infoShare ACADEMY

React

useState i obsługa zdarzeń

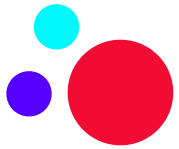
infoShare Academy

HELLO

Jakub Wojtach

Senior **full stack** developer

infoShare ACADEMY



Rozgrzewka i agenda

Zacznijmy ten dzień z przytupem!

Rozgrzewka

Zmysł smaku

Zmysł słuchu

Rozgrzewka

Iron Man

Thor

Rozgrzewka

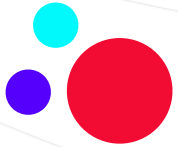
Jezus z Rio

Jezus ze Świebodzina

Rozgrzewka

Ksiądz

Pastor

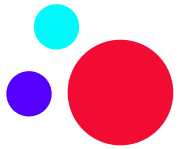


Agenda

- Podstawy teorii
- Definiowanie lokalnego stanu
- Obsługi powszechnie wykorzystywanych zdarzeń
- Myślenie Reactowe i naiwne strategie przekazywania danych w aplikacji

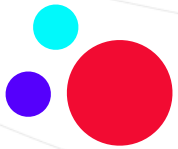
Współpraca

- Zadajemy pytania w dowolnym momencie - kanał **merytoryka**
- Krótkie przerwy (**5 min**) co godzinę
- Długa przerwa (**20 min**) po ostatnim bloku



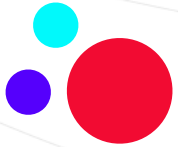
Podstawy teorii

infoShareAcademy.com



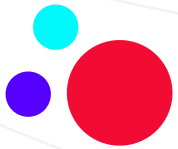
Samodzielne zdobywanie wiedzy

- https://www.simplilearn.com/tutorials/reactjs-tutorial/reactjs-state
- https://blog.logrocket.com/a-guide-to-usestate-in-react-ecb9952e406c/



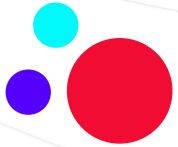
Wprowadzenie stan

- Aby zrozumieć, czym jest "stan" w programowaniu, zrozummy, co właściwie oznacza słowo "stan" ogólnie.
- Jeśli piję kawę, to moim aktualnym stanem jest to... że piję kawę. Możemy to również zawęzić: Być może przenoszę kubek z kawą do ust, więc moim stanem jest to, że trzymam kubek z kawą i podnoszę go do ust. Moim stanem jest również to, że moje usta są otwarte. Zatem "stan" to niekoniecznie tylko jedna informacja.
- Jako człowiek stale znajdujemy się w wielu różnych stanach - w zależności od tego, jak na to spojrzymy. Jeśli podzielimy to na części (jak powyżej), możemy przypisać stany do różnych części ciała (np. "usta są otwarte") lub spojrzeć na ogólny stan w jakim się znajdujemy ("picie kawy").



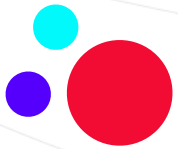
Wprowadzenie stan

- W programowaniu właściwie ta analogia jest adekwatna 1:1! W aplikacji internetowej stan ogólny może być taki, że wyświetlana jest okno "modal", które prosi odwiedzającego o podanie danych (np. okno logowania). Oczywiście, możemy to podzielić na mniejsze części:
- Stan okna "modal" jest otwarty.
- Stan formularza w modalu jest niewypełniony (wszystkie pola są puste).
- Mówiąc bardziej formalnie czym jest stan w kontekście aplikacji webowej jest to powiemy, że:
  - *state is a condition of an object stored as data.*
  - ***stan to obraz danego obiektu przechowywanego w postaci danych.***



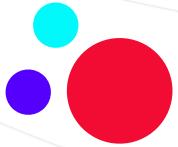
Wprowadzenie stan aplikacji *state*

- Komponent potrzebuje własnego stanu (state) wówczas gdy musi sprawdzać, czy związane z nim dane zmieniają się w czasie.
- Komponent Checkbox poprzez zmienną isChecked mógłby śledzić, czy jest zaznaczony, a komponent AccountBalance mógłby przechowywać pobieraną wartość salda rachunku.
- Dla każdego fragmentu danych zmieniających się w czasie powinien istnieć tylko jeden komponent, który taki stan posiada na wyłączność.



