

Tworzenie aplikacji z użyciem React

Mateusz Jabłoński



Kim jestem?

- ◆ programista od 2011 roku
- ◆ głównie: Javascript / Typescript / Java / dawniej: PHP
- ◆ szkoleniowiec / trener / mentor od 2016 roku
- ◆ prywatnie mąż i tata
- ◆ ostatnio również aktor w lokalnym teatrze i gracz papierowych RPG

Zaufaj naszemu doświadczeniu

18 lat

na rynku usług IT

29 560+

przeszkolonych osób

500+

klientów biznesowych

4 877+

zorganizowanych szkoleń i warsztatów

GWARANCJA JAKOŚCI USŁUG



98%

zadowolonych klientów*

* Średnia z ankiet poszkoleniowych przeprowadzanych wśród uczestników naszych szkoleń.

Edukacja na najwyższym poziomie

sages

Wiedza specjalistyczna dla branży IT

Oferujemy szeroki katalog szkoleń z technologii mainstreamowych i specjalistycznych, wschodzących i legacy. Zajęcia prowadzimy w trybie warsztatowym, a programy są oparte o praktyczne know-how. Specjalizujemy się w prowadzeniu dedykowanych szkoleń technologicznych, których agendę dostosowujemy do potrzeb naszych klientów i oczekiwania uczestników.

Wybitni eksperci

Od początku naszego istnienia przeszkołiliśmy dziesiątki tysięcy osób, co pomogło absolwentom podnieść konkurencyjność na rynku pracy i jakość projektów realizowanych na co dzień. Nasze szkolenia prowadzą najlepsi trenerzy, a nasze produkty są oparte na najnowocześniejszej technologii. Ich niezawodność i dopasowanie do potrzeb klientów są możliwe dzięki zespołowi składającemu się z wybitnych ekspertów i ekspertek, którzy/e są na pierwszej linii teorii i praktyki tworzenia i wdrażania innowacji technologicznych.

NASZE POZOSTAŁE MARKI EDUKACYJNE

STACJA.IT

kodo/amacz
by sages

Najlepsze standardy usług edukacyjnych

Stosujemy Standard Usługi Szkoleniowej Polskiej Izby Firm Szkoleniowych, a nasze usługi realizowane są na najwyższym poziomie, o czym świadczą stale powracający klienci oraz wdrożony certyfikat ISO 9001. Metodologia prowadzonych przez nas zajęć oparta jest na współczesnych narzędziach i dostosowana do potrzeb i oczekiwania klientów.

Ponadto jesteśmy firmą wpisaną do rejestru instytucji szkoleniowych w Wojewódzkim Urzędzie Pracy w Warszawie pod nr 2.14/00133/2019.

Zaufali nam najlepsi

Wśród naszych klientów są takie firmy jak Alior Bank, OLX Group, Bank Zachodni WBK, Orange Polska, Lufthansa i wiele innych.

STUDIA PODYPLOMOWE

Wspieramy organizację zaawansowanych kierunków studiów podyplomowych. Realizujemy zajęcia na kierunkach: Data Science, Big Data i Wizualna analityka danych, AI & Data Driven Business oraz User Experience Design – projektowanie doświadczeń cyfrowych.



Instytut Informatyki
Wydział Elektroniki i Technik Informacyjnych
Politechniki Warszawskiej



AKADEMIA
LEONA KOŽMIŃSKIEGO

Uwaga

Wszelkie materiały (treści tekstowe, wideo, ilustracje, zdjęcia itp.) wchodzące w skład szkoleń, kursów i webinarów organizowanych przez Sages są objęte prawem autorskim i podlegają ochronie na mocy Ustawy o prawie autorskim i prawach pokrewnych z dnia 4 lutego 1994 r. (tekst ujednolicony: Dz.U. 2006 nr 90 poz. 631). Kopiowanie, przetwarzanie, rozpowszechnianie tych materiałów w całości lub w części jest zabronione.



Ustalenia

- ▶ Cel i agenda
- ▶ Wzajemne oczekiwania
- ▶ Pytania i dyskusje
- ▶ Elastyczność
- ▶ Otwartość i uczciwość

Agenda, czyli co nas czeka?

1. Wprowadzenie
2. Podstawy React
3. React Hooks
4. React w praktyce
5. Zaawansowany React
6. Praca z wybranym mechanizmem zarządzania stanem aplikacji
7. Testowanie (opcja)

github.com/matwjablonski/react-apt-0226

React

	Angular	React	Vue	Svelte
Elastyczność	umiarkowana	bardzo wysoka	wysoka	wysoka
Krzywa uczenia się	stroma	średnia	łagodna	łagodna
Wielkość frameworka	ciężki (500+kB)	lekka biblioteka (40-50kB)	lekki (około 30kB)	najlżejszy (2-5kB)
Wsparcie społeczności	dość duże, bardziej korporacyjne (+Google)	największa (+Facebook)	bardzo duża i szybko rosnąca (+Laravel)	mniejsza, ale bardziej zaangażowana
Top Use Cases	Google, The Guardian	Facebook, Twitter, Instagram	9GAG, Gitlab	The New York Times, Spotify
Wydajność	DOM	Virtual DOM	Virtual DOM	DOM
Tło	Typescript, 2016, Google	Javascript lub Typescript, 2013, Facebook	Javascript lub Typescript, 2014, Evan You	Javascript lub Typescript, 2016, Rich Harris
Popularność	0,3% wszystkich stron internetowych	4,6% wszystkich stron internetowych	0,9% wszystkich stron internetowych	brak danych

Transpilacja

Transpilacja to proces przepisywania kodu do jego odpowiednika w innym języku (lub w tym samym, ale w innej wersji)

Przykłady narzędzi: Babel, TypeScript, SWC (Speedy Web Compiler)

Dlaczego transpilacja jest potrzebna?

- ▶ nie wszystkie przeglądarki obsługują najnowsze standardy ECMAScript
- ▶ nowe funkcje JS (np. `optional chaining`, `nullish coalescing`, `BigInt`) mogą nie działać w starszych przeglądarkach
- ▶ transpilacja pozwala zachować kompatybilność wsteczną

Kod ES6+:

Przykładowy kod:

```
const greet = (name) => `Hello, ${name}!`;
```

Transpilowany kod ES5:

Przykładowy kod:

```
var greet = function(name) {
  return 'Hello, ' + name + '!';
};
```

Polyfills vs Transpilacja

- ◆ **Transpilacja** zmienia składnię kodu, aby była zgodna ze starszymi wersjami JS
- ◆ **Polyfills** dodają brakujące funkcje do środowiska wykonawczego

Przykładowy kod:

```
// Polyfill dla Array.from
if (!Array.from) {
  Array.from = function(iterable) {
    return [].slice.call(iterable);
  };
}
```

Automatyzacja

- ◆ bundlery i narzędzia takie jak **Webpack**, **Vite**, **Parcel** integrują transpilację i polyfille automatycznie
- ◆ możemy ustawić docelowe środowisko (targets) w Babel, np.:

Przykładowy kod:

```
{  
  "presets": [  
    ["@babel/preset-env", {  
      "targets": "> 0.25%, not dead"  
    }]  
  ]  
}
```

Dobre praktyki

- ▶ transpiluj tylko to, co jest potrzebne (tzw. modern + legacy builds)
- ▶ stosuj polyfille tylko tam, gdzie funkcje są naprawdę używane
- ▶ regularnie testuj kod w różnych przeglądarkach



Babel

- ◆ darmowy transpiler javascriptowy
- ◆ pierwszy raz wydany w 2014 roku
- ◆ transpiluje kod z ES2015+ do starszych wersji, zgodnie ze wsparciem
- ◆ zezwala na transformacje niestandardowych technik jak: JSX do kodu JS
- ◆ zawiera zestaw polyfili aby zezwolić na używanie funkcji, które nie są dostępne w standardzie ES5
- ◆ konfiguracja jest trzymana w pliku `.babelrc`
- ◆ konfiguracja jest podzielona na dwie grupy: `presets` - reguły transformacji kodu oraz `plugins` - rozszerzenia dla silnika Babel'a

Babel w React

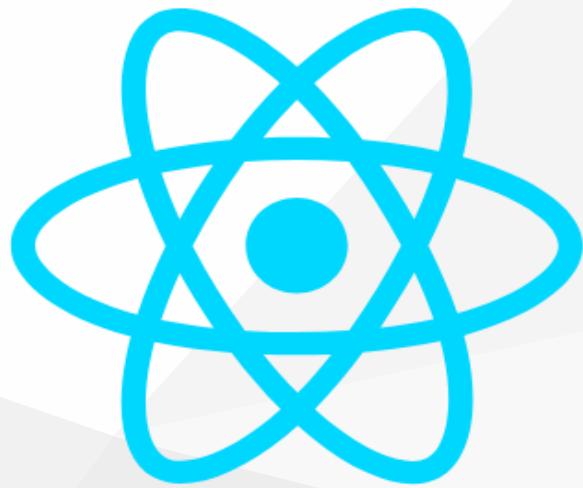
- ▶ odgrywa kluczową rolę w komplikacji kodu JSX, który nie jest natywnie obsługiwany przez przeglądarki
- ▶ jest jednym z podstawowych narzędzi wykorzystywanych w procesie development aplikacji React
- ▶ nowoczesne narzędzia, takie jak Vite, oferują szybsze alternatywy (np. SWC - Speedy Web Compiler), Babel nadal jest szeroko stosowany w projektach React, zwłaszcza tam, gdzie zależy na dużej kompatybilności z różnymi środowiskami i wersjami przeglądarek

Przykładowy kod:

```
{  
  "presets": ["@babel/preset-react"],  
  "plugins": ["react-hot-loader/babel"]  
}
```

SPA

- ▶ SPA - Single Page Applications
- ▶ SPA jest alternatywą dla MPA (Multi Page Applications)
- ▶ serwer zwraca prosty plik HTML, który zawiera tylko jednego diva i informacje o wymaganych stylach i skryptach
- ▶ odpowiedź z serwera jest bardzo szybka
- ▶ cała logika aplikacji i nawigacji jest przeniesiona na przeglądarkę
- ▶ problematyczne SEO
- ▶ problem z udostępnianiem treści w mediach społecznościowych (meta tags)
- ▶ aplikacje SPA mogą być budowane w sposób modułowy i komponentowy
- ▶ nie ma dużych wymagań od serwera (tylko pliki statyczne: html, css, js)
- ▶ koszt infrastruktury został przeniesiony na klienta końcowego



21

Czym jest React?

- ▶ biblioteka, której głównym zadaniem jest ograniczenie liczby mutacji na strukturze DOM
- ▶ został zaprojektowany i wprowadzony na rynek przez Facebook
- ▶ zdejmuje odpowiedzialność za renderowanie i aktualizację stanu drzewa dom z programisty
- ▶ React opiera się dość mocno na koncepcjach ze świata programowania funkcyjnego
- ▶ React służy do budowania dynamicznych i złożonych interfejsów użytkownika w sposób deklaratywny i modułowy
- ▶ struktura widoku i logika są trzymane w jednym miejscu (tylko w wykorzystaniem języka Javascript)
- ▶ React buduje widoki w architekturze komponentowej, gdzie komponent to element oprogramowania posiadający dobrze wyspecyfikowany interfejs oraz zachowanie

Przykładowy kod:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <!-- Load React. -->
  <!-- Note: when deploying, replace "development.js" with "production.min.js". -->
  <script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></script>
  <script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>

