproszenia tablicy users w this.state a następnie dodania nowego elementu i zapisania całości do aktualnego stanu. Oto poprawne przykłady.

Pierwsze rozwiązanie:

```
const users = [...this.state.users];
users.push("Marek");

this.setState({
  users
});
```

Drugie rozwiązanie:

```
this.setState((prevState) => ({
  users: [...prevState.users, "Marek"]
}));
```

W jednym z poprzednich przykładów można było zauważyć taki kawałek kodu:

```
this.state = {
  //...
  currentUser: this.props.user
};
```

To połączenie to zainicjowanie lokalnego stanu za pomocą danej przekazanej w atrybucie przez `props`.

Takie połączenie jest poprawne.

Sprawia ono, że komponent wywołujący może ustalić jakąś wartość początkową dla jakiejś wartości stanu, która potem może ulec zmianie.

Żaden komponent nie może i nie powinien interesować się stanem innego komponentu.

Nazywamy state – stanem **lokalnym** właśnie dlatego, że należy on tylko do danego komponentu. Nie mają do niego dostępu ani rodzic, ani dziecko, ani rodzeństwo.

W jaki sposób komponenty mogą się więc "porozumiewać" między sobą?

Zasada ogólna mówi, że **dane idą z góry w dół – czyli z rodzica na dziecko**.

Wyobraź sobie komponenty jako wodospad. Kolejne komponenty mogą być dodatkowymi źródłami, jednak zawsze woda (dane) leci z góry na dół.

Za chwilę dowiesz się, w jaki sposób przekazywać i odbierać dane.

Dane przekazywane są po prostu jako atrybuty JSX. Są więc odbierane przez `props` przez komponent niżej.

Dane możemy przekazać m.in. z własnych `props`.

```
class User extends Component{
  render(){
    const {user} = this.props;
    return (
      <div className="user">
        <img src={user.avatar} />
        <strong>{user.name}</strong>
      </div>
    )
  }
}
```

```
class FullPost extends Component {
  render() {
    const {post} = this.props;
    return (
      <div>
        <User user={post.author} />
        <div className="post">
          <h1>{post.title}</h1>
          <p>{post.body}</p>
        </div>
      </div>
    )
  }
}
```

Przeanalizujmy ten przykład.

Komponent `FullPost` przyjmuje jako `props` `post`.

`FullPost` renderuje komponent `User` przekazując mu atrybut `user` z własnych `props`.

`User` może korzystać z przekazanych danych przez `props`.

Komponent `App` renderuje `FullPost`

```
<FullPost post={post} />
```

Komponent `FullPost`

```
const {post} = this.props;
```

```
<User user={post.author} />
```

Komponent `User`

```
const {user} = this.props;
```

```
<strong>{user.name}</strong>
```

Dane przekazywane są po prostu jako atrybuty JSX. Są więc odbierane przez `props` przez komponent niżej.

Dane możemy przekazać również z własnego `state`.

Spójrz na przykład na następnej stronie. Komponent `App` ma własny stan. Renderuje on komponent `Clock` przekazując mu atrybut `time` z własnego `state`.

```
class Clock extends Component {
  render(){
    return <strong>{this.props.time.toLocaleTimeString()}</strong>;
  }
}

class App extends Component {
  state = {
    time: new Date()
  }
  render() {
    return (
      <>
        <h1>Czas na świecie</h1>
        <Clock time={this.state.time} />
      </>
    );
  }
}
```

Przeanalizujmy, co dokładnie się dzieje.