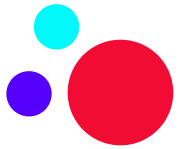
Wprowadzenie events

- Obsługa zdarzeń w elementach reactowych ma kilka różnic składniowych:
  - Nazwy procedur obsługi zdarzeń używają konwencji camelCase, a nie są pisane małymi literami.
  - W składni JSX procedury obsługi zdarzeń przekazuje się jako funkcje, a nie jako ciągi znaków.



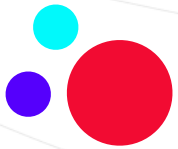
Wprowadzenie hooki

- Hooki to funkcje, które pozwalają "zahaczyć się" w mechanizmy stanu i cyklu życia Reacta z wewnątrz komponentów funkcyjnych.
- React dostarcza kilku wbudowanych hooków, ale można też tworzyć własne hooki.



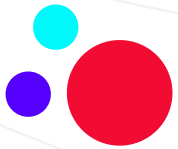
Lokalny stan komponentu

infoShareAcademy.com



Poznajemy pierwszy hook useState

- Do obsługi stanu w komponentach funkcyjnych służy hook useState.
- Mechanizm ten został wprowadzony wraz z nowym reactem, wcześniej komponenty funkcyjne nie miały możliwości korzystania ze stanu, przez co pisaliśmy je jako klasy
- Najlepiej wyjaśnić jego działanie na gotowym przykładzie.



Poznajemy pierwszy hook useState

- Poniższy komponent służy do ustawienia i wyświetlenia etykiety temperatury:

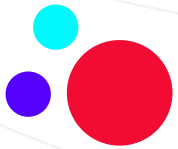
```
import { useState } from 'react'

function HookExample() {
  const [temp, setTemp] = useState('⛄ZIMNO⛄')

  return (
    <div>
      <p>Klimatyzacja ustawiona na: {temp}</p>
      <button onClick={() => setTemp('🌞CIEPŁO🌞')}>CIEPŁO</button>
      <button onClick={() => setTemp('⛄ZIMNO⛄')}>ZIMNO</button>
    </div>
  )
}
```

- Przeanalizujmy powyższy kod krok po kroku, aby zrozumieć jak on działa.

infoShare ACADEMY



Poznajemy pierwszy hook useState

- Definicja komponentu nie jest Wam już obca:

```
// ...

function HookExample() {
  // ...

  return (
    <div>
      <p>Klimatyzacja ustawiona na: {temp}</p>
      <button {/* ... */}>CIEPŁO</button>
      <button {/* ... */}>ZIMNO</button>
    </div>
  );
}
```

- To po prostu zwykły komponent, zwracający tekst ze zmienną temp oraz dwa przyciski.

Definiujemy stan komponentu

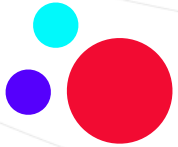
- Stan deklarujemy, używając hooka setState:

```
import { useState } from 'react'

//function HookExample() {
const [temp, setTemp] = useState('⛄ZIMNO⛄')

// }
```

- useState zwraca tablicę dwuelementowa, która zawiera aktualną wartość stanu oraz funkcję, która będzie tą wartość aktualizować.
- Stan nazwaliśmy temp, a funkcję ustawiającą setTemp (poprzedzamy słowem set ponieważ ta funkcja "ustawia" temperaturę - stały pattern, bardzo często stosowany).
- **useState** przyjmuje jeden argument, będący początkową wartością stanu, czyli naszej temp (w tym przypadku domyślnie "⛄ZIMNO⛄").
- Zapis z nawiasem kwadratowym może wydawać się dziwny, ale jest to zwykła destrukturyzacja tablic, którą już znacie!