  <!-- Load our React component. -->
  <script src="like_button.js"></script>

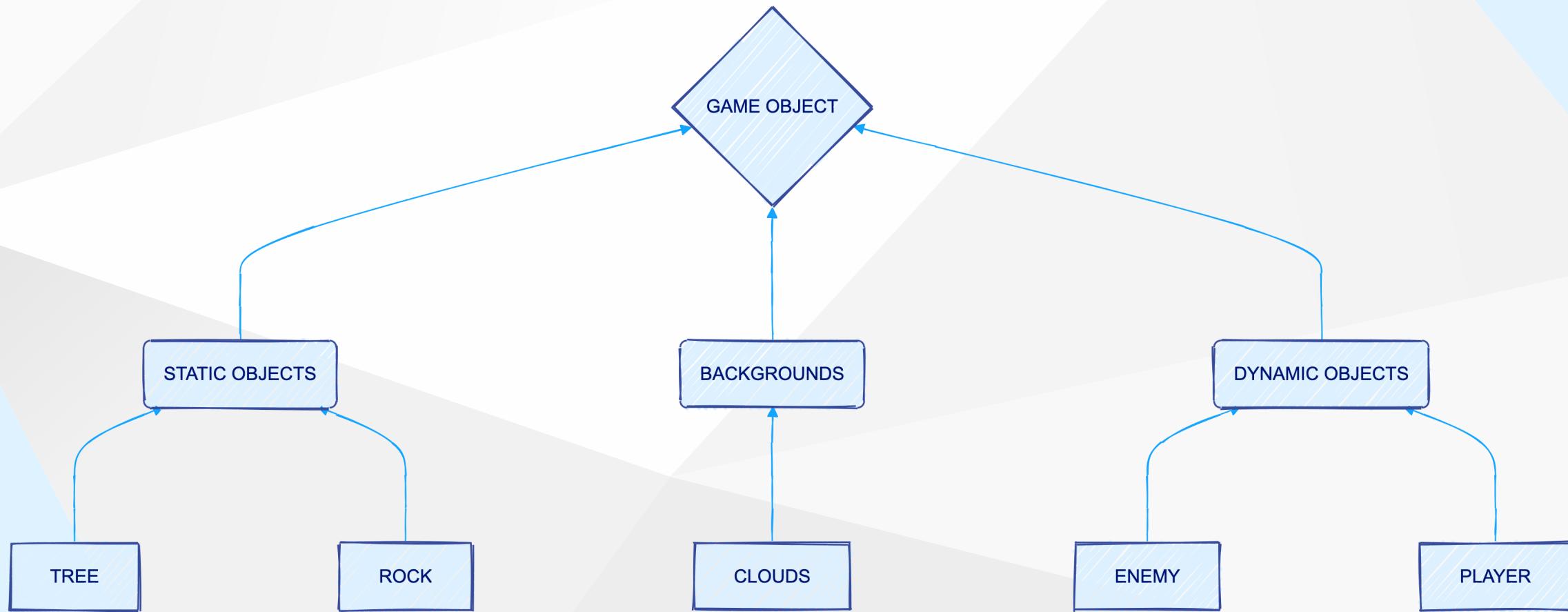
</body>
</body>
</html>
```

z czego składa się React?

- ◀ `react` - podstawowa biblioteka obsługująca model kompozycyjny React'a
- ◀ `react-dom` - biblioteka udostępniająca specyficzne dla DOM metody, która dzieli się na: `react-dom/client` - moduł, który zezwala na tworzenie i używanie aplikacji React po stronie klienta oraz `react-dom/server` - moduł, który zezwala na budowanie komponentów React w wersji serwerowej

Architektura komponentowa

- ▶ komponenty mogą być wykorzystane w wielu aplikacjach
- ▶ skraca czas budowy nowych aplikacji i obniża koszty projektu - poprzez wykorzystanie istniejących komponentów
- ▶ wykorzystując istniejące komponenty po pierwsze, nie musimy ich pisać od nowa, po drugie testować i dokumentować
- ▶ poszczególne komponenty mogą być tworzone równolegle
- ▶ poprzez dobre wyspecyfikowanie interfejsów poszczególnych komponentów można łatwo zlecać implementacje komponentów na zewnątrz
- ▶ wspiera tzw. modyfikowalność aplikacji - konkretna funkcjonalność skupia się w dedykowanych komponentach programowych, więc w przypadku konieczności wprowadzenia zmiany, z reguły proces ten dotyczy modyfikacji jednego lub co najwyżej kilku komponentów a nie całej aplikacji
- ▶ spójność (ang. coherence) komponentów wspiera modyfikowalność - im większa spójność tym łatwiej modyfikować aplikację



Loose Coupling

W idealnej architekturze komponenty powinny o sobie wiedzieć jak najmniej. Komponent **Player** nie powinien wiedzieć, jak zaimplementowany jest **Tree**. **Komunikacja odbywa się przez zdefiniowane kontrakty (interfejsy).**

Pozwala to na wymianę jednego komponentu na inny (np. zmianę **Tree** z wersji 2D na 3D) bez dotykania reszty logiki gry.



Kompozycja ponad dziedziczeniem

- ◀ współczesna architektura (np. ECS w grach [Entity-component-system](#) lub komponenty w Vue/Astro) promuje kompozycję
- ◀ zamiast tworzyć skomplikowane hierarchie klas, budujemy aplikację z małych, odizolowanych komponentów
- ◀ komponenty są łączone (komponowane) w większe jednostki funkcjonalne
- ◀ ułatwia to ponowne wykorzystanie kodu i testowanie
- ◀ zmiany w jednym komponencie mają minimalny wpływ na resztę systemu

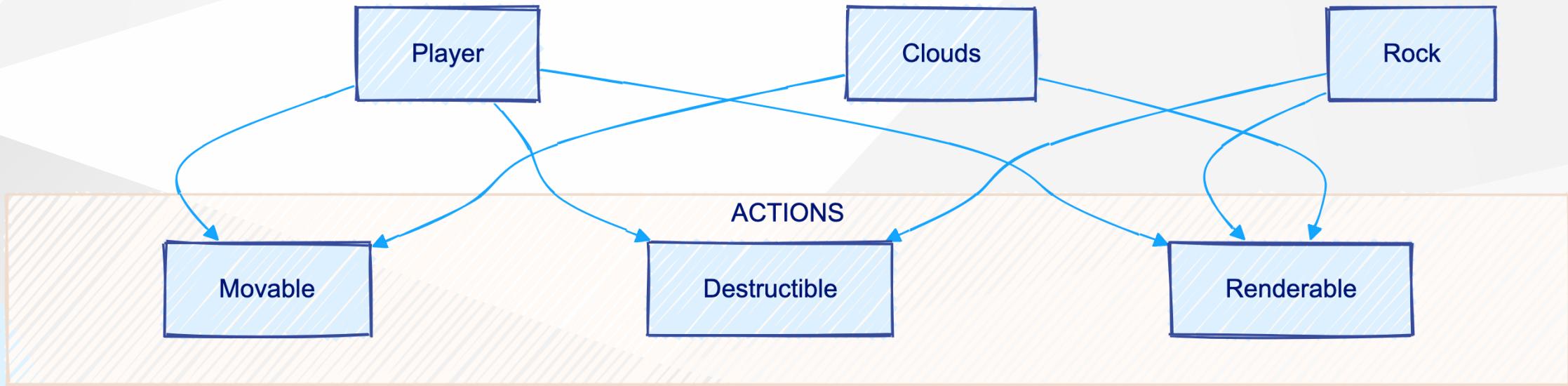


+



=





Kontrakt

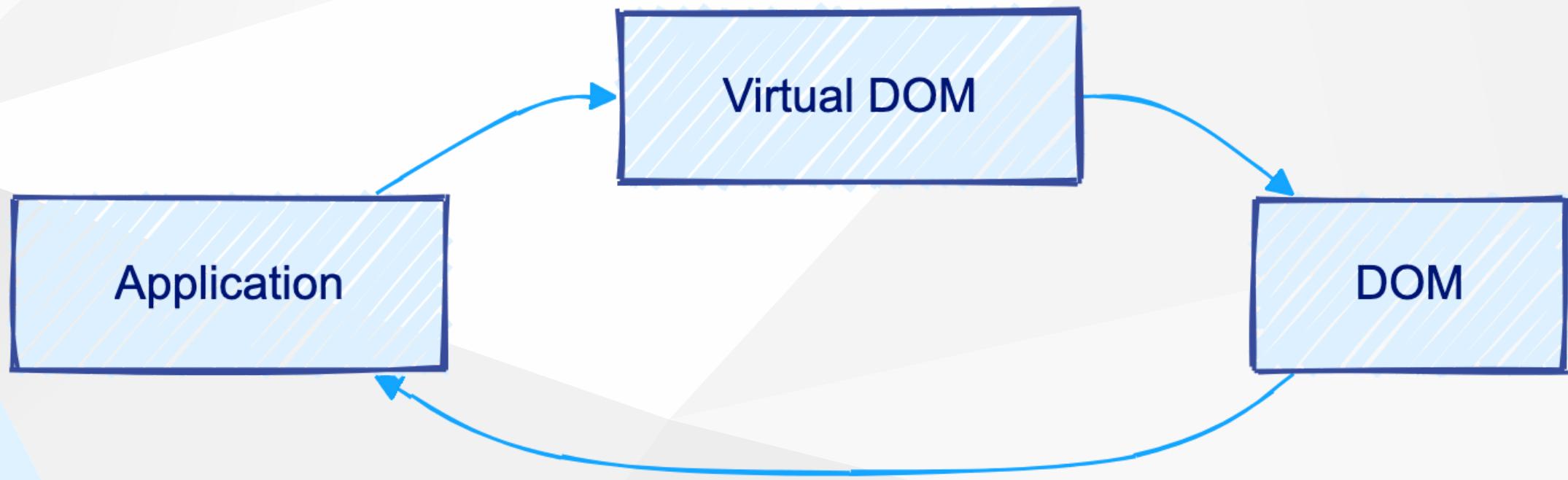
- ◀ każdy komponent powinien mieć jasno zdefiniowany interfejs (kontrakt)
- ◀ kontrakt określa, jakie dane komponent przyjmuje
- ◀ kontrakt powinien być jawny
- ◀ kontrakt powinien realizować zasadę Open/Closed (otwarty na rozszerzenia, zamknięty na modyfikacje) - komponent powinien być otwarty na rozbudowę (przez propsy/sloty), ale zamknięty na modyfikację (nie musimy zmieniać jego kodu źródłowego, by zmienić jego zachowanie)
- ◀ API komponentu powinno być stabilne i dobrze udokumentowane
- ◀ zmiany w API wymagają dyscypliny: semver , deprecations , changelog

Architektura komponentowa działa najlepiej, gdy komponenty mają wysoką spójność, niskie sprzężenie i stabilne kontrakty, a ich reużywalność jest wspierana przez wersjonowanie, testy kontraktowe i dokumentację.

**React nie renderuje zmian bezpośrednio, ale
poprzez Virtual-DOM**

VDOM

- ◆ VirtualDOM to uproszczona reprezentacja struktury DOM (Document Object Model)
- ◆ React buduje drzewo Virtual DOM w sposób deklaratywny, a następnie przygotowuje na jego podstawie drzewo DOM w przeglądarce
- ◆ w przypadku wystąpienia zmian w strukturze, React na nowo przygotuje drzewo Virtual DOM dla zmienionych komponentów
- ◆ biblioteka ReactDOM porównuje aktualne i nowe drzewo Virtual DOM, a następnie aktualizuje tylko te miejsca, które wymagają zmiany
- ◆ VDOM należy traktować jako obiekt JS



zarządzanie poprzez DOM vs VDOM

	React (Virtual DOM)	Angular (Real DOM)	Svelte (Direct DOM)
Wydajność	Większa wydajność aktualizacji DOM	Aktualizacja DOM przy większych aplikacjach może być mniej wydajna	Bardzo wysoka, dzięki komplikacji
Złożoność	Średnia (deklaratywność + Virtual DOM)	Wysoka (dużo funkcji wbudowanych)	Niska (prostota + brak Virtual DOM)
Rozmiar pakietu	Większy (przez Virtual DOM runtime)	Duży	Najmniejszy
Ekosystem	Duży	Rozbudowany ekosystem	Najmniejszy

Plusy użycia VDOM

- ◀ **efektywność aktualizacji** : React używa Virtual DOM do minimalizacji bezpośrednich operacji na rzeczywistym DOM, porównuje różnice (diff) między stanami wirtualnego DOM i aktualizuje tylko zmienione elementy w rzeczywistym DOM
- ◀ optymalizuje wydajność w aplikacjach z dużą ilością dynamicznych zmian
- ◀ **deklaratywność** - React pozwala programiście skupić się na co powinno się zmienić, a nie jak, to czyni kod bardziej zrozumiałym i mniej podatnym na błędy
- ◀ **cross-platform** - dzięki temu podejściu React może działać nie tylko w przeglądarce (React DOM), ale również na innych platformach (np. React Native)

Minusy użycia VDOM

- ◀ **warstwa abstrakcji** - dodatkowa warstwa (Virtual DOM) może prowadzić do narzutu wydajnościowego w prostych aplikacjach, gdzie bezpośrednia manipulacja DOM mogłaby być szybsza
- ◀ **brak pełnej kontroli nad DOM** - Virtual DOM ukrywa szczegóły rzeczywistego DOM, co może być problemem w specyficznych przypadkach wymagających szczegółowych optymalizacji

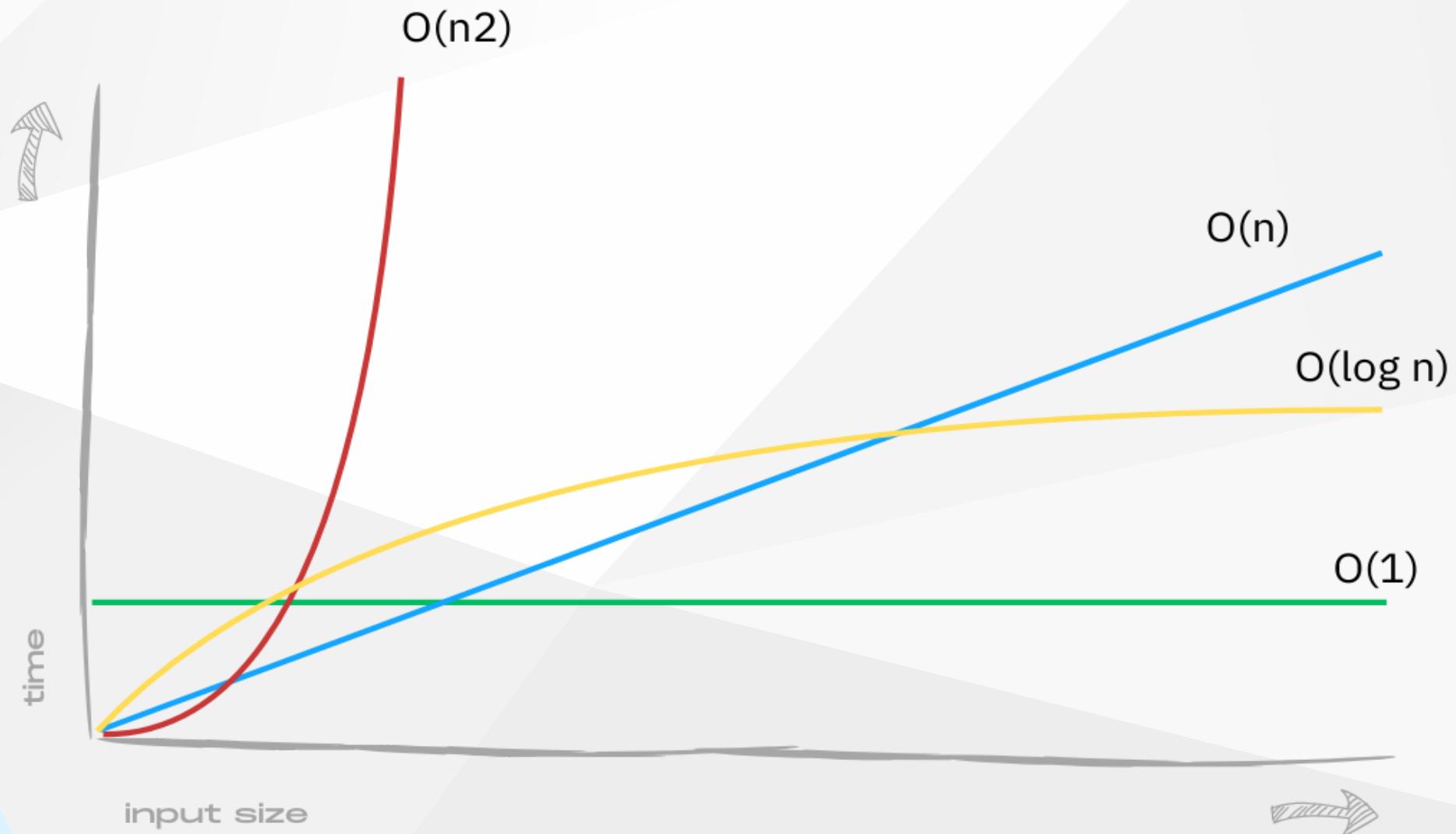
Virtual DOM

Zezwala na generowanie abstrakcyjnego DOM, który może być renderowany jako UI (User Interface) na wielu platformach:

- ▶ react-dom - przeglądarkowy DOM
- ▶ React Native - natywne aplikacje dla iOS i Android
- ▶ react-blessed - terminal
- ▶ react-canvas - elementy HTML Canvas
- ▶ react-vr / react 360 - aplikacje 3D

Diffing algorithm (algorytm różnicujący)

- ▶ pozwala Reactowi wykrywać zmiany i aktualizować tylko te fragmenty DOM, które naprawdę tego wymagają
- ▶ działa z złożonością (Big O notation) $O(n)$, gdzie n to liczba węzłów w drzewie, osiąga to dzięki rekurencyjnemu porównywaniu drzew i użyciu kluczy w listach
- ▶ chociaż algorytm jest szybki, nadal istnieje pewien narzut związany z utrzymywaniem i porównywaniem Virtual DOM - jest to tzw. koszt abstrakcji
- ▶ od 2017 roku React używa algorytmu o nazwie **Fiber**, który wprowadza koncepcję przerwanych i asynchronicznych aktualizacji, co pozwala na bardziej responsywne aplikacje



Concurrent Rendering

- ▶ pozwala Reactowi na bardziej elastyczne i responsywne aktualizacje interfejsu użytkownika
- ▶ umożliwia przerwanie długotrwałych operacji renderowania i kontynuowanie ich w późniejszym czasie
- ▶ wprowadza nowe mechanizmy, takie jak `startTransition`, które pozwalają na oznaczanie niepilnych aktualizacji

Przykładowy kod:

```
import { startTransition, useState } from 'react';
function App() {
  const [count, setCount] = useState(0);

  const handleClick = () => {
    startTransition(() => { setCount(c => c + 1) });
  };