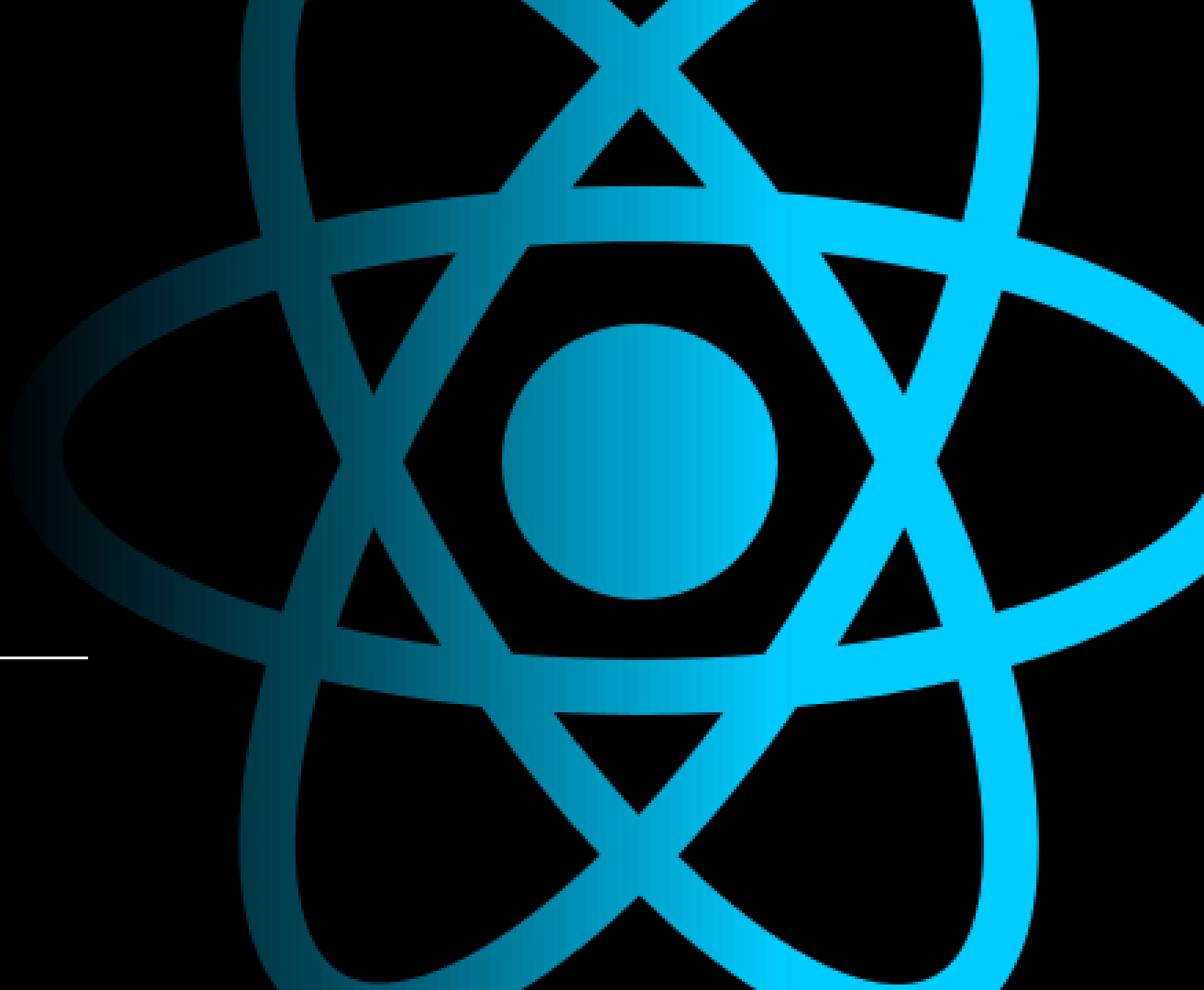
Komponent `App` zostaje stworzony. Jest inicjowane `state`, m.in. z właściwością `time`.

`App` renderuje komponent `Clock` przekazując mu props `time` z własnego `state`.

`Clock` może korzystać z przekazanych danych przez `props`.

```
<App />
//...
state = { time: new Date() }
//...
<Clock time={this.state.time} />
//...
<strong>{this.props.time.toLocaleTimeString()}</strong>
```

Cykl życia komponentu



A gdybyśmy chcieli napisać kod, który liczy, ile sekund jesteśmy na stronie? Moglibyśmy co sekundę wywoływać `ReactDOM.render()`, ale chcemy uniknąć zbędnego renderowania komponentu.

Powinien on zawierać całą własną logikę i sam zarządzać swoimi zmianami – dlatego skorzystamy ze `state`.