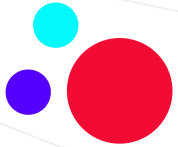
Stan

- Stan odczytujemy bezpośrednio ze zmiennej temp:

```
<p>Klimatyzacja ustawiona na: {temp}</p>
```

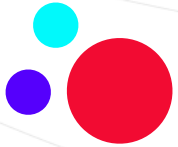
- Stan aktualizujemy używając metody setTemp:

```
<button onClick={() => setTemp("🌞CIEPŁO🌞")}>CIEPŁO</button>
<button onClick={() => setTemp("⛄ZIMNO⛄")}>ZIMNO</button>
```



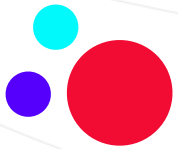
Stan – podsumowanie

- Importujemy hooka useState
- Wewnątrz komponentu deklarujemy nowy stan const [temp, setTemp] = useState("⛄ZIMNO⛄"), przypisując mu początkową wartość na "⛄ZIMNO⛄".
- Stan odczytujemy bezpośrednio ze zmiennej temp.
- Kiedy użytkownik kliknie przycisk, funkcja setTemp zaktualizuje wartość stanu na taką, jaka została przekazana do setTemp jako argument wywołania, czyli albo "🌞CIEPŁO🌞", albo "⛄ZIMNO⛄".



Stan - pare faktów

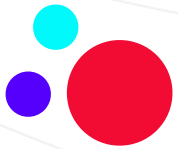
- React może zgrupować kilka wywołań metody setState() w jedną paczkę w celu zwiększenia wydajności aplikacji.
- Z racji tego, że zmienne this.props i this.state mogą być aktualizowane asynchronicznie, nie powinno się polegać na ich wartościach przy obliczaniu nowego stanu.
- Dochodzi do tego, że często w aplikacji możemy mieć sytuację, że nie mamy dostępu do nowej wartości komponentu i następuje desynchronizacja danych
- Aby temu zaradzić, wystarczy użyć alternatywnej wersji metody setState(), która jako argument przyjmuje funkcję zamiast obiektu. Funkcja ta otrzyma dwa argumenty: aktualny stan oraz aktualne atrybuty komponentu.



Stan - pare faktów

```
// Źle
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

infoShare ACADEMY

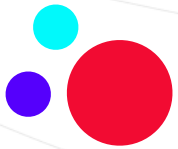


Stan - pare faktów

```
// Źle
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

```
// Dobrze
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

infoShare ACADEMY



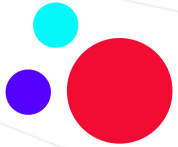
Stan - pare faktów

```
// Źle
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

```
// Dobrze
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

```
// Dobrze
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

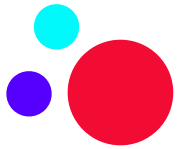
infoShare ACADEMY



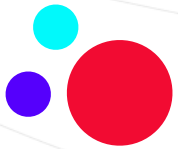
Stan - pare faktów

- Ani komponenty-rodzice, ani ich dzieci nie wiedzą, czy jakiś komponent posiada stan, czy też nie. Nie powinny się również przejmować tym, czy jest on funkcyjny, czy klasowy.
- Właśnie z tego powodu stan jest nazywany lokalnym lub enkapsulowanym. Nie mają do niego dostępu żadne komponenty poza tym, który go posiada i modyfikuje.
- Komponent może zdecydować się na przekazanie swojego stanu w dół struktury poprzez atrybuty jego komponentów potomnych
- Taki przepływ danych nazywany jest powszechnie jednokierunkowym (ang. unidirectional) lub “z góry na dół” (ang. top-down). Stan jest zawsze własnością konkretnego komponentu i wszelkie dane lub części UI, powstałe w oparciu o niego, mogą wpłynąć jedynie na komponenty znajdujące się “poniżej” w drzewie.
- Wyobraźcie sobie, że drzewo komponentów to wodospad atrybutów, a stan każdego z komponentów to dodatkowe źródło wody, które go zasila, jednocześnie spadając w dół wraz z resztą wody.