  return (
    <div>
      <button onClick={handleClick}>Increment</button>
      <p>Count: {count}</p>
    </div>
  );
}
```

Elementy o różnych typach

- ▶ w procesie różnicowania React porównuje dwa drzewa Virtual DOM (aktualne i nowe)
- ▶ proces sprawdzania rozpoczyna od sprawdzenia typów elementów najwyższego poziomu (root)
- ▶ kiedy typy są różne (np. **SECTION** i **DIV**) - React niszczy stare drzewo, wraz zinstancjami wcześniejszych komponentów i na podstawie nowego Virtual DOM renderuje na nowo aplikację

Podczas tworzenia drzewa na nowo:

- ▶ stare elementy DOM są niszczone
- ▶ instancje komponentów otrzymają informację, że komponent zostanie odmontowany
- ▶ nowe drzewo zostanie wyrenderowane w miejsce starego
- ▶ nowe komponenty otrzymają informację, że zostały zamontowane

Elementy DOM o tym samym typie

Kiedy elementy DOM mają ten sam typ:

- ◀ React sprawdza argumenty (właściwości) obu porównywanych elementów i w kolejnym kroku aktualizuje tylko te wartości, które wymagają zmiany, np. `styles` , `className`
- ◀ bez niszczenia drzewa idzie do dzieci i rozpoczyna różnicowanie na elementach potomnych

Komponenty elementów o tym samym typie

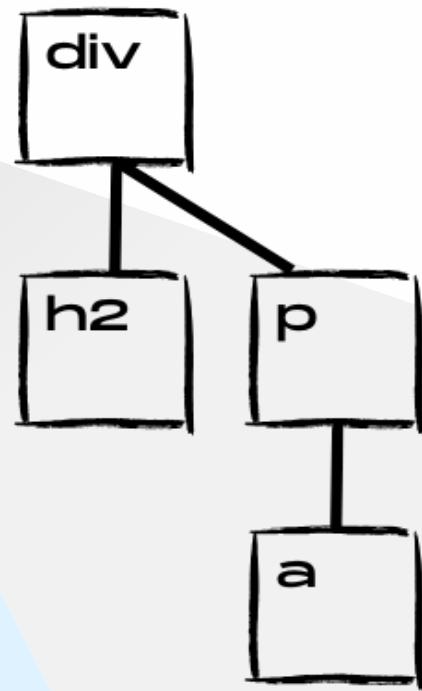
Kiedy są takie same:

- ◆ pomiędzy re-renderowaniemami instancja komponentu pozostaje niezmieniona

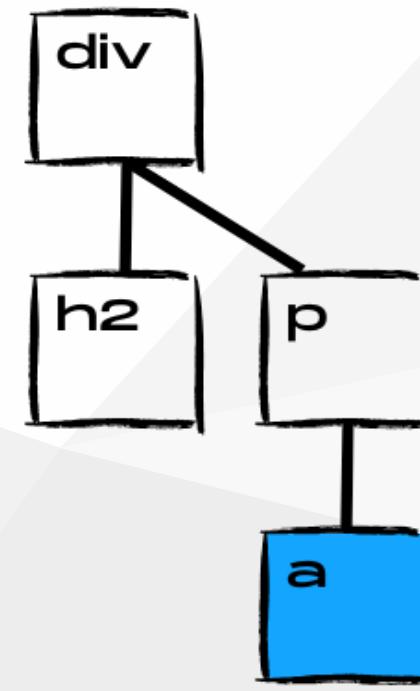
Kiedy w komponencie wykryto zmianę:

- ◆ komponent zostanie poinformowany o potrzebie wykonania re-renderowania.

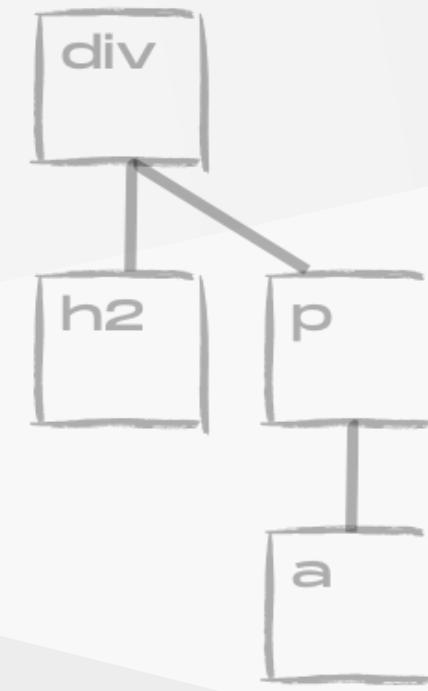
AKTUALNY
DOM



NEW VIRTUAL
DOM



VIRTUAL
DOM



UPDATE

DIFF

Rekurencja na dzieciach

- ◀ domyślnie React iteruje się poprzez listę dzieci w tym samym czasie i generuje mutacje w przypadku napotkania różnic
Jeśli React znajdzie nowy element na końcu listy - doda go do niej
- ◀ gorzej jeśli nowy element pojawi się na początku lub w środku listy - React porównując elementy nie będzie w stanie wykryć, który element jest nowy - w związku z tym wyrenderuje całą listę od momentu wystąpienia różnicy
- ◀ tak przeprowadzona aktualizacja ma bardzo negatywny wpływ na wydajność aplikacji
- ◀ aby uniknąć takich sytuacji - elementy listy powinny otrzymać atrybut `key`

Key

- ◀ element kolekcji powinien mieć dodany atrybut `key`
- ◀ `key` powinien być unikalny i stabilny pomiędzy re-renderowaniami komponentu. Na jego podstawie React jest w stanie rozpoznać elementy, odpowiednio je porównać i podjąć decyzję czy powinny być ponownie wyrenderowane
- ◀ dzięki kluczom React jest w stanie optymalnie wyrenderować listę i pominąć zbędne re-renderowania komponentu, a zatem wykonać na drzewie DOM możliwie najmniej operacji (mutacji)
- ◀ jeśli `key` otrzyma nową wartość, React przerenderuje całą listę ponownie

Za aktualizację drzewa DOM odpowiada algorytm różnicujący (Diffing Algorithm). Proces aktualizacji drzewa DOM to rekoncyliacja.

**Algorytm różnicujący jest tylko szczegółem implementacyjnym. Możemy go pominąć.
Wówczas każda zmiana będzie powodowała re-render całej aplikacji.**

Tworzenie elementów w aplikacji React

Przykładowy kod:

```
React.createElement(  
  'div',  
  {  
    id: 'root_element',  
    className: 'root-element-class',  
    style: {  
      borderTop: '1px solid black',  
    }  
  },  
  "Hello!"  
)
```

JSX

- ◆ składnia przypominająca HTML pozwalająca na używanie rozwiniętej składni HTML w kodzie JS
- ◆ składnia wzbogaca "HTML" o możliwość użycia zmiennych i wyrażeń

Przykładowy kod:

```
const Section = <section>
  <h1>title</h1>
  <h2 className="subtitle" id="hook_to_title">daily</h2>
  <ul>
    <li>ts</li>
    <li>react</li>
  </ul>
</section>

React.render(Section, document.getElementById('app'));
```

Przykładowy kod:

```
const section = {  
  title: 'title',  
  subtitle: 'daily',  
}  
  
const Section = <section>  
  <h1>{section.title}</h1>  
  <h2 className="subtitle" id="hook_to_title">{section.subtitle}</h2>  
  <ul>  
    <li>ts</li>  
    <li>react</li>  
  </ul>  
</section>;  
  
React.render(Section, document.getElementById('app'));
```

Przykładowy kod:

```
const section = {
  title: 'title',
  subtitle: 'daily',
  items: [
    {
      id: 1,
      name: 'ts',
    },
    {
      id: 2,
      name: 'react',
    },
  ],
};

const Section = <section>
  <h1>{section.title}</h1>
  <h2 className="subtitle" id="hook_to_title">{section.subtitle}</h2>
  <ul>
    {section.items.map(item => <li key={item.id}>{item.name}</li>)}
  </ul>
</section>;
```

Komponenty w React

- ▶ komponent to podstawowy blok aplikacji Reactowej
- ▶ komponent to element w strukturze DOM, który jest zarządzany przez React'a
- ▶ **funkcja / klasa komponentu** zawiera kod Javascriptowy, który kontroluje wygląd i zachowanie elementu
- ▶ **instancja komponentu** to element, który został wyrenderowany przez React'a za pośrednictwem swojej klasy / funkcji
- ▶ komponenty mogą być zagnieżdżane w dowolne struktury, podobnie jak HTML czy XML
- ▶ możemy przekazać do komponentu dowolne dane jako atrybuty, w taki sam sposób jak w HTMLu
- ▶ w przeciwieństwie do HTMLa poprzez atrybuty możemy przekazywać również obiekty (a nie tylko stringi).
- ▶ przekazane w ten sposób parametry nazywają się właściwościami komponentu (ang. Component properties, w skrócie **props**)

Przykładowy kod:

```
<MyComponent option={variable} title="text">  
  Text or other components  
</MyComponent>
```

Przykładowy kod:

```
// Pseudoclass ES6

var createReactClass = require('create-react-class');
var Greetings = createReactClass({
  render: function() {
    return React.createElement('h1', {}, 'Hello,' + this.props.name)
    // return <h1>Hello, {this.props.name}</h1>
  }
});
```

Przykładowy kod:

```
// ES6 class

class Greetings extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

Przykładowy kod:

```
// TS class

interface GreetingsProps {
  name: string;
}

class Greetings extends React.Component<GreetingsProps> {
  public render(): ReactNode {
    return <h1>Hello, {this.props.name}</h1>
  }
}
```

Przykładowy kod:

```
// function component  
  
const Greetings = ({ name }) => {  
  return <h1>Hello, {name}</h1>  
}
```

Komponent funkcyjny – wcześniej nazywany jako bezstanowy komponent funkcyjny. Od wprowadzenia Hook'ów komponenty funkcyjne mogą mieć swój stan i nim zarządzać.

Przykładowy kod:

```
// function component in ts

interface GreetingsProps {
  name: string;
}

const Greetings = ({ name }: GreetingsProps) => {
  return <h1>Hello, {name}</h1>
}

const Greetings2: React.FC<GreetingsProps> = ({ name }) => {
  return <h1>Hello, {name}</h1>
}
```

Create React App (deprecated)

Wrapper dla aplikacji Reactowej. CRA ukrywa szczegóły implementacyjne, w szczególności konfigurację bundler'a.

Przykładowy kod:

```
# for typescript  
npx create-react-app my-app --template typescript
```

```
# for javascript  
npx create-react-app my-app
```

vite

Narzędzie do budowania aplikacji frontendowych. Wspiera wiele frameworków, w tym React. Vite jest szybszy niż CRA, szczególnie podczas uruchamiania i pracy w trybie deweloperskim.

Przykładowy kod:

```
npm create vite@latest my-app -- --template react
```

20 minut

zadanie nr 1

- uruchom nowy projekt, wykorzystując create react app

```
npm create vite@latest my-app -- --template react-ts
```

- zmodyfikuj główny komponent App.tsx, tak aby wyświetlał informacje o aplikacji, tj:

- tytuł: "Witaj w naszej aplikacji książkowej"
- polecane książki: listę `ul / li` a w niej 3 tytuły książek
- stopkę z informacją: "Aplikacja przygotowana przez <imię i nazwisko>"



Stan i props

- ◆ React „reaguje” na zmiany danych w komponentach (w przypadku zmiany zostaje wywołany kolejny render z Virtual DOM)
- ◆ renderowanie może zostać wykonane w dwóch przypadkach:
 - kiedy komponent otrzyma nowe dane poprzez właściwości (`props`) oraz kiedy zmienił się wewnętrzny stan komponentu (`state`)
- ◆ przekazanie nowych props do komponentu lub zmiana wewnętrznego stanu komponentu spowoduje re-render komponentu
- ◆ nigdy nie powinniśmy aktualizować stanu i props’ów ręcznie (taka zmiana nie zostanie wykryta przez Reacta)

Przykładowy kod:

```
import React from 'react';

const Todo = props => (
  <div>
    <h3>{props.title}</h3>
  </div>
);

const App = () => (
  <Todo title="Say hello" />
)
```

Przykładowy kod:

```
import React from 'react';

interface TodoProps {
  title: string;
}

class Todo extends React.Component<TodoProps> {
  render() {
    return (
      <div>
        <h3>{this.props.title}</h3>
      </div>
    )
  }
}

class App extends React.Component {
  render() {
    return (<Todo title="Say hello" />)
  }
}
```

Przykładowy kod:

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

function Clock(props) {
  return <h3>{props.name}</h3>
}

Clock.defaultProps = {
  name: 'Hello'
}
```

Przykładowy kod:

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

class Clock extends Component {
  // default props
  static defaultProps = {
    name: 'Hello',
  }

  constructor(props) {
    super(props);

    // default state
    this.state = { date: new Date() }
  }
}
```

20 minut

zadanie nr 2

- ▶ wydziel tytuł strony do osobnego komponentu `Header`
- ▶ wydziel stopkę strony do komponentu `Footer`
- ▶ przygotuj komponenty `Books` oraz `Book`
- ▶ komponent `Books` powinien renderować książki na podstawie danych otrzymanych przez `props`
- ▶ komponent `Book` powinien wyświetlić tytuł, autora oraz datę publikacji - dane te powinien otrzymać z `props`
- ▶ przykładowe książki możesz pobrać z obiektu dołączonego do zadania



Aktualizacja stanu

- React musi wiedzieć kiedy stan komponentu się zmienił
- nigdy nie zmieniaj stanu bezpośrednio!**
- do zmiany stanu w React służą dwie funkcje: `setState` i hook `useState`
- funkcje zmiany stanu działają asynchronicznie
- kiedy stan jest zależny od poprzedniej wartości stanu powinniśmy użyć funkcji wywołania zwrotnego jako pierwszy argument w funkcji zmiany stanu

Aktualizacja stanu w komponentach klasowych

- zmiany są scalane (kiedy zmieniamy tylko jedną wartość obiektu stanu, pozostałe pozostaną bez zmian)
- użycie wartości stanu zaraz po jego aktualizacji może spowodować błąd (komponent może nie zdążyć się wyrenderować ponownie)
- `this.replaceState({})` - metoda do bezpośredniego nadpisania całego obiektu stanu

Przykładowy kod:

```
this.setState((prevState, props) => {
  return {
    counter: prevState.counter + props.increment,
  }
})
```

Przykładowy kod:

```
interface AProps {  
}  
  
interface AState {  
}  
  
class A extends React.Component<AProps, AState> {  
}
```

Aktualizacja stanu w komponentach funkcyjnych

Zmiany stanu nadpisują poprzedni stan (w przeciwieństwie do komponentów klasowych).

Przykładowy kod:

```
const Product = () => {
  const [isFavorite, setIsFavorite] = useState(false);

  const handleSetAsFavorite = () => {
    setIsFavorite(true);
  }

  return (
    <div>
      <span className="star" onClick={handleSetAsFavorite /}>
    </div>
  )
}
```

10 minut

zadanie nr 3

- ▶ do komponentu `Book` dodaj przycisk, który będzie miał dwa stany: "Dodaj do przeczytanych" lub "Usuń z przeczytanych"
- ▶ po kliknięciu w przycisk stan komponentu powinien się zaktualizować / treść przycisku powinna się zmienić a obok tytułu powinna się pojawić informacja, że książka została przeczytana
- ▶ do wykonania zadania wykorzystaj `useState` i `onClick`



Prop Types

- mechanizm służący do weryfikacji typów i właściwości przekazywanych do komponentów
- dzięki niemu możemy upewnić się, że komponenty otrzymują dane w odpowiednim formacie
- pomaga wykrywać błędy w czasie dewelopmentu
- jeśli nie używamy w projekcie Typescript zaleca się korzystanie z `propTypes`

Przykładowy kod:

```
class Comp extends React.Component {  
}  
  
Comp.propTypes = {  
  name: PropTypes.string,  
}
```

Dostępne typy w Prop Types

Typ	Opis
PropTypes.string	Właściwość powinna być typu <code>string</code>
PropTypes.number	Właściwość powinna być typu <code>number</code>
PropTypes.bool	Właściwość powinna być typu <code>boolean</code>
PropTypes.array	Właściwość powinna być typu <code>array</code>
PropTypes.object	Właściwość powinna być typu <code>object</code>
PropTypes.func	Właściwość powinna być funkcją
PropTypes.node	Dowolny element renderowalny (tekst, liczba, element, komponent)
PropTypes.element	Element React (np. lub komponent)
PropTypes.any	Właściwość może mieć dowolny typ

validatory zaawansowane

Typ	Opis
<code>PropTypes.arrayOf(PropTypes.number)</code>	Tablica elementów określonego typu (np. liczby)
<code>PropTypes.objectOf(PropTypes.string)</code>	Obiekt z wartościami określonego typu
<code>PropTypes.oneOf(['red', 'blue'])</code>	Jedna z wymienionych wartości
<code>PropTypes.oneOfType([PropTypes.string, PropTypes.number])</code>	Jeden z kilku typów
<code>PropTypes.shape({ key: PropTypes.type })</code>	Obiekt o określonej strukturze
<code>PropTypes.exact({ key: PropTypes.type })</code>	Obiekt o dokładnie określonej strukturze

10 minut

zadanie nr 4

- ▶ za pomocą PropTypes opisz propsy wchodzące do komponentów `Book` oraz `Books`
- ▶ zwróć uwagę, że komponent `Books` przyjmuje tablicę obiektów w jednym kształcie - warto do opisania tych obiektów użyć `PropTypes.shape`



zdarzenia

- ◆ zdarzenia w React możemy rejestrować bezpośrednio na elementach za pośrednictwem specjalnych props'ów, prefixowanych słowem `on`
- ◆ zdarzeniami możemy zarządzać z poziomu kodu komponentu
- ◆ funkcję możemy również wykonać bezpośrednio podczas deklaracji zdarzenia

Przykładowe propsy dla zdarzeń:

- ◆ `onClick` - zdarzenia kliknięcia w element
- ◆ `onChange` - zdarzenie zmiany wartości pola formularza
- ◆ `onSubmit` - zdarzenie wysłania formularza

Synthetic Event

- obiekty zdarzeń w React mają inny typ niż obiekty zdarzeń w przeglądarce
- `SyntheticEvent` to klasa, która opakowuje natywny obiekt zdarzenia
- interfejs `SyntheticEvent` jest zgodny z interfejsem natywnego zdarzenia
- w przypadku konieczności odwołania się do natywnego zdarzenia, możemy wykorzystać właściwość `nativeEvent`
- React normalizuje zdarzenia, tak by ich właściwości były jednakowe w różnych przeglądarkach
- aby zarejestrować procedurę obsługi zdarzenia w fazie przechwytywania (ang. capturing phase), dodaj na końcu nazwy `Capture` (np. `onClickCapture`)
- dla specyficznych zdarzeń React przygotował specjalne typy, np. dla zdarzenia `change` istnieje typ `ChangeEvent`

Przykładowy kod:

```
const Input = () => {
  const [value, setValue] = useState()

  return (
    <input value={value} onChange={(e: SyntheticEvent) => setValue(e.target.value)} />
  )
}
```

Przykładowy kod:

```
class Input extends React.Component {  
  handleChange(e) {  
    let value = e.currentTarget.value;  
    this.setState({value: value})  
  }  
  
  render() {  
    return (  
      <input value={this.state.value} onChange={this.handleChange} />  
    )  
  }  
}  
  
const Input = () => {  
  const [value, setValue] = useState()  
  
  const handleChange = (e) => setValue(e.target.value);  
  
  return <input value={value} onChange={handleChange} />  
}
```

Formularze

- ▶ domyślnie pola formularza w React oznaczone są jako pole niekontrolowane (uncontrolled components)
- ▶ pola niekontrolowane nie są kontrolowane w pełni przez mechanizm Reacta (za zmianę ich wartości odpowiada przeglądarka)
- ▶ jeśli do pola formularza dodany atrybut `value`, wówczas pole zmienia się w pole kontrolowane (controlled component)
- ▶ pole kontrolowane jest kontrolowane przez Reacta a zatem zmiana jego wartości może odbyć się tylko za pośrednictwem mechanizmu Reacta (tj. poprzez zmianę stanu lub propsów - w większości przypadków poprzez zmianę stanu)
- ▶ kiedy pole jest kontrolowane tylko komponent może zmienić jego wartość (na UI pole nie będzie reagowało bez implementacji funkcji zmiany wartości)
- ▶ kod `<input value={this.state.myValue} />` sprawia, że pole staje się niemożliwe do edycji ręcznie
- ▶ elementy typu `<select>` i `<textarea>` również mogą przyjmować atrybut `value`

zarządzanie formularzem za pomocą stanu

- do zarządzania formularzem w React możemy wykorzystać stan komponentu
- każda zmiana wartości pola formularza powinna być obsłużona przez funkcję, która zaktualizuje stan komponentu

Przykładowy kod:

```
import React, { useState } from 'react';

const MyForm = () => {
  const [inputValue, setInputValue] = useState('');

  const handleChange = (event: React.ChangeEvent<HTMLInputElement>) => {
    setInputValue(event.target.value);
  };

  return (
    <form>
      <input type="text" value={inputValue} onChange={handleChange} />
    </form>
  );
};
```

zarządzanie formularzem za pomocą referencji

- do zarządzania formularzem w React możemy też wykorzystać referencje
- referencje pozwalają na bezpośredni dostęp do elementu DOM

Przykładowy kod:

```
import React, { useRef } from 'react';

const MyForm = () => {
  const inputRef = useRef<HTMLInputElement>(null);
  const handleSubmit = (event: React.FormEvent<HTMLFormElement>) => {
    event.preventDefault();
    inputRef.current && console.log(inputRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input type="text" ref={inputRef} />
      <button type="submit">Submit</button>
    </form>
  );
};


```

Kiedy używać stanu a kiedy referencji?