Komponent `TimeOnPage` powinien być w stanie zarządzać sam sobą. Chcemy zamknąć w nim całą logikę odliczania sekund (enkapsulacja).

Aby dowiedzieć się, jak to zrobić poprawnie, najpierw przejdziemy do cyklu życia komponentów.

Potem wrócimy do ostatniego zagadnienia stanu lokalnego – jego aktualizacji.

Jak zmieniać state?

```
class TimeOnPage extends Component {
  state = {
    seconds: 0
  }

  render(){
    return <h1>Minęło {this.state.seconds} sekund</h1>;
  }
}

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

Cykl życia komponentu można określić jako etapy, w jakich może on się znaleźć. Do takiego etapu w cyklu życia zaliczamy m.in. jego stworzenie, zniszczenie, czy też aktualizację.

React rozróżnia trzy ważne fazy, jeżeli chodzi o życie komponentu:

1. **Zamontowanie** (z ang. mount). Jest to faza **pierwszego realnego pojawienia się komponentu w prawdziwym drzewie DOM**.
2. **Aktualizacja** (z ang. update). Jest to faza **aktualizacji komponentu, np. o nowy zestaw** `props` **od komponentu nadrzędnego**.
3. **Odmontowanie** (z ang. unmount). Jest to zdarzenie **usunięcia (bez chęci aktualizacji) komponentu z prawdziwego drzewa DOM**.

W React możesz zareagować na podane dalej zdarzenia cyklu życia komponentu. Aby je obsłużyć wystarczy w klasie komponentu stworzyć odpowiednią metodę.

`componentDidMount()`

Jest to najczęściej używana metoda w cyklu życia komponentu

Zostanie ona uruchomiona tuż **po** stworzeniu odpowiednich elementów w realnym DOM-ie (po metodzie `render()`).

Metoda ta uruchomi się **tylko raz** podczas życia komponentu.

Jest to idealne miejsce na:

- uruchomienie interwałów,
- timerów,
- rozpoczęcie pobierania danych z serwera.

```
shouldComponentUpdate(nextProps, nextState)
```

Ta metoda zostanie uruchomiona, jeżeli zmieni się wewnętrzny `state` lub `props`. Przekazane argumenty to kolejno: nowe props i nowy state. Domyślnie przy zmianie `props` z zewnątrz lub `state` z wewnątrz automatycznie aktualizowany jest komponent (więcej o tym dowiesz się za chwilę).

Używając tej metody i zwracając z niej `false` możesz to zablokować (bo np. wiesz, że zmiana którejś opcji nie wpływa na zmianę wyglądu).

Zdarzenia aktualizacji

```
componentDidUpdate(prevProps,
    prevState, snapshot)
```

Metoda uruchamia się zaraz po aktualizacji komponentu. **Nie jest uruchamiana przy zamontowaniu.**

Tą aktualizacją może być nowy zestaw propsów lub bezpośrednia modyfikacja state danego komponentu. Jest przydatna np. w sytuacji, gdy przez `props` otrzymujemy ID jakiegoś produktu, który mamy wyświetlić. Jego zmiana będzie skutkowała pobraniem z API danych na jego temat. To będzie najlepsze miejsce na taką właśnie akcję.

Zdarzenia odmontowania

```
componentWillUnmount()
```

Metoda, która zostaje uruchomiona tuż przed całkowitym usunięciem z DOM. Idealne miejsce aby wyczyścić wszelkie zasoby (np. wyłączyć interwały i timery).

Pamiętaj o czyszczeniu zasobów (więcej o tym za chwilę).

Wracamy do naszego przykładu. Wiemy już, że możemy wykorzystać metodę: `componentDidMount()`, aby zainicjować np. nasz interwał. Tak też zrobimy. Zapisujemy do właściwości obiektu identyfikator tego interwału aby móc go później zatrzymać.

```
componentDidMount() {
  this.intervalId = setInterval(() => {
    //Tu będziemy aktualizować state
  }, 1000);
}
```

```
class TimeOnPage extends Component {
  state = {
    seconds: 0
  }

  componentDidMount() {
    this.intervalId = setInterval(() => {
      //Tu będziemy aktualizować state
    }, 1000);
  }

  render() {
    return <h1>Minęło {this.state.seconds} sekund.</h1>;
  }
}
```

Zawsze należy po sobie sprzątać, więc chcemy pozostawić przeglądarkę czystą. Zasada jest prosta – po dodaniu i usunięciu komponentu nie powinny pozostawać żadne "śmieci" (np. w postaci uruchomionego interwału). Dodajemy to do kodu.