Zadanie



Obsługa zdarzeń



- Obsługa zdarzeń w React jest bardzo podobna do tej, którą znacie z drzewa DOM. Są jednak pewne różnice w składni.
- Zdarzenia reactowe pisane są **camelCasem**, a nie małymi literami, a procedury obsługi zdarzeń przekazuje się jako funkcje
- W HTML-u:

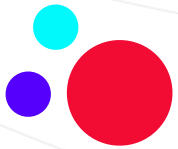
```
<!--
mamy tu małymi literami: onclick
ciąg znaków: "pickTreasure()"
-->

<button onclick="pickTreasure()">Podnieś skarb</button>
```

- Natomiast w React:

```
// mamy tu
// - camelCasem: onClick
// - przekazanie funkcji: {pickTreasure}

<button onClick={pickTreasure}>Podnieś skarb</button>
```



- Zdarzenia myszy

onClick onContextMenu **onDoubleClick** onDrag onDragEnd onDragEnter onDragExit onDragLeave onDragOver onDragStart onDrop onMouseDown **onMouseEnter** **onMouseLeave** onMouseMove onMouseOut onMouseOver onMouseUp

- Zdarzenia klawiatury

onKeyDown, onKeyPress oraz onKeyUp

- Zdarzenia formularza

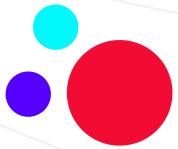
onChange, onInput, onInvalid, onReset, onSubmit

- Zdarzenia focusu:

ofFocus, onBlur

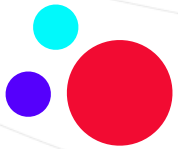
- Zdarzenia schowka

onCopy, onCut, onPaste



Synthetic Event

- Napisane przez ciebie event handlery będą otrzymywać jako argument instancję SyntheticEvent - klasy opakowującej natywne zdarzenie, niezależnej od przeglądarki.
- Posiada ona taki sam interfejs jak natywne zdarzenia, wliczając w to metody stopPropagation() oraz preventDefault(), gwarantuje jednak identyczne działanie na wszystkich przeglądarkach.
- Jeśli w którymś momencie zechcesz skorzystać z opakowanego, natywnego zdarzenia, możesz odwołać się do niego poprzez właściwość nativeEvent. Syntetyczne zdarzenia różnią się od natywnych zdarzeń przeglądarki i można ich stosować wymiennie. Na przykład, w zdarzeniu onMouseLeave wartość event.nativeEvent będzie wskazywać na zdarzenie mouseout



- Obsługa zdarzeń w React jest bardzo podobna do tej, którą znacie z drzewa DOM. Są jednak pewne różnice w składni.
- Zdarzenia reactowe pisane są **camelCasem**, a nie małymi literami, a procedury obsługi zdarzeń przekazuje się jako funkcje
- W HTML-u:

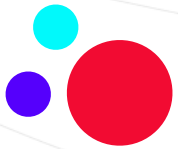
```
<!--
mamy tu małymi literami: onclick
ciąg znaków: "pickTreasure()"
-->

<button onclick="pickTreasure()">Podnieś skarb</button>
```

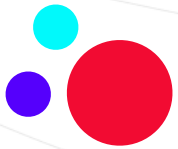
- Natomiast w React:

```
// mamy tu
// - camelCasem: onClick
// - przekazanie funkcji: {pickTreasure}

<button onClick={pickTreasure}>Podnieś skarb</button>
```