- ◀ stan jest bardziej odpowiedni, gdy wartość pola formularza jest potrzebna do renderowania komponentu lub do **walidacji danych**
- ◀ referencje są bardziej odpowiednie, gdy potrzebujemy bezpośredniego dostępu do elementu DOM, np. do ustawienia **focus** lub do integracji z bibliotekami zewnętrznymi
- ◀ w większości przypadków zarządzanie formularzem za pomocą stanu jest bardziej zgodne z filozofią Reacta i ułatwia utrzymanie spójności
- ◀ **JEDNAK** w niektórych sytuacjach użycie referencji może być bardziej efektywne, np. gdy mamy bardzo duży formularz i chcemy uniknąć nadmiernych renderów lub gdy chcemy szybko odczytać wartości pól dopiero przy **submit**
- ◀ wybór między stanem a referencjami zależy od konkretnego przypadku użycia!

30 minut

zadanie nr 5

- ▶ przygotuj komponenty `Input` oraz `Form`
- ▶ załóż, że komponent `Input` w propsach może przyjąć wszystkie atrybuty, które może mieć normalny `input` html'owy oraz prop `label`, który dodać `<label>` do komponentu
- ▶ komponenty `Input` i `Form` powinny też przyjąć propsa z akcją, którą wykonają - odpowiednio `onChange` oraz `onSubmit`
- ▶ na podstawie tych komponentów przygotuj komponent `ContactForm`, który umieścisz w stopce i którego zadaniem będzie wysłanie zapytania (chwilowo będzie to po prostu wypisanie informacji w `console` przeglądarki)
- ▶ formularz powinien zawierać pola `name` , `message` , `phoneNumber`



zagnieżdżanie komponentów

- komponenty mogą być zagnieżdżane
- każdy kod JSX, który jest przekazany pomiędzy znacznikiem otwarcia i zamknięcia komponentu, będzie dostępny w tym komponencie jako props `children`
- jako dzieci komponentu możemy zagnieżdżać stringi, tablice elementów JSX i wszystko co wypełnia typ `ReactNode`

Przykładowy kod:

```
const LastItem = () => <p>I am last!</p>

const List = ({ data, children }) => {
  return (
    <ul>
      {data.map(item => <li key={item.id}>{item.title}</li>)}
      {children}
    </ul>
  )
}

const App = () => (
  <List data={[]}>
    <LastItem />
  </List>
)

export default List;
```

ReactNode

ReactNode to szeroko stosowany typ w ekosystemie React, który może reprezentować dowolne treści renderowalne w aplikacji. Jest to złożony typ używany do określenia, co może być przekazane jako dzieci (**children**) lub zawartość komponentu.

Przykładowy kod:

```
type ReactNode =  
  ReactElement  
  string  
  number  
  boolean  
  null  
  undefined  
  ReactFragment  
  ReactPortal;
```

ReactNode czy ReactElement?

	ReactNode	ReactElement
Opis	Dowolna treść, która może być wyrenderowana.	Pojedynczy element React (wynik <code>React.createElement</code>).
Może zawierać	String, number, JSX, null, array, boolean, etc.	Tylko elementy React (JSX lub <code>React.createElement</code>).
Typowy użytku	children props, zawartość renderowalna.	Konkretne elementy React.

Prop drilling (także: threading) to proces przekazywania danych przez propsy w dół drzewa komponentów do komponentu docelowego.

Przykładowy kod:

```
const UserBox = ({ user }) => <div>{user.name}</div>;  
  
const UserProfile = ({ user }) => <UserBox user={user} />;  
  
const Menu = ({ user }) => <UserProfile user={user} />;  
  
const Nav = ({ user }) => <Menu user={user} />;  
  
const App = () => {  
  const user = {  
    name: 'Mateusz',  
  };  
  
  return <Nav user={user} />  
}
```

10 minut

zadanie nr 6

- załóż, że nagłówek `Header` może przyjąć prop `children`
- dodaj do nagłówka nowo przygotowany komponent, który wyświetli informacji o zalogowanym użytkowniku poprzez `children`



95

Cykl życia komponentu

- ◀ każdy komponent ma zaimplementowany mechanizm cyklu życia
- ◀ cykl życia zawiera metody, które połączone są z poszczególnymi etapami istnienia komponentu w strukturze Virtual DOM
- ◀ etapy te pozwalają nam kontrolować komponent od jego zamontowania do odmontowania w drzewie Virtual DOM
- ◀ pozwalają nam np. na wykonywanie dodatkowych operacji w wybranych przez nas momentach istnienia komponentu

Fazy cyklu życia

Faza	Metoda cyklu	Opis
Montowanie	<code>UNSAFE_componentWillMount()</code>	przed zamontowaniem w DOM
Montowanie	<code>componentDidMount()</code>	po zamontowaniu w DOM
Aktualizacja	<code>UNSAFE_componentWillReceiveProps(newProps)</code>	komponent otrzyma nowe propsy
Aktualizacja	<code>getDerivedStateFromProps(newProps)</code>	komponent otrzymał nowe propsy
Aktualizacja	<code>shouldComponentUpdate(nextProps, nextState)</code>	jeśli zwróci <code>false</code> , React pominie następny render
Aktualizacja	<code>UNSAFE_componentWillUpdate()</code>	komponent będzie renderowany, nie zmieniaj stanu
Aktualizacja	<code>componentDidUpdate()</code>	komponent jest wyrenderowany, DOM jest stabilny

Faza	Metoda cyklu	Opis
Aktualizacja	<code>getSnapshotBeforeUpdate(prevProps, prevState)</code>	tuż przed aktualizacją struktury DOM w przeglądarce
Odmontowywanie	<code>componentWillUnmount()</code>	przed usunięciem komponentu z DOM
Obsługa wyjątków	<code>getDerivedStateFromError(newProps)</code>	
Obsługa wyjątków	<code>componentDidCatch()</code>	

Metody do obsługi wyjątków są wykonywane w przypadku wystąpienia błędów podczas procesu renderowania, w metodach cyklu życia lub w metodach konstruktora komponentów potomnych.

Nie jest zalecane manipulowanie strukturami DOM wygenerowanymi przez Reacta, ale istnieje taka możliwość. Aby uzyskać dostęp do elementów DOM możemy użyć mechanizmu referencji.

Przykładowy kod:

```
class Form extends React.Component {  
  constructor(props) {  
    super(props);  
  
    this.inputRef = React.createRef();  
  }  
  
  componentDidMount(): void {  
    this.inputRef.current.focus();  
  }  
  
  render(): React.ReactNode {  
    return (  
      <form>  
        <input ref="inputRef"/>  
      </form>  
    )  
  }  
}
```

Przykładowy kod:

```
const Form = ({ data }) => {
  const inputRef = useRef<HTMLInputElement>(null);

  const handleSubmit = (e) => {
    e.preventDefault();

    inputRef.current && inputRef.current.focus();
  }

  return (
    <form onSubmit={handleSubmit}>
      <input ref={inputRef}/>
    </form>
  )
}
```

Manipulacje na DOM

Aby uzyskać dostęp do elementu DOM możemy użyć:

- ◆ hook'a `useRef`
- ◆ stworzyć referencję w komponencie klasowych za pomocą `createRef`
- ◆ `ReactDOM.findDOMNode(this.refObj.current);` - przykład ten jest rodzajem wyjścia awaryjnego, React odradza używanie tej funkcji, ponieważ zaburza abstrakcję struktury komponentów

Ponadto:

- ◆ pamiętajmy, że wartości referencji będą dostępne tylko gdy komponent zostanie wyrenderowany
- ◆ używajmy referencji rozważnie, powinny być wykorzystywane tylko w specyficznych metodach cyklu życia
- ◆ przechowywanie danych w referencji spowoduje pominięcie re-renderowania komponentu po zmiany tych danych

Przykładowy kod:

```
const Form = () => {  
  return (  
    <form>  
      <input ref={element => element.focus()} />  
    </form>  
  )  
}
```

15 minut

zadanie nr 7

- ▶ sprawdź wysokość listy książek (użyj `useRef` oraz `getBoundingClientRect`)
- ▶ jeśli wysokość jest wyższa niż 500px to wypisz nad listą informację: "Masz masę książek"
- ▶ jeśli jest mniejsza to wypisz "Zbieraj dalej"



forwardRef

`forwardRef` to funkcja umożliwiająca przekazanie referencji (ref) z rodzica do elementu DOM wewnątrz komponentu potomnego, zwykłe komponenty funkcyjne w React nie obsługują ref, więc `forwardRef` pozwala obejść to ograniczenie

Przykładowy kod:

```
import React, { forwardRef } from 'react';

const Input = forwardRef((props, ref) => (
  <input {...props} ref={ref} />
));

```

Przykładowy kod:

```
import React, { useRef } from 'react';

function Parent() {
  const inputRef = useRef();

  return (
    <div>
      <Input ref={inputRef} />
      <button onClick={() => inputRef.current.focus()}>
        Skup się na polu tekstowym
      </button>
    </div>
  );
}
```

20 minut

zadanie nr 8

- do napisanego wcześniej formularza kontaktowego dodaj przycisk "Napisz do nas", który po kliknięciu "postawi" kurSOR w pierwszym polu formularza
- użyj do wykonania tego zadania `useRef` i `.focus()`
- użyj `useRef` z pozycji komponentu rodzica, wykorzystaj do tego `forwardRef`



React Hooks

- ▶ funkcje w Javascript nie posiadają stanu - oznacza to, że każde wywołanie funkcji powoduje stworzenie jej instancji na nowo, a zatem funkcja ma dostęp tylko do danych, które przekazywane są przez parametry lub do zmiennych z zasięgów wyższych (closure)
- ▶ w React komponenty są funkcjami - dlatego domyślnie nie mogą trzymać stanu pomiędzy kolejnymi wywołaniami (re-renderami)
- ▶ hooki zostały wprowadzone w React 16.8
- ▶ hooki są funkcjami, które mogą być umieszczane wewnętrz komponentu, ale tylko na najwyższym poziomie (top scope)
- ▶ hooki dodają do komponentów funkcyjnych możliwości używania referencji, obsługi stanu i wywoływanie efektów ubocznych
- ▶ hooki mogą być wywoływane tylko w komponentach Reactowych lub w innych własnych hookach
- ▶ możemy tworzyć własne hooki (nazwa funkcji hook'a muszą rozpoczynać się od słowa kluczowego `use`)

Kolejność wywoływania Hooków jest ważna!

Jak działają Hooki pod maską?

- ◀ każda instancja komponentu ma własną listę hooków dzięki czemu React zapewnia izolację stanu hooków
- ◀ stan, efekty i referencje są przechowywane osobno dla każdej instancji, co zapobiega konfliktom między różnymi komponentami
- ◀ podczas renderowania komponentu React buduje wewnętrzną listę hooków a każdy wywołany hook jest przypisywany do odpowiedniego indeksu w tej liście
- ◀ hooki, takie jak `useState` i `useEffect`, wykorzystują domknięcia, aby "zapamiętać" dostęp do zmiennych stanu i funkcji między renderami

Przykładowy kod:

```
function Counter() {  
  const [count, setCount] = useState(0);  
  
  const increment = () => setCount(count + 1); // "count" jest zapamiętane w domknięciu  
  
  return <button onClick={increment}>Count: {count}</button>;  
}
```

useState

- ◆ jest wykorzystywany do zarządzania stanem komponentu
- ◆ podczas wywoływania tego hook'a możemy przekazać do niego wartość (będzie to wartość, która zostanie ustawiona jako domyślny stan)
- ◆ w komponencie możemy wywołać ten hook więcej niż jeden raz do zarządzania różnymi stanami dla różnych celów
- ◆ dobrą praktyką jest grupować elementy stanu wg domeny biznesowej
- ◆ `useState` nadpisuje cały stan (w przeciwieństwie do klasowego `setState`)
- ◆ wywołanie `useState` zwraca tablicę dwuelementową (tuple / krotka) - pierwsza wartość to zmienna przechowująca bieżący stan, druga to funkcja zmieniająca stan
- ◆ domyślny stan może być również funkcją - wówczas podczas tworzenia domyślnego stanu domyślna wartość będzie wartością zwroconą z wywołania tej funkcji

Przykładowy kod:

```
const Item = ({ data }) => {
  const [ clicked, setClicked ] = useState(0);

  const handleClick = () => {
    setClicked((prevValue) => prevValue + 1);
  }

  return (
    <div onClick={handleClick}>I am clicked {clicked} times</div>
  )
}
```

Przykładowy kod:

```
function MyComponent(props) {
  const [stateValue, setStateValue] = React.useState(() => {
    return calculations(props);
  });
}
```

15 minut

zadanie nr 9

- do każdej książki dodaj przycisk "Głosuj na tę pozycję"
- na liście pokaż informację, ile głosów oddano na daną pozycję
- użyj `useState` ale z założeniem, że wartość domyślna będzie wynikiem następującej operacji `<liczba liter w nazwisku autora> * 2 / 5` - pamiętaj, że liczba musi być liczbą całkowitą (użyj do tego `Math.round()` lub `Math.floor()`)



useEffect

- ▶ `useEffect` zezwala nam na wykonywanie „side effect’ów”
- ▶ po każdym renderowaniu DOM możemy wywołać imperatywne operacje na komponencie lub mające wpływ na aspekty nie dotyczące komponentu za pomocą hooka `useEffect`

Przykładowe zdarzenia uboczne:

- ▶ zmiana tytułu okna przeglądarki
- ▶ pobranie danych z serwera / api

Przykładowy kod:

```
const WindowTitleHandler = () => {
  const [title, setTitle] = useState('Default title')

  useEffect(() => {
    document.title = title;
  })

  return <div>
    <button onClick={() => setTitle('New title')}></button>
  </div>
}
```

Każde renderowanie komponentu powoduje nowe wywołanie efektu. Działa to podobnie do metod cyklu życia: `componentDidMount()` i `componentDidUpdate()`

Przykładowy kod:

```
const WindowResizeWatcher = () => {
  const [windowWidth, setWindowWidth] = useState(window.innerWidth);

  const handleResizeAction = () => {
    setWindowWidth(window.innerWidth);
  }

  useEffect(() => {
    window.addEventListener('resize', handleResizeAction);

    return () => {
      window.removeEventListener('resize', handleResizeAction);
    }
  })

  return <div>
    Moje okno ma teraz {windowWidth}px szerokości
  </div>
}
```

useEffect w różnych wersjach

- ◀ drugim argumentem hooka `useEffect` jest tablica zależności (wartości od których zależy ponowne wywołanie danego useEffectu)
- ◀ jeśli drugi argument nie został przekazany, useEffect wywoła się przy każdym renderze
- ◀ jeśli drugi argument jest pustą tablicą, useEffect wywoła się tylko raz (podczas pierwszego renderu)
- ◀ jeśli drugi argument zawiera zależności, useEffect wywoła się jeśli którakolwiek z zależności ulegnie zmianie (**Uwaga na wartości referencyjne!**)

Przykładowy kod:

```
useEffect(() => {}, []);
// only after first render

useEffect(() => {});
// after every render

useEffect(() => {}, [ props.title, value ]);
// after change of dependencies
```

Jak działa useEffect za kulismi?