```
componentWillUnmount() {
  clearInterval(this.intervalId);
}
```

```
class TimeOnPage extends Component {
  state = {
    seconds: 0
  }

  componentDidMount() {
    this.intervalId = setInterval(() => {
      //Tu będziemy aktualizować state
    }, 1000);
  }

  componentWillUnmount() {
    clearInterval(this.intervalId);
  }

  render() {
    return <h1>Minęło {this.state.seconds} sekund.</h1>;
  }
}
```

Dodamy kod, który co sekundę wykonuje `this.setState()`, aktualizując stan wewnętrzny komponentu. Pobiera wcześniejszą wartość `seconds` i zwiększa ją o 1.

Kod na kolejnym slajdzie realizuje swoje zadanie zgodnie ze wszystkimi dobrymi praktykami, które dotąd poznaliśmy:

- → Zamyka całą swoją funkcjonalność w jednej, hermetycznej klasie komponentu;
- → Sam uruchamia potrzebny interwał i sprząta po sobie po usunięciu elementu;
- → Poprawnie korzysta ze `state`.

```
class TimeOnPage extends Component {
  state = {
    seconds: 0
  }

  componentDidMount(){
    this.intervalId = setInterval(() => {
      this.setState( (prevState) => {
        return {
          seconds: prevState.seconds + 1
        }
      });
    }, 1000);
  }

  componentWillUnmount(){
    clearInterval(this.intervalId);
  }

  render(){
    return <h1>Minęło {this.state.seconds} sekund.</h1>;
  }
}
```

Przeanalizujmy działanie przykładowego kodu krok po kroku:

1. Tuż po wstawieniu naszego komponentu na stronę ustalamy, że początkowy stan przechowuje informację o sekundach z wartością 0.

```
state = {
  seconds: 0
}
```

2. Następnie wstawiamy nasz element, wywoływana jest metoda `render()`, w której zwracamy widok komponentu.

```
render(){
  return <h1>Minęło {this.state.seconds} sekund.</h1>;
}
```

3. Tuż po wyświetleniu komponentu wywoływana jest metoda `componentDidMount()`, w której tworzymy nowy interwał i przechowujemy jego identyfikator w obiekcie.

```
componentDidMount(){
  this.intervalId = setInterval(() => { // ...
```

4. Po sekundzie uruchamia się pierwszy raz nasza funkcja i aktualizuje wartość klucza `seconds` dodając 1.

```
this.setState( (prevState) => {
  return {
    seconds: prevState.seconds + 1
  }
});
```

5. Wywołanie zmiany stanu wymusza aktualizację, a więc znów uruchamia metodę `render()`. Ponownie renderujemy komponent, tym razem z liczbą 1.

```
render(){
  return <h1>Minęło {this.state.seconds} sekund.</h1>;
}
```

6. Ponieważ uruchomiliśmy interwał, kroki 4 i 5 są powtarzane co sekundę.
7. Kiedy nasz komponent przestanie być potrzebny, zostanie uruchomiona metoda cyklu życia `componentWillUnmount()`. W niej czyścimy zajęte zasoby, czyli usuwamy interwał.

```
componentWillUnmount() {
  clearInterval(this.intervalId);
}
```

Komponent przestaje być potrzebny w dwóch sytuacjach:

1. Kiedy strona/zakładka jest zamykana;
2. W momencie, gdy przestajemy wyświetlać komponent, np. poprzez komendę `if`.

Spójrz na przykład na dole. W momencie, kiedy użytkownik się wyloguje, komponent `UserInfo` przestanie być potrzebny i zostanie usunięty.

```
let userInfo;
if (userLoggedIn) {
  userInfo = <UserInfo user={user} />;
} else {
  userInfo = <a href="/login">Zaloguj</a>;
}
return <div>{ userInfo }</div>
```