- Przekazywanie funkcji do obsługi zdarzeń może być na początku nieco zawiłe, gdyż jest możliwe na wiele sposobów. Poniżej kilka przykładów:
- Powiedzmy, że piszemy grę i chcemy "na szybko" sprawdzić, czy klikanie w diva z planszą jest w ogóle możliwe i czy plansza "reaguje" na klik. Zazwyczaj robimy console.log i jeżeli coś leci w konsoli, to znaczy, że plansza reaguje. Możemy to zrobić na kilka sposobów, lecz tylko niektóre będą poprawne:



```
// 👍 kod, który na klik zadziała właściwie (wyświetli tzw. SyntheticBaseEvent)
<div onClick={console.log}>GameBoard</div>

// 👍 ten kod też zadziała właściwie na klik (wyświetli obiekt SyntheticBaseEvent)
<div onClick={(e) => console.log(e)}>GameBoard</div>

// 👍 ten też na klik będzie OK, wyświetli słowo 'działa'
<div onClick={() => console.log("działa")}>GameBoard</div>

// 👎 ten kod zadziała niewłaściwie (nic nie wyświetli, uruchomi się tylko raz, nie
// będzie reagował na kolejne wciśnięcia przycisku
<div onClick={console.log()}>GameBoard</div>

// 👎 ten też zadziała niewłaściwie (wyświetli 'działa", ale nie będzie reagował na kliknięcia)
<div onClick={console.log("działa")}>GameBoard</div>
```

- Pozostańmy w wątku gry. Chcemy na kliknięcie uruchomić lasery. Tworzymy w komponencie funkcję, która odpali lasery.

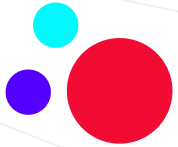
```
// niech 🚀 oznacza strzał z lasera
const fireLaser = () => {
  console.log('🚀')
}
```

- A następnie wywołujemy tę funkcję każdorazowo na kliknięcie:

```
// 👍 to zadziała poprawnie, na każde kliknięcie odpali się laser
// do onClick przekazujemy zmienną, do której przypisaliśmy funkcję strzałkową
<button onClick={fireLaser}>Fire 🚀</button>

// 👎 to nie zadziała poprawnie (bo tylko raz)
// przekazujemy rezultat wywołania funkcji
<button onClick={fireLaser()}>Fire 🚀</button>

// 👍 to też zadziała poprawnie, na każde kliknięcie odpali się laser
// przekazujemy funkcję (tak naprawdę po prostu jest to skopiowana funkcja fireLaser)
<button
  onClick={() => {
    console.log("🚀")
  }}
>
  Fire 🚀
</button>
```



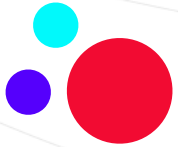
- Może się okazać, że mamy kilka rodzajów laserów:

```
const lasers = ['🚀', '☢', '💥', '🔱', '🌠']
```

- Więc dla każdego z nich zrobimy osobny przycisk, mapując tablicę laserów (pomińmy użycie propsa key dla uproszczenia):

```
//  👍👎 Zadziała, ale nie do końca tak, jak byśmy chcieli, gdyż
// wprawdzie wyrenderuje 5 przycisków, ale każdy będzie odpalał 🚀...
<div>
  {lasers.map(laser => (
    // ...dlatego że do onClick przekazujemy zmienną do której
    // przypisana jest funkcja strzelająca tylko 🚀
    <button onClick={fireLaser}>{laser}</button>
  ))}
</div>
```

- Naprawmy to, aby każdy przycisk odpalał swój laser. Najpierw poprawmy nieco funkcję odpalającą lasery:

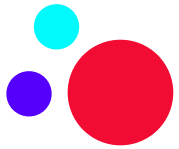


```
const betterFireLaser = laserType => {
  console.log(laserType)
}
```

- A następnie wywołajmy naszą lepszą funkcję, przekazując jako argument jej wywołania typ lasera odpowiedni dla każdego z przycisków:

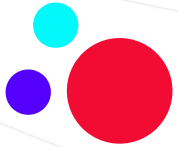
```
//  👍 Zadziała tak, jak chcemy, każdy przycisk strzela swoim laserem...
<div>
  {lasers.map(laser => (
    // ...dlatego że do funkcji strzelającej przekazujemy zmienną
    // `laser`, która jest inna dla każdego z przycisków
    <button onClick={() => betterFireLaser(laser)}>{laser}</button>
  ))}
</div>
```

Zadanie



Myślenie “Reactowe”

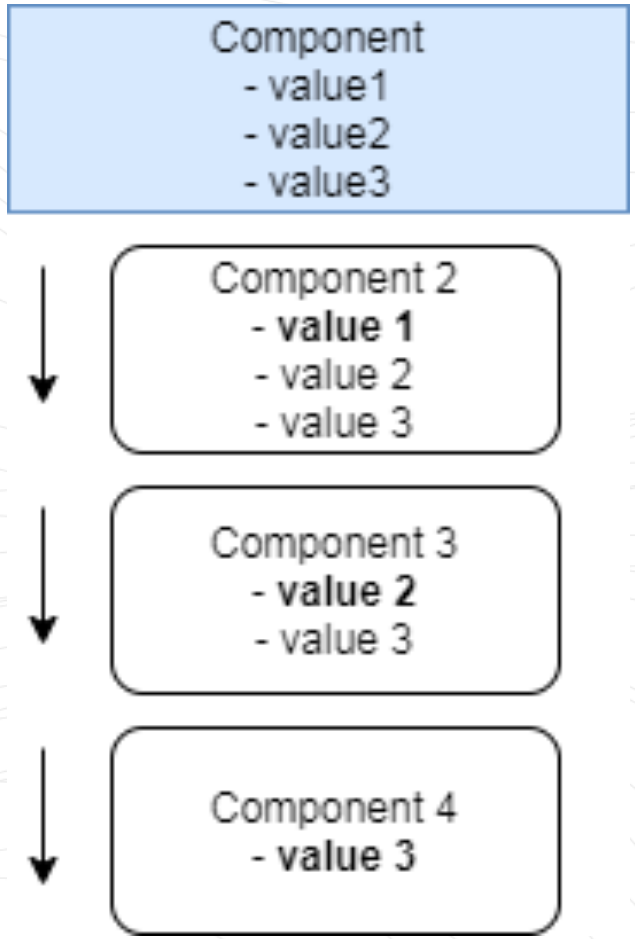
infoShareAcademy.com

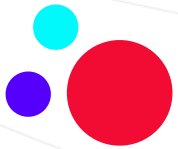


Thinking in React

- React może zmienić sposób, w jaki myślisz o projektach i tworzonych aplikacjach.
- Kiedy tworzysz interfejs użytkownika za pomocą Reacta, najpierw rozbijasz go na części zwane komponentami.
- Następnie opiszesz różne stany wizualne dla każdego z komponentów.
- Na koniec połączysz ze sobą swoje komponenty, aby dane przepływały przez nie.
- Wraz z tą dokumentacją przejdziemy sobie *skrótowo* przez kolejne kroki modelowego tworzenia aplikacji

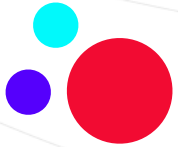
Props Drilling





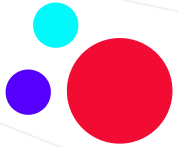
Props Drilling Omówienie Problemu

- Wartość pogrubiona oznacza, że komponent wykorzystuje daną wartość, zaś wartość niepogrubiona oznacza, że komponent daną wartość posiada, mimo że z niej nie korzysta.
- Component przekazuje wartości: value 1, value 2, value 3 do Component 2
- Component 2 przekazuje wartości: value 2, value 3 do Component 3, zaś korzysta tylko z wartości value 1.
- Component 3 przekazuje wartość: value 3 do Component 4 , zaś korzysta tylko z wartości value 2.
- Component 4 korzysta z wartości value 3.