1. Podczas pierwszego renderowania:

- ▶ funkcja efektu jest dodawana do listy efektów
- ▶ efekt zostaje uruchomiony po renderowaniu (w fazie "commit")

2. Podczas kolejnych renderowań:

- ▶ React porównuje tablicę zależności z poprzednią wersją
- ▶ jeśli zależności się zmieniły, efekt jest ponownie uruchamiany

3. Cleanup:

- ▶ jeśli efekt zwraca funkcję, React wywołuje ją przed uruchomieniem nowego efektu lub przed odmontowaniem komponentu

Fazy render i commit

- ◀ **Render Phase** to faza obliczeniowa, w której React przygotowuje VDOM, analizując zmiany w state i props. W tej fazie React nie manipuluje rzeczywistym DOM.
- ◀ **Commit Phase** to moment, w którym React aktualizuje DOM i wykonuje efekty uboczne związane z komponentem (np. pobieranie danych, subskrypcje). Jest to etap, w którym zmiany są rzeczywiście wprowadzane do interfejsu użytkownika.

Główne różnice pomiędzy fazami

	Render Phase	Commit Phase
Zmiany w DOM	Nie wprowadza zmian w DOM	Wprowadza zmiany w rzeczywistym DOM
Efekty uboczne	Brak	Uruchamia efekty (<code>componentDidMount</code> , <code>useEffect</code>)
Asynchroniczność	Może być asynchroniczna (w Concurrent Mode)	Zawsze synchroniczna
Czyszczenie zasobów	Brak	Wykonywane w funkcji czyszczącej z <code>useEffect</code> lub <code>componentWillUnmount</code>

Przykładowy kod:

```
function MyComponent() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    console.log('Effect triggered'); // To wykonuje się w fazie Commit
    return () => {
      console.log('Cleanup'); // Wykonywane przy czyszczeniu efektu => Commit Phase
    };
  }, [count]);

  // Render Phase: React oblicza nową wersję wirtualnego DOM, bazując na stanie i propsach
  return <button onClick={() => setCount(count + 1)}>Count: {count}</button>;
}
```

10 minut

zadanie nr 10

- ▶ zmień tytuł okna przeglądarki (`document.title`) tak, aby zawierał liczbę książek dostępnych w naszej aplikacji
- ▶ tytuł powinien brzmieć: "Twój zbiór książek liczy książek"
- ▶ wykorzystaj `useEffect`



125

useLayoutEffect

- ▶ zachowanie podobne do `useEffect`
- ▶ różnica polega na tym, że jest wywoływany synchronicznie po mutacji na strukturze DOM (po renderowaniu)
- ▶ przydatny kiedy chcemy uspójnić widok z danymi w sytuacji bezpośredniej mutacji struktury DOM (np. wszelkiego rodzaju animacje, pomiary DOM)

Inne hooki

Hook	Opis	Przykład zastosowania
<code>useContext</code>	Umożliwia dostęp do danych z Context API bez korzystania z render props.	Dostęp do globalnego stanu aplikacji.
<code>useReducer</code>	Używany do bardziej złożonego zarządzania stanem niż <code>useState</code> .	Zarządzanie stanem formy z wieloma polami.
<code>useMemo</code>	Zapamiętuje wynik funkcji, aby uniknąć ponownego obliczania przy każdym renderze.	Optymalizacja kosztownego obliczenia.
<code>useCallback</code>	Zapamiętuje funkcję, aby nie była tworzona na nowo przy każdym renderze.	Przekazanie funkcji do komponentu potomnego, aby zapobiec niepotrzebnym renderom.
<code>useRef</code>	Umożliwia dostęp do referencji DOM oraz przechowywanie mutowalnych wartości.	Uzyskanie referencji do elementu DOM bez powodowania ponownego renderowania.

Hook	Opis	Przykład zastosowania
<code>useImperativeHandle</code>	Pozwala na dostosowanie wartości zwracanej przez <code>ref</code> w komponentach.	Udostępnianie niestandardowych metod komponentu potomnego.
<code>useDebugValue</code>	Ułatwia debugowanie hooków poprzez dodanie niestandardowych wartości w narzędziach React DevTools.	Pokazanie aktualnego stanu w niestandardowym hooku.
<code>useId</code>	Generuje unikalne identyfikatory, przydatne w dostępności lub powiązaniach formularzy.	Tworzenie unikalnych <code>id</code> dla elementów formularza.
<code>useTransition</code>	Umożliwia oznaczenie pewnych aktualizacji stanu jako "niepilnych".	Zarządzanie animacjami lub ładowaniem przy mniejszym obciążeniu interfejsu.
<code>useDeferredValue</code>	Odkłada aktualizację wartości do momentu, gdy przestanie być pilna.	Optymalizacja interakcji w przypadku dużej liczby elementów na ekranie.

15 minut

zadanie nr 11

- ▶ dodaj funkcjonalność usuwania książek
- ▶ obok przycisku przeczytane dodaj przycisk "Usuń"
- ▶ po kliknięciu książka powinna zniknąć z listy a lista się odświeżyć
- ▶ użyj funkcji tablicowej `filter` oraz `useState`



129

Komunikacja z API

- ▶ zapytania do API w ramach aplikacji React mogą być wykonywane w ramach różnych strategii
- ▶ React nie dostarcza jednego rozwiązania tego problemu
- ▶ do obsługi zapytań możemy wykorzystać zewnętrzne narzędzie i/lub użyć natywnych rozwiązań z funkcjami React'a

useEffect + fetch / axios

- ▶ najprostszy sposób na pobieranie danych przy renderowaniu komponentu
- ▶ można używać w połączeniu z `fetch`, `axios` czy innymi bibliotekami HTTP

Przykładowy kod:

```
const List = (props: any) => {
  const [data, setData] = useState([]);

  const fetchData = async () => {
    const res = await fetch('//example.com/api/data');
    const data = await res.json();

    setData(data);
  }

  useEffect(() => {
    fetchData();
  }, [])
}

return <div>
  {data.map(item => <ListItem />)}
</div>
}
```

Przykładowy kod:

```
const [code, setCode] = useState([]);  
const [isLoading, setIsLoading] = useState(false);  
  
const fetchCode = () => {  
  setIsLoading(true);  
  fetch('//example.com/api/data')  
    .then(res => res.json())  
    .then(data => {  
      if (data.status === 404) {  
        throw new Error()  
      }  
  
      setCode(data);  
    })  
    .catch((err) => {  
      // handle error  
    })  
    .finally(() => {  
      setIsLoading(false);  
    })  
}  
}
```

Przykładowy kod:

```
const [isLoading, setIsLoading] = useState(false);

const fetchData = async () => {
  setIsLoading(true);
  try {
    const res = await fetch('//example.com/api/data');
    const data = await res.json();

    if (!res.ok) {
      throw new Error()
    }

    setData(data);
  } catch(err) {

  } finally {
    setIsLoading(false);
  }
}

useEffect(() => {
  fetchData();
}, []);
```

React Query lub RTK Query (TanStack Query)

- ▶ narzędzia do zarządzania zapytaniami do API
- ▶ obsługują cache, powtarzanie zapytań, buforowanie zapytań, synchronizację zapytań pomiędzy sobą
- ▶ RTK Query jest wbudowany w Redux Toolkit oraz łączy zarządzanie stanem globalnym z funkcjonalnościami fetch

SSR/SSG z Next.js

- ▶ pobieranie danych po stronie serwera w `getServerSideProps` lub `getStaticProps`
- ▶ redukuje obciążenie klienta i przyspiesza renderowanie

25 minut

zadanie nr 12

- ▶ endpoint <https://jsonplaceholder.typicode.com/users> zwraca listę użytkowników (czytelnicy)
- ▶ przygotuj listę czytelników naszej aplikacji, która będzie miała 2 stany: ukryty (wyświetli tylko informację o ilości użytkowników) oraz rozwinięty (wyświetli listę użytkowników)
- ▶ pamiętaj o dodanie przycisku który będzie ukrywał i pokazywał listę
- ▶ na liście rozwiniętej powinny być widoczne takie dane jak imię, email, username, website
- ▶ użyj `useEffect` i `fetch` do pobrania danych



136

20 minut

zadanie nr 13

- ▶ napisz swój własny hook do pobierania danych z API



137

Router

- ▶ pozwala zastąpić renderowany komponent na widoku w zależności od bieżącego adresu URL w przeglądarce
- ▶ operuje na obiekcie `location` (oczywiście dostosowanego do potrzeb aplikacji SPA)
- ▶ zezwala na dodanie parametrów do ścieżek
- ▶ zezwala na wykorzystywanie dynamicznych linków (bazujących na zmiennych)

Przykładowy kod:

```
import React from "react";
import ReactDOM from "react-dom/client";
import { BrowserRouter } from "react-router-dom";

import "./index.css";
import App from "./App";

ReactDOM.createRoot(document.getElementById("root")).render(
  <React.StrictMode>
    <BrowserRouter>
      <App />
    </BrowserRouter>
  </React.StrictMode>
);
```

Przykładowy kod:

```
<Routes>
  <Route path="/" element={<Layout />}>
    <Route index element={<Home />} />
    <Route path="about" element={<About />} />
    <Route path="dashboard" element={<Dashboard />} />
    <Route path="post/:id" element={<Post />} />
    <Route path="*" element={<NoMatch />} />
  </Route>
</Routes>
```

Przykładowy kod:

```
<nav>
  <Link to="/">Home</Link>
  <Link to="/about">About</Link>
  <Link to="/dashboard">Dashboard</Link>
  <Link to={`/post/${id}`}>New post</Link>
</nav>
```

Przykładowy kod:

```
const Post = () => {
  const { id } = useParams()

  return (
    <div>
      Post ID: ${id}
    </div>
  )
}
```

Przykładowy kod:

```
const navigate = useNavigate();

navigate(`/transaction/${id}`);
```

React Router v7 jako "framework"

Przykładowy kod:

```
import React from 'react';
import { createBrowserRouter, RouterProvider } from 'react-router-dom';

// Komponenty
import Home from './Home';
import ErrorPage from './ErrorPage';

// Funkcja loadera do ładowania danych dla danej trasy
const homeLoader = () => fetch('/api/homeData')
  .then(res) => res.json()
  .catch(() => []);

// Tworzenie routera
const router = createBrowserRouter([
  {
    path: "/",
    element: <Home />,
    loader: homeLoader, // Użycie loadera do ładowania danych
    errorElement: <ErrorPage />, // Komponent na wypadek błędu
  },
]);

export const App = () => <RouterProvider router={router} />;
```

30 minut

zadanie nr 14

- ▶ zainstaluj React Router `npm i react-router`
- ▶ dodaj `BrowserRouter` jako Provider dla całej aplikacji
- ▶ przygotuj ścieżki dla: strony głównej (informacji o tym co to za aplikacja, listy książek, pojedynczej książce i formularza kontaktowego)
- ▶ strona główna powinna mieć ścieżkę `/`
- ▶ lista książek `/books`
- ▶ pojedyncza książka `/books/:id`
- ▶ formularz kontaktowy `/contact`
- ▶ dodaj nawigację w nagłówku aplikacji wykorzystując komponent z `react-router`

Przykładowy kod:

```
<Link to="/">Home</Link>
```



css inline

Przykładowy kod:

```
const User = ({ name }) => (
  <div style={{
    display: 'flex',
    color: 'blue'
  }}>
    <h2 style={{
      fontSize: '1.1rem',
      textTransform: 'uppercase'
    }}>{name}</h2>
  </div>
)
```

ICSS

- ▶ w 2015 roku wprowadzono rozszerzenie do specyfikacji CSS, którego głównym zadaniem było umożliwienie ograniczenia zasięgu stylów do poziomu komponentu, jednocześnie nadal pozostawiając selektory dostępne z poziomu zasięgu globalnego
- ▶ rozszerzenie to nazwano Interoperable CSS (ICSS)
- ▶ ICSS w zasadzie dodał dwa pseudoselektory CSS `:import` oraz `:export`
- ▶ głównym zadaniem była możliwość wyeksportowania na zewnątrz stylów i zainportowania ich w takich sposób, aby udostępnione style były dostępne jako obiekt z lokalnymi aliasami jako jego właściwościami
- ▶ lokalny alias to po prostu nazwa składająca się z liter i cyfr
- ▶ otworzyło to drogę do rozwoju CSS Modules

CSS Modules

- ▶ domyślny sposób deklarowania stylów w CRA
- ▶ pozwala na wykorzystanie preprocessor'ów takich jak Sass
- ▶ pliki ze stylami są traktowane jako moduły
- ▶ każdy komponent posiada swój własny plik / moduł ze stylami
- ▶ nazwa każdego modułu ze stylami powinna być zakończona rozszerzeniem `.module.${ext}`
- ▶ bardzo łatwy sposób na enkapsulację stylów (enkapsulacja za pomocą generowanej ID dodawanego do nazw klas)
- ▶ bardzo dobra adaptacja w przeglądarce - nie ma większych problemów z wydajnością, ponieważ docelowo style są traktowane tak samo jak zwykłe style CSS
- ▶ klasy CSS są dostępne na poziomie komponentu w obiekcie ze stylami
- ▶ style globalne powinny być trzymane w osobnym pliku / katalogu

Przykładowy kod:

```
import React, { FunctionComponent } from 'react';
import styles from './Counter.module.scss';

const Counter: FunctionComponent<CounterProps> = ({ days, nextItemName }) => {
  return (
    <div className={styles.counter}>
      <div className={styles.counterText}>
        <span className={styles.nextTitle}>Następny {nextItemName}</span>
        <span className={styles.remainingDays}>{prepareDaysLabel()}</span>
      </div>
      <div className={styles.icon}>
        <Image src={calendar || `/icons/calendar.svg`} width={24} height={24} alt="" />
      </div>
    </div>
  )
}

export default Counter;
```

- ◆ pierwsza próba tworzenia stylów z wykorzystaniem Javascript
- ◆ koncepcja została zaproponowana i przygotowana w 1996 roku przez firmę Netscape (wówczas to rozwiązanie nie uzyskało wystarczająco dużego wsparcia przeglądarki i upadło, funkcjonowało wówczas pod nazwą JSSS)
- ◆ współcześnie JSS jest podobnym konceptem trzymania CSS w JS za pomocą obiektów Javascriptowych
- ◆ zasadność wykorzystania JSa do tworzenia CSS opiera się na dużej reużywalności obiektów i sporymi możliwościami enkapsulacji stylów
- ◆ JSS nie jest powiązany z żadnym frameworkm
- ◆ aby używać JSS w React możemy użyć pakietu react-jss

Przykładowy kod:

```
jss.setup(preset())
const styles = {
  '@global': {
    a: {
      textDecoration: 'underline'
    }
  },
  withTemplates: `
    background-color: green;
  `,
  button: {
    fontSize: 12,
    '&:hover': {
      background: 'blue'
    }
  },
  ctaButton: {
    extend: 'button',
    '&:hover': {
      background: color('blue').darken(0.3).hex()
    }
  },
  '@media (min-width: 1024px)': {
    button: {
      width: 200
    }
  }
}
```

Styled components

- ▶ bazuje na rozwiązaniach JSS
- ▶ biblioteka do tworzenia stylów jako komponentów
- ▶ wykorzystuje Javascriptowy mechanizm tagged templates
- ▶ styled komponent może być tworzony jako dedykowany dla innego komponentu React lub jako samodzielny działający dla całej aplikacji
- ▶ zachowują się i działają w sposób taki jak inne elementy JSX
- ▶ zezwala na rozszerzania komponentów React poprzez dodawanie stylów do nich
- ▶ styled komponenty są reużywalne, mogą korzystać z propsów i trzymać logikę powiązaną z zarządzaniem stylami

Przykładowy kod:

```
import styled from 'styled-components';

const Header = styled.div`  
  background-color: #ddd;  
  color: blue;  
  border-bottom: 2px solid;  
  padding: ${({ padding }) => `${padding}px`}  
  
// usage  
  
<Header padding={20}>
```

Przykładowy kod:

```
const functionA = value => {  
  return value[0] + 'string';  
}  
  
functionA`test`; // teststring
```

Czym są Tagged Templates?