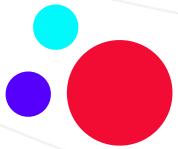
Props Drilling Omówienie Problemu

- Jak łatwo zauważyć, Component 3, mimo że używa tylko jednej wartości i tak musi otrzymać wartość value 3, tylko po to, aby przekazać ją niżej do Component 4 itd.
- Takie przekazywanie wartości w dół drzewa komponentów może być mało intuicyjne.
- Jeżeli będziemy przekazywali większa ilość wartości, łatwo będzie się w tym pogubić.
- Dodatkowo wysyłamy wartość z jednego komponentu tylko po to, aby przekazać go niżej, w ogóle go nie używając.



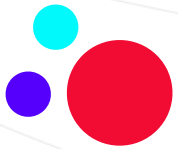
Props Drilling Omówienie Problemu

- Wyobraźmy sobie, że chcemy przekazać nową wartość, która pochodzi z Component do Component 4.
- Przy takim podejściu musimy znów przekazywać wartość niżej komponent po komponencie.
- Oczywiście możemy przekazać obiekt z wartościami, wtedy wystarczy dodać wartość do obiektu w Component i będzie on dostępny w Component 4.
- Tutaj jednak tracimy trochę kontrolę nad tym, co jest potrzebne danemu komponentowi do poprawnego działania, dodatkowo następuje zmiana „propsów”.



Props Drilling Omówienie Problemu

- Załóżmy inną sytuację.
- Sprawdzamy użycie komponentu Component 2 i widzimy, że przyjmuje on 3 wartości.
- Aby mieć pewność, z których wartości tak naprawdę korzysta komponent musimy po prostu prześledzić zawartość komponentu.
- Przy większej złożoności staje się to dosyć skomplikowane...

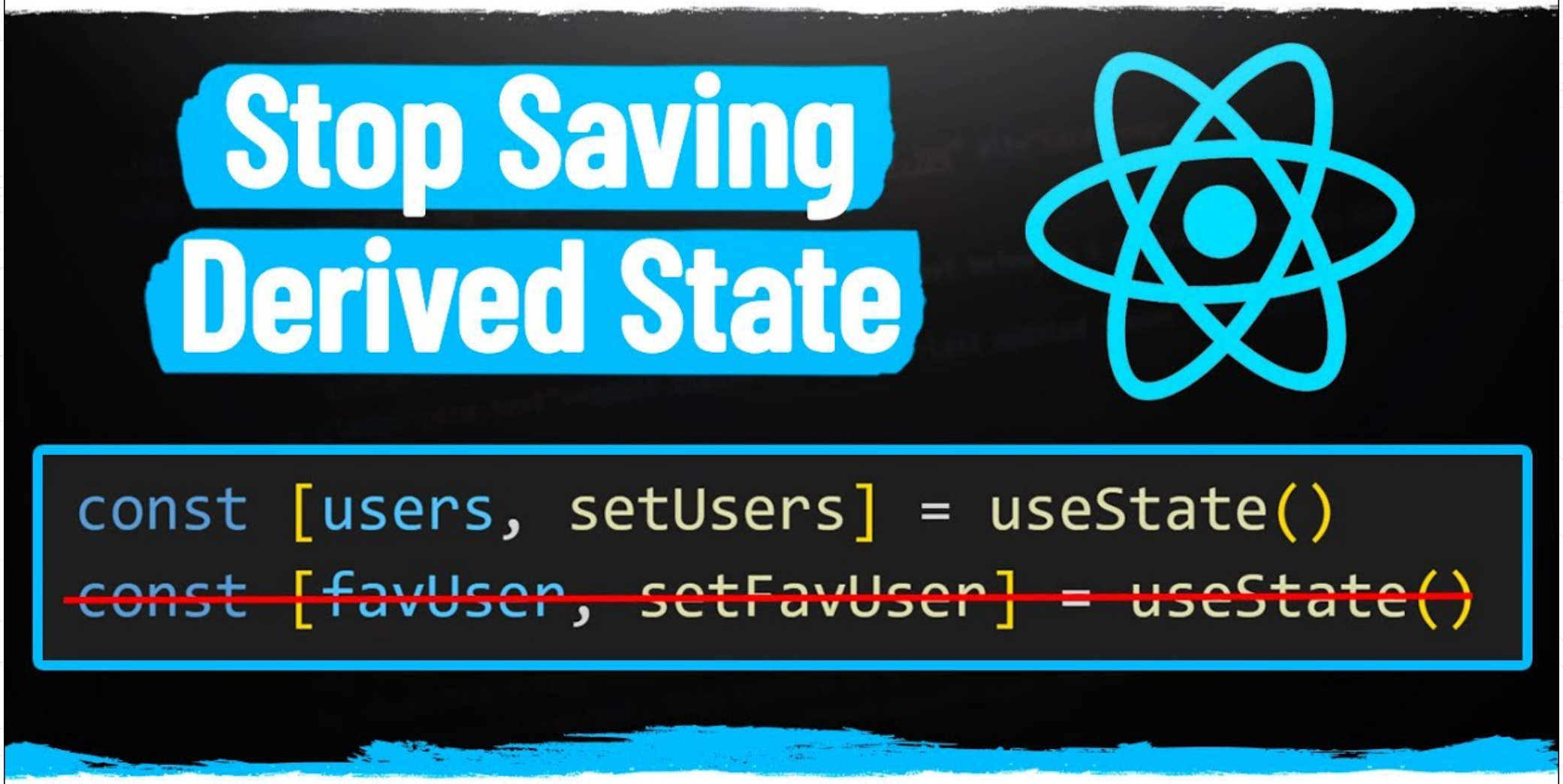


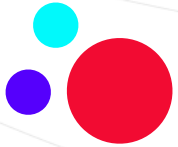
Props Drilling Rozwiązanie

- Aby rozwiązać ten problem możemy skorzystać np. Z Context API, o którym będzie na innych zajęciach
- Inne rozwiązanie to biblioteka zewnętrzna służąca do zarządzania stanem, jak np. **Redux**
- **WAŻNE**: Context API jest świetne, jeśli potrzebujemy zarządzać stanami z mniejszej części aplikacji.
- Jeśli mamy do czynienia z większymi porcjami danych i złożonymi aplikacjami warto zainwestować czas w kilka kontekstów, albo wspomniany już wcześniej state management.

Derived State

Derived State

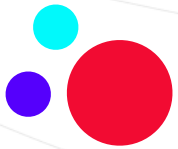




Derived State

- Chodzi o to, aby skłaniać się ku przechowywaniu jak najmniejszej ilości danych w twoim stanie.
- Sposobem na to jest unikanie przechowywania zmiennych stanu, które można wyprowadzić lub obliczyć w locie.
- Obliczanie zmiennych zamiast przechowywania ich w stanie ułatwia synchronizację danych w przypadku wystąpienia zmian.
- Mówiąc już zupełnie skrótowo - jeśli próbujemy odnieść się do elementu, który zależy od innego stanu - nie róbmy tego. Dane nie będą synchronizowane. Odnośmy się do części danych, która pozwoli nam na szybkie wyliczenie/znalezienie tego elementu i korzystanie z aktualnej części stanu, która jest nam w danej chwili potrzebna

Bonus task / homework



Linki dla chętnych

- https://www.robinwieruch.de/react-function-component/#react-function-component-state
- https://pl.reactjs.org/docs/hooks-state.html
- https://pl.reactjs.org/docs/events.html

Dziękuję za uwagę

Jakub Wojtach