- zaawansowana funkcja w JS, która umożliwia manipulację literalami szablonowymi (template literals) za pomocą specjalnych funkcji zwanych tag functions
- dzięki temu możemy dynamicznie przekształcać ciągi znaków lub fragmenty kodu

Przykładowy kod:

```
function myTag(strings, ...values) {  
  // strings: tablica stałych fragmentów szablonu  
  // values: tablica interpolowanych wartości  
  return `Modified: ${strings.join('')} | ${values.join(', ')}`;  
}  
  
const name = "Alice";  
const age = 25;  
  
const result = myTag`Name: ${name}, Age: ${age}`;  
console.log(result); // Modified: Name: , Age: | Alice, 25
```

Tagged templates w praktyce

Tagged templates pozwalają na manipulację szablonem przed jego ostatecznym wygenerowaniem, np. sanitizację danych, translację czy logowanie.

Przykładowy kod:

```
function sanitize(strings, ...values) {
  return strings.reduce((result, str, i) => {
    const safeValue = String(values[i] || '').replace(/</g, "&lt;").replace(/>/g, "&gt;");
    return result + str + safeValue;
  }, '');
}

const userInput = "<script>alert('x')</script>";
const output = sanitize`User input: ${userInput}`;
console.log(output); // User input: &lt;script&gt;alert('x')&lt;/script&gt;
```

Tailwind

- ▶ framework wydany w 2019 roku
- ▶ stylowanie za pomocą atomowych klas CSS, tzw. utility classes
- ▶ pisanie kodu CSS jest szybkie i intuicyjne
- ▶ brak predefiniowanych komponentów, opiera się o tzw. Utility classes

Najważniejsze cechy Tailwind CSS

- ◀ **podejście Utility-First** - nie piszemy własnych klas css, korzystamy tylko z gotowych odpowiedzialnych za konkretne działanie, np (`text-center`)
- ◀ **elastyczność** - nie mamy narzuconego konkretnego wyglądu
- ◀ **łatwość konfiguracji** - cała konfiguracja w jednym pliku `tailwind.config.js`, gdzie możemy zdefiniować też własne kolory itp.
- ◀ **responsywność** - Tailwind dostarcza wbudowane klasy odpowiedzialne za responsywność, np.: `md:text-xl`
- ◀ **jakość kodu** - brak nadmiarowego kodu (zombie css)
- ◀ **integracja z narzędziami** - Tailwind działa świetnie z frameworkami takimi jak React, Vue czy Angular
- ◀ **ekosystem** - rozbudowana dokumentacja i bogata społeczność, ponadto istnieją dodatkowe rozszerzenia, takie jak Tailwind UI (płatne komponenty), czy pluginy wspierające rozwój aplikacji

Przykładowy kod:

```
export default function App() {
  return (
    <div className="text-center">
      <h1 className="text-3xl font-bold underline">
        Hello, world!
      </h1>
    </div>
  );
}
```

Przykładowy kod:

```
// tailwind.config.js
module.exports = {
  content: [
    "./src/**/*.{js,jsx,ts,tsx}",
  ],
  theme: {
    extend: {},
  },
  plugins: [],
}
```

Przykładowy kod:

```
/* src/index.css */
@tailwind base;
@tailwind components;
@tailwind utilities;
```

30 minut

zadanie nr 15

- ◀ dodaj do projektu Material UI (dokumentacja)[<https://mui.com/material-ui/getting-started/overview/>]
- ◀ z wykorzystaniem MUI dodaj komponent **Avatar**, aby wyświetlić zdjęcie użytkownika (<https://placeholder.co/150>)
- ◀ za pomocą **CSS modules** ostyluj listę książek
- ◀ za pomocą **Styled Components** ostyluj nawigację, stopkę oraz zmień styl awatara tak, aby otrzymał dodatkową ramkę wokoło



- ▶ w projekcie CRA możemy deklarować i używać zmiennych środowiskowych
- ▶ nazwy zmiennych środowiskowych powinny być prefixowane za pomocą REACT_APP_
- ▶ zadeklarowane zmienne nie powinny przechowywać sekretów
- ▶ nigdy nie trzymaj sekretów w repozytorium
- ▶ pamiętaj, że wartości ze zmiennych środowiskowych w komponentach klienckich zostaną przeniesione do kodu produkcyjnego

Rodzaje plików .env

Plik	Opis
.env	Zasadniczy plik ze zmiennymi
.env.development	Używany w środowisku deweloperskim
.env.production	Używany podczas produkcyjnego builda
.env.test	Używany podczas testów
.env.local	Nadpisuje wartości w innych .env plikach dla lokalnego środowiska
.env.development.local	Specyficzny dla lokalnego środowiska deweloperskiego
.env.production.local	Specyficzny dla lokalnych ustawień produkcyjnych

**Nigdy nie trzymaj sekretów w aplikacji React –
(przede wszystkim prywatnych kluczy API)**

Przykładowy kod:

```
# .env.local  
REACT_APP_API_URL=http://localhost:3000
```

Przykładowy kod:

```
const buildUrl = (endpoint: string): string =>  
  `${process.env.REACT_APP_API_URI}/${endpoint}`;
```

20 minut

zadanie nr 16

- ▶ dodaj plik .env.local
- ▶ zadeklaruj zmienną środowiskową REACT_APP_API_URI z wartością
<https://jsonplaceholder.typicode.com>
- ▶ przebuduj naszego hook'a do wykonywania zapytań w taki sposób, aby pobierał uri z plików środowiskowych, a do parametrów przekazywalibyśmy tylko endpoint
- ▶ obsłuż możliwość pobierania plików z `public` (wówczas api url powinien zostać zignorowany)



165

zarządzanie uprawnieniami

Przykładowy kod:

```
enum Roles {
  ADMIN = 'admin',
  USER = 'user',
}

interface ProtectedRouteType<T = {}> {
  children: ReactNode;
  necessaryRole?: Roles;
  userRole?: Roles;
  auth: boolean;
}

const ProtectedRoute = ({ userRole, auth, necessaryRole, children }: ProtectedRouteType) => {
  const canActivate = () => !(auth === true && userRole === necessaryRole);

  if (!canActivate()) {
    return <Navigate to="/" replace />
  }

  return <>{children}</>;
}
```

Przykładowy kod:

```
<Routes>
<Route path="/" index element={<HomePage />} />
  <Route path="/history" element={<TransactionsPage />} />
  <Route path="/protected" element={
    <ProtectedRoute auth={false}>
      <TransactionsPage />
    </ProtectedRoute>
  } />
</Routes>
```

25 minut

zadanie nr 17

- ▶ przygotuj zabezpieczoną ścieżkę dla widoku z czytelnikami (jeśli nie masz takiego widoku dodaj nowy **route**)
- ▶ jeśli nastąpi próba wejścia na widok czytelników bez uprzedniego zalogowania, przekieruj użytkownika na widok z informacją, że nie ma uprawnień do przeglądania tej strony



Portale

- ◀ mechanizm, który zezwala na renderowanie komponentów React poza miejscem w strukturze, gdzie powinny one zostać wyrenderowane
- ◀ zezwala na referowanie elementów w dowolnym miejscu w DOM lub w oknach zależnych (new Window)
- ◀ portale mogą być użyteczne, gdy stylowanie komponentów ogranicza możliwości poprawnego renderowania komponentów, np. gdy komponent ma ustawiony `z-index` lub `overflow` z wartością `hidden`
- ◀ element renderowany za pośrednictwem Portalu jest nadal pod kontrolą React
- ◀ komponenty renderowane w Portalu mogą współdzielić stan, mają dostęp do przekazywanych propsów i do context'u
- ◀ event bubbling (bąbelkowanie) będzie działało w ten sam sposób jak bez Portalu, dlatego zdarzenia odpalone wewnątrz Portalu będą propagowana w górę drzewa React

Przykładowy kod:

```
import { createPortal } from 'react-dom';

const Modal = ({ children }) => {
  return createPortal(
    <div>{children}</div>,
    document.getElementById('modal'),
  )
}
```

Przykładowy kod:

```
<body>
  <div id="app"></div> /* React app */
  <div id="modal"></div> /* Portal */
</body>
```

20 minut

zadanie nr 18

- ▶ z użyciem React Portal przygotuj modal, który będzie wyświetlał się po kliknięciu zaloguj
- ▶ logowanie odbędzie się dopiero po kliknięciu "Zaloguj" w modalu



Context API

- ◀ context pozwala na przekazywanie danych w drzewie komponentów bez przekazania ich poprzez propsy elementów potomnych
- ◀ jeśli użycie Context API ma pozwolić nam uniknąć prop drillingu przez kilka poziomów drzewa React, łatwiejszych i lepszych rozwiązaniami będzie użycie **kompozycji komponentów**
- ◀ kiedy Context podejmuje decyzję co powinno być wyrenderowane ponownie sprawdza referencje elementów, oznacza to, że w niektórych przypadkach kolejne rendery rodziców Providera Contextu mogą wywołać kolejne niechciane re-rendery wszystkich konsumentów (Consumer) dla konkretnego contextu
- ◀ każdy obiekt context'u ma swój własny komponent dostawcy (Provider'a), który zezwala komponentom subskrybować się na zmiany w tym context'cie
- ◀ komponent React, który subskrybuje się na zmiany w context'cie to Consumer - Consumer może nasłuchiwać na zmiany w ramach swojej struktury

Przykładowy kod:

```
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // Use a Provider to pass the current theme to the tree below.
    // Any component can read it, no matter how deep it is.
    // In this example, we're passing "dark" as the current value.
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
}
```

Przykładowy kod:

```
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}

function CountDisplay() {
  const {count} = React.useContext(CountContext)
  return <div>{count}</div>
}
```

Problemy z Context API

- ◀ kiedy stan w kontekście się zmienia, wszystkie komponenty, które korzystają z tego kontekstu, zostaną ponownie wyrenderowane, nawet jeśli ich dane wcale się nie zmieniły
- ◀ zarządzanie złożonym stanem w Context API może być trudne i nieefektywne - każda zmiana jakiejkolwiek części stanu może prowadzić do ponownego renderowania całego drzewa komponentów
- ◀ trudności z typowaniem w TypeScript
- ◀ w dużych aplikacjach, kiedy działa wiele kontekstów nieodpowiednie zarządzanie nimi może prowadzić do chaosu - używanie zbyt wielu kontekstów może utrudnić zarządzanie danymi i ich przepływem
- ◀ przekazanie funkcji do kontekstu (np. funkcji do aktualizacji stanu), może prowadzić do problemów z referencjami, każda zmiana funkcji spowoduje ponowne renderowanie komponentów

25 minut

zadanie nr 19

- ▶ w naszej aplikacji chcemy obsłużyć dwa theme'y: `dark` i `light`
- ▶ przygotuj context, który będzie przechowywał informacje o wybranych theme
- ▶ dodaj przycisk w nagłówku do zmiany theme'u
- ▶ w stopce dodaj informację o aktualnie wybranych theme



Kompozycja

- ◀ kluczowy wzorzec projektowy w React, pozwala na tworzenie złożonych komponentów poprzez łączenie mniejszych, wielokrotnego użytku
- ◀ React preferuje kompozycję zamiast dziedziczenia (które jest częstsze w tradycyjnych OOP)
- ◀ kompozycja sprzyja elastyczności i modularności aplikacji

Główne założenia kompozycji:

- ◀ komponenty mogą być używane jako dzieci innych komponentów lub jako ich elementy składowe
- ◀ dzięki przekazywaniu dzieci (`props.children`) lub funkcji, komponenty rodziców mogą definiować, jak komponenty dzieci powinny być renderowane

props.children

Przykładowy kod:

```
function Card({ children }) {
  return (
    <div style={{ border: '1px solid #ddd', padding: '10px', borderRadius: '5px' }}>
      {children}
    </div>
  );
}

function App() {
  return (
    <Card>
      <h2>Tytuł</h2>
      <p>To jest zawartość karty.</p>
    </Card>
  );
}
```

render props, czyli funkcje renderujące

Przykładowy kod:

```
function DataFetcher({ render }) {
  const data = ['Element 1', 'Element 2', 'Element 3'];
  return <div>{render(data)}</div>;
}

const App = () => (
  <DataFetcher
    render={(data) => (
      <ul>
        {data.map((item, index) => (
          <li key={index}>{item}</li>
        )))
      </ul>
    )}
  />
)
```

Wzorzec „Slot”

Przykładowy kod:

```
function Layout({ header, content, footer }) {
  return (
    <div>
      <header>{header}</header>
      <main>{content}</main>
      <footer>{footer}</footer>
    </div>
  );
}

function App() {
  return (
    <Layout
      header={<h1>Nagłówek</h1>}
      content={<p>To jest główna treść.</p>}
      footer={<small>Stopka</small>}
    />
  );
}
```

Higher Order Component

Przykładowy kod:

```
function withLogger(Component) {
  return function WrappedComponent(props) {
    console.log('Renderowanie komponentu z props:', props);
    return <Component {...props} />;
  };
}

function MyComponent({ message }) {
  return <p>{message}</p>;
}

const MyComponentWithLogger = withLogger(MyComponent);

function App() {
  return <MyComponentWithLogger message="Cześć!" />;
}
```

Code splitting

- ▶ wraz z rozrastającym się kodem aplikacji, rośnie również rozmiar plików produkcyjnych
- ▶ Code Splitting jest mechanizmem wspieranym przez takie narzędzia jak: Webpack, Rollup, Browserify
- ▶ Code Splitting zezwala na tworzenie dużej ilości małych paczek kodu, które mogą być ładowane na żądanie w trakcie działania aplikacji
- ▶ dobrym miejscem na podział aplikacji za pomocą Code Splittingu są ścieżki w routerze
- ▶ sposoby na Code splitting w aplikacji React: `import()` oraz `React.lazy()`

Przykładowy kod:

```
const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />}/>
        <Route path="/about" element={<About />}/>
      </Routes>
    </Suspense>
  </Router>
);
```

Default export vs named export

Komponent, który importujemy należy eksportować domyślnie. Wówczas składnia importu jest prostsza.

Brak eksportu domyślnego powoduje błąd, ponieważ `lazy` działa tylko z eksportami domyślnymi. Poniższy przykład pokazuje jak zmapować `named export` na `default export`, gdy musimy zachować `named export`.

Przykładowy kod:

```
const Home = lazy(() => import('./routes/Home')).then((module) => ({ default: module.Home }));
```

Przykładowy kod:

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

class App extends Component {
  handleClick = () => {
    import('./moduleA')
      .then(({ moduleA }) => {
        // użyj modułu A
      })
      .catch(err => {
        // obsługa błędów
      });
  };

  render() {
    return (
      <div>
        <button onClick={this.handleClick}>Load</button>
      </div>
    );
  }
}

export default App;
```

Przykładowy kod:

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

15 minut

zadanie nr 20

- ▶ wykorzystując code splitting podziel naszą aplikację wg ładowanych ścieżek
(użyj `React.lazy` i `<Suspense />`)



188

Concurrent Features

- ▶ nazywany w poprzednich wersjach Concurrent Mode
- ▶ zestaw funkcji w React, które umożliwiają bardziej płynne i responsywne renderowanie interfejsu użytkownika
- ▶ zaprojektowane, aby poprawić wydajność aplikacji poprzez lepsze zarządzanie czasem potrzebnym na renderowanie oraz interakcje z użytkownikiem
- ▶ w skład wchodzą między innymi takie funkcjonalności jak: `Suspense` , `useTransition` , `useDeferredValue`

useTransition

- ▶ hook wprowadzony w React 18
- ▶ umożliwia zarządzanie przejściami stanu w aplikacjach, poprawiając wrażenie płynności i responsywności interfejsu użytkownika
- ▶ pozwala oznaczyć część aktualizacji stanu jako „niepilną” i przeprowadzić ją w tle, bez blokowania innych ważniejszych aktualizacji, takich jak responsywność interfejsu

Przykładowy kod:

```
const [isPending, startTransition] = useTransition();
```

- ▶ isPending - boolean, wskazujący, czy niepilne aktualizacje są w toku
- ▶ startTransition - funkcja, która otacza niepilne operacje.

Przykładowy kod:

```
import { useState, useTransition } from 'react';

function SearchableList({ items }) {
  const [filteredItems, setFilteredItems] = useState(items);
  const [isPending, startTransition] = useTransition();

  const handleInputChange = (e) => {
    // Filtruj elementy w liście
    startTransition(() => {
      setFilteredItems(
        items.filter((item) => item.toLowerCase().includes(newQuery.toLowerCase())))
    );
  });
}

return (
  <div>
    {isPending && <p>Updating list...</p>}
    <ul>
      {filteredItems.map((item, index) => (
        <li key={index}>{item}</li>
      ))}
    </ul>
  </div>
);
}
```

Memoizacja

- ◀ technika optymalizacji w programowaniu polegająca na przechowywaniu wyników funkcji dla określonych zestawów argumentów, aby uniknąć wielokrotnego przeliczania tych samych wyników w przyszłości
- ◀ memoizacja poprawia wydajność, szczególnie w przypadku funkcji, które są kosztowne obliczeniowo lub są często wywoływane z tymi samymi argumentami
- ◀ przy pierwszym wywołaniu funkcji z danym zestawem argumentów, wynik obliczeń jest zapisywany w specjalnej strukturze danych, zazwyczaj w obiekcie
- ◀ przy kolejnych wywołaniach z tymi samymi argumentami funkcja nie wykonuje obliczeń, lecz zwraca zapisany wcześniej wynik

Gdzie przyda się memoizacja?

Poniższy kod nie jest zbyt efektywny. Funkcja `fibonacci` wykonuje się rekurencyjnie wiele razy z tymi samymi argumentami.

Czas: 797ms

Przykładowy kod:

```
function fibonacci(n) {  
    if (n <= 1) return n;  
    return fibonacci(n - 1) + fibonacci(n - 2);  
}  
  
console.log(fibonacci(40));
```

Po dodaniu memoizacji

Czas: 0.09ms

Przykładowy kod:

```
function fibonacciMemo() {  
  const memo = {};  
  
  function fibonacci(n) {  
    if (n in memo) return memo[n]; // Sprawdź, czy wynik jest zapisany  
    if (n <= 1) return n;  
    memo[n] = fibonacci(n - 1) + fibonacci(n - 2); // Zapisz wynik  
    return memo[n];  
  }  
  
  return fibonacci;  
}  
  
const fibonacci = fibonacciMemo();  
console.log(fibonacci(40));
```

React.memo

- ▶ **React.memo** dba o to, aby komponent renderował się tylko, gdy zmienią się jego propsy
- ▶ **React.memo** porównuje propsy za pomocą porównania płytkego (shallow comparison)
- ▶ dla propsów referencyjnych (np. tablice, obiekty, funkcje) sprawdzana jest tylko ich referencja, a nie rzeczywista zawartość

Przykładowy kod:

```
const Component = ({ name }) => {
  return <h2>{name}</h2>
}

export const MemoizedComponent = React.memo(Component);
```

Drugi argument w memo

- ▶ drugi argument w funkcji `React.memo` to opcjonalna funkcja porównania (ang. comparison function)
- ▶ pozwala na bardziej zaawansowane kontrolowanie, kiedy komponent powinien być ponownie renderowany
- ▶ domyślnie `React.memo` porównuje tylko płytko (shallow comparison) poprzednie i nowe propsy, ale w niektórych przypadkach, zwłaszcza gdy propsy zawierają złożone obiekty lub funkcje, może być konieczne użycie własnej logiki porównania

Przykładowy kod:

```
const MyComponent = React.memo(  
  (props) => {  
    // renderowanie komponentu  
  },  
  (prevProps, nextProps) => {  
    // funkcja porównania  
    return prevProps.value === nextProps.value; // przykład porównania  
  }  
);
```

useMemo i useCallback

- ◀ **useMemo** służy do zapamiętywania (memoizacji) wyniku jakiejś obliczeniowej operacji
- ◀ przydatna w przypadku kosztownych obliczeń

Przykładowy kod:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

- ◀ **useCallback** działa podobnie jak **useMemo**, ale zamiast zapamiętywać wynik obliczeń, zapamiętuje samą funkcję
- ◀ przydatne, gdy nie chcemy, aby funkcje były tworzone na nowo przy każdym renderowaniu

Przykładowy kod:

```
const memoizedCallback = useCallback(() => { doSomething(a, b); }, [a, b]);
```

20 minut

zadanie nr 21

- ▶ użyj useMemo memoizacji obiektu z stylami liniowymi w komponencie Book
- ▶ w komponentach związanych z książkami mamy kilka funkcji do filtrowania i usuwania danych - użyj na nich useCallback
- ▶ zmemoizuj komponent z awatarem użytkownika



200

<React.StrictMode>

- ▶ dodany do React w 2018 roku (pierwotnie dotyczył tylko komponentów klasowych)
- ▶ ma za zadanie pomagać deweloperom w aktualizacji aplikacji poprzez rezygnację ze starego, niewspieranego API
- ▶ dodaje zestaw podpowiedzi i ostrzeżeń, aby pomóc uniknąć typowych pułapek związanych z procesem developmentu
- ▶ pomaga uniknąć trzymania w kodzie „nieczystych” funkcji
- ▶ sprawdza czy funkcje komponentów React są idempotentne (pomaga w tym podwójne wywołania kodu)
- ▶ działa tylko w trybie deweloperskim
- ▶ efektem ubocznym jest podwójne renderowanie aplikacji w procesie development (pomaga dzięki temu wyłapać błędy w asynchronicznie działających API Reacta)

Unikaj pułapek w StrictMode

1. nie twórz efektów ubocznych w funkcji renderującej

- ↳ efekty uboczne, takie jak modyfikacja DOM czy zapytania sieciowe, powinny być umieszczane w `useEffect`

2. używaj funkcji czyszczących

- ↳ `useEffect` powinien zwracać funkcję czyszczącą, szczególnie w przypadku: Subskrypcji, Timerów, WebSocket

3. unikaj niekontrolowanego dostępu do zmiennych

- ↳ każda zmienna, która jest używana w efekcie, powinna być dodana do listy zależności (`dependency array`).

React Dev Tools

- ◆ rozszerzenie przeglądarki oraz narzędzie developerskie
- ◆ pozwala na inspekcję i debugowanie aplikacji napisanych w React
- ◆ pozwala lepiej zrozumieć strukturę komponentów, ich stan oraz propsy
- ◆ przydatne przy monitorowaniu wydajności aplikacji

Wtyczki:

- ◆ Chrome: [React Developer Tools for Chrome](#)
- ◆ Firefox: [React Developer Tools for Firefox](#)

Problem server state

Użycie useEffect i fetch:

- brak cache
- brak retry
- brak refetch
- brak synchronizacji
- ręczne `isLoading`
- ręczne error handling

Czym jest TanStack Query?

- ▶ biblioteka do zarządzania server state
- ▶ oparta o hooki
- ▶ bardzo rozbudowana, ale można używać tylko tego, co potrzebujemy
- ▶ powstała jako React Query, ale teraz jest framework-agnostic (React, Vue, Solid, Svelte)
- ▶ bardzo mały bundle (ok. 10KB gzipped)

Przykładowe funkcje:

- ▶ cache
- ▶ background refetch
- ▶ retry
- ▶ deduplikacja zapytań
- ▶ synchronizacja wielu komponentów

QueryClient

QueryClient to klasa, która zarządza cache'em i konfiguracją zapytań. Aby korzystać z React Query, musimy utworzyćinstancję QueryClient i udostępnić ją naszej aplikacji za pomocą QueryClientProvider.

Przykładowy kod:

```
import { QueryClient, QueryClientProvider } from '@tanstack/react-query';

const queryClient = new QueryClient();

root.render(
  <QueryClientProvider client={queryClient}>
    <App />
  </QueryClientProvider>
);
```

useQuery

Przykładowy kod:

```
import { useQuery } from '@tanstack/react-query';

const fetchUsers = async () => {
  const res = await fetch('/users');
  return res.json();
};

const Users = () => {
  const { data, isLoading, error } = useQuery({
    queryKey: ['users'],
    queryFn: fetchUsers,
  });

  if (isLoading) return <p>Loading...</p>;
  if (error) return <p>Error</p>;

  return (
    <ul>
      {data.map(user => <li key={user.id}>{user.name}</li>)}
    </ul>
  );
};
```

Jako zewnętrzny hook

Przykładowy kod:

```
import { useQuery } from '@tanstack/react-query';

export const useUsers = () => {
  return useQuery({
    queryKey: ['users'],
    queryFn: async () => {
      const res = await fetch('/users');
      return res.json();
    },
  });
};
```

Caching

- ◆ dane są cache'owane - domyślnie przez 5 minut
- ◆ ponowny mount = brak refetch - dane są pobierane z cache, a nie z API
- ◆ automatyczne odświeżanie w tle - dane są odświeżane w tle, ale nie blokują interfejsu użytkownika

Stale Time vs Cache Time

staleTime

- ↳ jak długo dane są uznawane za "świeże"
- ↳ w tym czasie nie ma automatycznego refetch

cacheTime (gcTime w v5)

- ↳ jak długo dane pozostają w pamięci po unmount

Przykładowy kod:

```
useQuery({  
  queryKey: ['users'],  
  queryFn: fetchUsers,  
  staleTime: 1000 * 60, // 1 minuta  
});
```

Automatyczny Background Refetch

Refetch on window focus

- ▶ powrót do zakładki = dane się odświeżają
- ▶ reconnect internetu = dane się odświeżają

Przykładowy kod:

```
useQuery({  
  queryKey: ['users'],  
  queryFn: fetchUsers,  
  refetchOnWindowFocus: false,  
});
```

useMutation – coś więcej niż POST

- ▶ `useMutation` to hook, który pozwala na wykonywanie operacji mutujących dane (np. POST, PUT, DELETE) w sposób łatwy i zorganizowany
- ▶ w przeciwieństwie do `useQuery`, `useMutation` nie jest automatycznie wywoływany przy renderze komponentu, ale jest wywoływany ręcznie (np. w odpowiedzi na zdarzenie, takie jak kliknięcie przycisku)

Przykładowy kod:

```
const queryClient = useQueryClient();

const mutation = useMutation({
  mutationFn: addUser,
  onSuccess: () => {
    queryClient.invalidateQueries({ queryKey: ['users'] });
  },
});
```

Użycie w komponencie:

Przykładowy kod:

```
<button onClick={() => mutation.mutate({ name: 'New User' })}>  
  Add User  
</button>
```

15 minut

zadanie

- ▶ zainstaluj `tanstack` w projekcie
`npm install @tanstack/react-query`
- ▶ dodaj `QueryClientProvider` do drzewa komponentów, aby udostępnić klienta zapytań w całej aplikacji
- ▶ przepisz pobieranie czytelników z API, aby korzystało z hooka `useQuery` z TanStack Query
- ▶ dodaj obsługę stanu ładowania i błędów, wykorzystując zwrócone wartości z `useQuery`



214

Globalne zarządzanie stanem upraszcza komunikację między komponentami, ale wprowadza koszt architektoniczny, wydajnościowy i poznawczy.

Globalne zarządzanie stanem

- ◆ pojedynczy obiekt globalny dostępny do odczytu i zmiany z dowolnej części naszej aplikacji
- ◆ wszystkie zmiany i dane mogą być śledzone z jednego miejsca
- ◆ scentralizowane miejsce przechowywania i zarządzania danymi
- ◆ przydatne, gdy dane są potrzebne wielu komponentom w aplikacji, ponieważ zapewnia, że mają one dostęp do najnowszej i najbardziej aktualnej wersji danych
- ◆ komponenty mogą nasłuchiwać zmian w stanie i aktualizować swoje dane automatycznie, kiedy nastąpi zmiana w magazynie
- ◆ dzięki centralizacji unika się chaosu związanego z ręcznym przesyłaniem danych między komponentami, co ułatwia skalowanie aplikacji

Kiedy warto używać globalnego zarządzania stanem?

- ◀ **dane są współdzielone pomiędzy komponentami** - kiedy komponenty z różnych części aplikacji potrzebują tych samych danych, typu: koszyk w ecommerce, dane profilu, token, motyw, język, preferencje użytkownika
- ◀ **złożone zależności pomiędzy komponentami** - kiedy dane przekazywane są przez wiele poziomów aplikacji, pomimo że komponenty pośrednie tych danych nie potrzebują (w React zjawisko to nazywa się "prop drilling")
- ◀ **stan jest często modyfikowany** - kiedy stan aplikacji jest często zmieniany w różnych miejscach w aplikacji, wówczas stan globalny pozwala utrzymać spójność, przykład: system powiadomień
- ◀ **zapytania asynchroniczne i ich stan** - kiedy aplikacja wysyła asynchroniczne zapytania, a ich wynik wpływa na stan aplikacji w różnych jej miejscach, np: lista produktów / promocji - o ile nie korzystamy z dedykowanych narzędzi do zarządzania stanem serwera (np. TanStack Query)
- ◀ **zarządzanie stanem interfejsu użytkownika** - do obsługi globalnych elementów widoku, takich jak modale, toasty, spinners
- ◀ **wymóg spójności danych** - kiedy aplikacja wymaga spójności danych pomiędzy różnymi komponentami - jedna zmiana wpływa na kilka innych miejsc

Kiedy NIE używać globalnego zarządzania stanem?

- ◀ małe aplikacje - najczęściej w małych aplikacjach wystarczy lokalne zarządzanie stanem
- ◀ proste aplikacje bez złożonej logiki - dodanie narzędzia do zarządzania stanem globalnym, będzie dodaniem niepotrzebnej złożoności
- ◀ dane są izolowane - np. gdy dane są specyficzne per komponent
- ◀ gdy komponenty są izolowanymi wyspami - np. w architekturze wyspowej, gdzie każdy komponent zarządza swoim własnym stanem

WAŻNE

Globalne zarządzanie stanem to KOSZT, a nie benefit.

Problem ze stanem lokalnym

- ◀ **prop drilling** - powoduje nadmierne sprzężenie komponentów, utrudnia refaktoryzację i ponowne użycie komponentów
- ◀ **synchronizacja wielu komponentów** - gdy wiele komponentów potrzebuje tych samych danych, synchronizacja stanu między nimi może być trudna i podatna na błędy
- ◀ **stan współdzielony** (auth, koszyk, theme)
- ◀ trudność w debugowaniu
- ◀ **duplikacja logiki**

Przykładowy kod:

```
const [isLoading, setIsLoading] = useState(false);
const [error, setError] = useState(null);
```

3 rodzaje stanu w aplikacji

Rodzaj	Przykład	Narzędzie
UI state	modal, spinner	useState / Context
Client state	koszyk, auth	Zustand
Server state	lista użytkowników	TanStack Query

WAŻNE

Największy błąd to traktowanie server state jako client state.

Czym jest Zustand?

- ▶ minimalistyczny store
- ▶ bez boilerplate
- ▶ bez reducerów
- ▶ bez providera
- ▶ oparty o hooki
- ▶ bardzo mały bundle

Pierwszy store w Zustand

Przykładowy kod:

```
import { create } from 'zustand';

interface CounterStore {
  count: number;
  increment: () => void;
  decrement: () => void;
}

export const useCounterStore = create<CounterStore>((set) => ({
  count: 0,
  increment: () => set((state) => ({ count: state.count + 1 })),
  decrement: () => set((state) => ({ count: state.count - 1 })),
}));
```

WAŻNE

W Zustand store to funkcja, która zwraca hooka. Ten hook pozwala nam na dostęp do stanu i funkcji aktualizujących ten stan.

Użycie w komponencie

Przykładowy kod:

```
const Counter = () => {
  const { count, increment } = useCounterStore();

  return (
    <button onClick={increment}>
      {count}
    </button>
  );
};
```

A Provider?

Zustand nie wymaga providera, ponieważ każdy hook stworzony za pomocą `create` jest niezależny i może być używany bezpośrednio w komponentach. To oznacza, że nie musimy owijać naszej aplikacji w dodatkowy komponent provider, co **upraszcza strukturę naszej aplikacji** i **eliminuje problem zagnieżdżania providerów**.



Selektory

W Zustand, podobnie jak w Redux, możemy definiować selektory, które pozwalają nam na wybieranie konkretnych fragmentów stanu z naszego store'a. Selektory są funkcjami, które przyjmują cały stan jako argument i zwracają tylko te dane, które są nam potrzebne w danym momencie.

Przykładowy kod:

```
const useCount = () => useCounterStore((state) => state.count);
```

PODSUMOWANIE

W powyższym przykładzie `useCount` jest selektorem, który zwraca tylko wartość `count` z naszego store'a. Dzięki temu, gdy używamy `useCount` w naszym komponencie, komponent będzie się re-renderował tylko wtedy, gdy wartość `count` ulegnie zmianie, a nie wtedy, gdy inne części stanu się zmieniają.

Selektory a re-renderowania

Selektor powinien:

- ▶ zwracać możliwie mały fragment stanu
- ▶ zwracać stabilną referencję
- ▶ nie tworzyć nowych obiektów inline (chyba że używamy `shallow`)

WAŻNE

Największym błędem jest zwracanie całego stanu lub tworzenie nowych obiektów inline, co powoduje niepotrzebne re-renderowanie komponentów.

Shallow comparison

- Zustand domyślnie używa `Object.is` do porównywania wartości zwracanych przez selektor (jeśli zwracamy nowy obiekt lub tablicę, komponent będzie się re-renderował nawet jeśli dane w nich są takie same)
- `shallow` to funkcja, która porównuje obiekty lub tablice płytко (sprawdza tylko pierwszą warstwę)
- użycie `shallow` pozwala na zwracanie nowych obiektów lub tablic bez powodowania re-renderowania

Przykładowy kod:

```
import { shallow } from 'zustand/shallow';
const useUser = () => useUserStore((state) => ({ name: state.name, age: state.age }), shallow);
```

Czy shallow to dobra praktyka?

Używanie `shallow` jest przydatne, gdy chcemy zwracać obiekty lub tablice, ale musimy być świadomi, że może to ukrywać problemy z referencjami.

WAŻNE

Jeśli nie jesteśmy ostrożni, możemy przypadkowo zwracać nowe obiekty lub tablice, co może prowadzić do trudnych do debugowania błędów związanych z re-renderowaniem.

Selektory w Zustand są bardzo proste do zdefiniowania i używania, ponieważ korzystają z hooków, co pozwala na łatwe zarządzanie stanem w komponentach funkcyjnych.

Aktualizacja stanu w Zustand

W Zustand mamy dwie opcje aktualizacji stanu:

- ◀ aktualizacja częściowa (partial update)
- ◀ nadpisanie całego stanu (full replacement)

Aktualizacja częściowa vs nadpisanie

Aktualizacja częściowa	Nadpisanie całego stanu
<code>set({ count: state.count + 1 })</code>	<code>set((state) => ({ count: state.count + 1 }), true)</code>
aktualizuje tylko <code>count</code> , reszta stanu pozostaje bez zmian	nadpisuje cały stan, więc jeśli nie uwzględnimy innych właściwości, zostaną one utracone
zalecana w większości przypadków	użyteczna w niektórych sytuacjach, np. resetowanie stanu

Przykładowy kod:

```
const useCounterStore = create<CounterStore>((set) => ({  
  count: 0,  
  increment: () => set((state) => ({ count: state.count + 1 })),  
  reset: () => set({ count: 0 }, true), // nadpisanie całego stanu  
}));
```

Kompozycja middleware

- ▶ middleware w Zustand pozwala na rozszerzenie funkcjonalności store'a poprzez dodanie dodatkowych warstw logiki
- ▶ można je łączyć, tworząc bardziej złożone zachowania (np. logowanie + persist)
- ▶ kolejność middleware ma znaczenie - są wywoływane w kolejności, w jakiej zostały zadeklarowane

Przykładowy kod:

```
import { devtools, persist } from 'zustand/middleware';

const useStore = create(
  devtools(
    persist(
      (set) => ({
        count: 0,
        increment: () => set((state) => ({ count: state.count + 1 })),
      }),
      { name: 'counter-storage' }
    )
  )
);
```

Middleware (persistent state)

- ↳ Zustand oferuje wbudowane middleware `persist`, które pozwala na łatwe przechowywanie stanu w `localStorage` lub `sessionStorage`
- ↳ dzięki temu, stan naszego store'a może być zachowany nawet po odświeżeniu strony
- ↳ `persist` automatycznie serializuje i deserializuje stan, więc nie musimy się martwić o konwersję danych

Przykładowy kod:

```
import { persist } from 'zustand/middleware';

export const useAuthStore = create(
persist(
(set) => {
  token: null,
  setToken: (token) => set({ token }),
}),
{ name: 'auth-storage' }
);
```

zastosowanie:

- ◀ token
- ◀ koszyk
- ◀ theme

persist – ważne opcje konfiguracyjne

Przykładowy kod:

```
persist(  
  (set) => ({  
    token: null,  
    user: null,  
  }),  
  {  
    name: 'auth-storage',  
    partialize: (state) => ({ token: state.token }),  
  }  
)
```

- ◆ **name** - nazwa klucza w localStorage/sessionStorage, pod którym będzie przechowywany stan
- ◆ **partialize** - funkcja, która pozwala na wybieranie tylko części stanu do przechowywania (np. tylko token, bez danych użytkownika)
- ◆ **version** - numer wersji stanu, przydatny do migracji danych
- ◆ **migrate** - funkcja, która pozwala na migrację danych ze starej wersji stanu do nowej, przydatna gdy struktura stanu ulega zmianie

Produkcyjne użycie persist

Persist nie jest bezpieczny dla:

- ◀ wrażliwych tokenów (XSS - localStorage leak)
- ◀ bardzo dużych obiektów
- ◀ frequently-changing state (częste zapisy)

Inne middleware

- ◀ `devtools` - integracja z Redux DevTools
- ◀ `subscribeWithSelector` - pozwala na subskrybowanie zmian stanu z selektorami
- ◀ `immer` - pozwala na mutowanie stanu w sposób deklaratywny
- ◀ `combine` - łączenie kilku store'ów w jeden

zustand to nie tylko React

- ◀ `create` (wrapper Reactowy) vs `createStore` (vanilla store, niezależny od Reacta) - oba API są dostępne, ale `create` jest bardziej popularne w kontekście Reacta
- ◀ Zustand może być używany w dowolnym środowisku JavaScript, nie tylko w React, co czyni go uniwersalnym narzędziem do zarządzania stanem w różnych frameworkach

createBoundstore pattern

Przykładowy kod:

```
const counterStore = createStore((set) => ({
  count: 0,
  increment: () => set((s) => ({ count: s.count + 1 })),
}));

export const useCounterStore = create(counterStore)
```

- ◀ **createBoundStore** to wzorzec, który pozwala na stworzenie store'a niezależnego od Reacta, a następnie "wiązanie" go z hookiem do użycia w komponentach Reactowych
- ◀ pozwala na stworzenie separacji między logiką biznesową a warstwą prezentacji, co ułatwia testowanie i ponowne wykorzystanie kodu
- ◀ daje możliwość wielokrotnego bindowania tego samego store'a w różnych hookach do tej samej instancji, co może być przydatne w przypadku dużych aplikacji z wieloma komponentami korzystającymi z tego samego stanu

globalBoundStore pattern

Przykładowy kod:

```
import { create } from 'zustand';

export const useCounterStore = create((set) => ({
  count: 0,
  increment: () => set((s) => ({ count: s.count + 1 })),
}));
```

- ◀ **globalBoundStore** to wzorzec, który pozwala na stworzenie globalnego store'a, który jest dostępny w całej aplikacji
- ◀ store powstaje przy pierwszym importie modułu
- ◀ store jest tworzony jako singleton modułowy, co oznacza, że jest współdzielony między wszystkimi komponentami, które go używają
- ◀ jest to przydatne, gdy chcemy mieć centralne miejsce do zarządzania stanem, które jest łatwo dostępne w całej aplikacji

Porównanie wzorców

Cecha	globalBoundStore	createBoundStore
Singleton	Tak (modułowy)	Zależne od sposobu tworzenia instancji
SSR safe	Wymaga ostrożności	Umożliwia izolację (izolacja per request)
DI	Trudne	Naturalne
Testowalność	Średnia	Bardzo dobra
Boilerplate	Minimalny	Większy
Separacja domeny od UI	Ograniczona	Pełna

15 minut

zadanie

- ▶ zainstaluj `zustand` w projekcie
`npm install zustand`
- ▶ stwórz globalny store (`globalBoundStore pattern`), który:
 - ▶ przechowuje listę przeczytanych książek (np. jako tablicę ID)
 - ▶ pozwala oznaczyć książkę jako przeczytaną
 - ▶ pozwala cofnąć oznaczenie
 - ▶ pozwala sprawdzić, czy dana książka została przeczytana



243

10 minut

zadanie

- ▶ dodaj middleware `devtools` , aby móc korzystać z rozszerzenia Redux DevTools do debugowania stanu store'a
- ▶ dodaj middleware `persist` do stworzonego wcześniej store'a, aby przechowywać dane w `localStorage` - pamiętaj o odpowiedniej konfiguracji, aby zapisać tylko potrzebne dane (np. listę przeczytanych książek, a nie całą logikę store'a)



244

Cechy	Redux	Zustand	Jotai	Recoil	MobX
Boilerplate	Wysokie	Niskie	Niskie	Niskie	Niskie
Wydajność	Dobra	Bardzo dobra	Bardzo dobra	Dobra	Bardzo dobra
Reaktywność	Brak	Tak	Tak	Tak	Tak
Kompleksowość	Wysoka	Niska	Niska	Średnia	Średnia
Rozmiar	Duży	Mały	Mały	Średni	Średni
Debugowanie	Redux DevTools	Brak DevTools	Brak DevTools	Recoil DevTools	MobX DevTools
Ekosystem	Bardzo duży	Mniejszy	Mniejszy	Średni	Średni
Wsparcie społeczności	Bardzo duże	Średnie	Średnie	Średnie	Średnie
Obsługa hooków	Tak	Tak	Tak	Tak	Tak

Meta-Frameworki oparte o React

Meta-frameworki to frameworki oparte na React, które rozszerzają jego możliwości, dodając dodatkowe funkcje, takie jak: SSR, SSG, renderowanie hybrydowe (ISR), routing, ptrymalizacja wydajności, obsługa API.

Przykłady:

- ◆ Next.js
- ◆ Remix
- ◆ Gatsby
- ◆ Expo (z React Native)
- ◆ Blitz.js

Porównanie popularnych meta-frameworków

Framework	SSR	SSG	ISR	API Routes	Full-Stack	Obsługa danych
Next.js	✓	✓	✓	✓	✓	Dynamiczne ładowanie
Remix	✓	✓	✗	✓	✓	Optymalizacja serwera
Gatsby	✗	✓	✗	✗	✗	Kompilacja (SSG)
Blitz.js	✓	✓	✓	✓	✓	ORM z Prisma
Expo	✗	✗	✗	✗	✓	(Mobile) Natywne

TDD

- ◆ Test Driven Development
- ◆ testy bazują na cyklu: **Red / Green / Refactoring**
 - ◆ **Red** – w tej fazie myślimy o tym co chcemy zaprogramować
 - ◆ **Green** – ta faza pokrywa odpowiedź na pytanie jak zrobić, żeby przeszły testy
 - ◆ **Refactor** – myślimy o tym jak poprawić napisaną w poprzednich fazach implementację
- ◆ w tym podejściu do testowania testy piszemy zanim zostanie napisana rzeczywista implementacja

START



RED PHASE



GREEN PHASE

refaktoryzacja
implementacji
i testowanie

pisanie
minimalnej
implementacji

TDD: Red

- ▶ czerwona faza to zawsze punkt wyjściowy od którego zaczynamy testowanie
- ▶ celem tej fazy jest napisanie testów, które informują nas o tym co powinna robić implementacja funkcjonalności (definiują cel funkcjonalności)
- ▶ na końcu tej fazy wszystkie testy powinny być czerwone (funkcjonalność jeszcze nie istnieje)
- ▶ testy będą przehodzić dopiero kiedy wymagania zostaną pokryte

TDD: Green

- ▶ w zielonej fazie piszemy kod implementacji, tak aby testy napisane w fazie czerwonej przeszły (stały się zielone)
- ▶ celem jest znalezienie rozwiązania, bez głębszego wchodzenia w optymalizację naszej implementacji
- ▶ na końcu tej fazy „jesteśmy na zielono”. Możemy zacząć myśleć o optymalizacjach naszego kodu

TDD: Refaktoryzacja

- ▶ w tej fazie nadal „**jesteśmy na zielono**”
- ▶ istotna faza to moment na zastanowienie się jak napisać nasz kod lepiej lub bardziej wydajnie
- ▶ podczas refaktoryzacji nie chodzi o zmiany wyniku naszej funkcji, a o osiągnięcie tego samego efektu przy bardziej opisowym lub szybszym kodzie

Pytania, które powinniśmy sobie zadać kiedy myślimy o refaktoryzacji:

- ▶ Czy mogę poprawić moje testy, aby były lepiej dopasowane?
- ▶ Czy moje testy zwracają wartościowy feedback na temat funkcjonalności?
- ▶ Czy moje testy są odpowiednio odizolowane od innych testów / modułów?
- ▶ Czy mogę zredukować duplikację kodu w moich testach lub w samej implementacji?
- ▶ Czy moja implementacja może być czytelniejsza?
- ▶ Czy moja implementacja może być bardziej wydajna?

white Box

- ◀ testy strukturalne
- ◀ testy bazują na kodzie źródłowym
- ◀ o wiele łatwiej przygotować kod do całkowitej optymalizacji dzięki testom
- ◀ łatwiejsze do automatyzacji

Black Box

- ▶ testy funkcjonalne
- ▶ nie mamy dostępu do kodu źródłowego
- ▶ testujemy tylko zakładane funkcjonalności
- ▶ w tego typu testach mamy większą szansę na znalezienie większej liczby błędów, ale niekonicznie ich pochodzenia

Grey Box

- ▶ testy hybrydowe, łączące Black Box i White Box
- ▶ tester ma ograniczoną wiedzę o kodzie źródłowym, np. zna architekturę systemu, ale nie ma pełnego dostępu do kodu
- ▶ umożliwia bardziej efektywne testowanie, gdyż pozwala na lepsze zrozumienie potencjalnych słabych punktów aplikacji
- ▶ często stosowane w testach bezpieczeństwa oraz testach integracyjnych

Antywzorce w testach

Obywatel drugiej kategorii

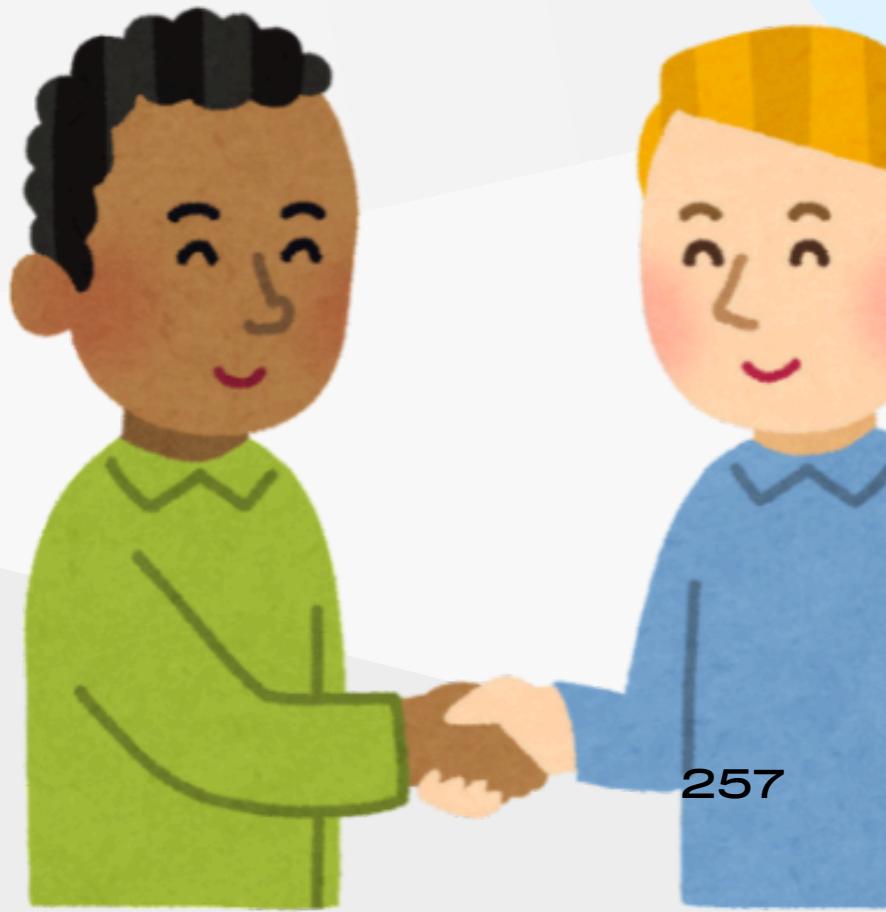
- ◀ kod testów traktowany jako ten gorszy
- ◀ nie jest utrzymywany, wszędzie mamy do czynienia z duplikacją kodu, chaosem i mylącymi nazwami



256

Brak szacunku

- ◀ deweloper robi jakieś zmianę i nie aktualizuje unit testów
- ◀ po wprowadzeniu system continuous integration zwraca błąd spowodowany nieprzechodzącymi testami
- ◀ zamiast poprawić testy usuwa się je, zakomentowuje albo ignoruje



Optymistyczna ścieżka

- ◀ testowanie tylko podstawowego działania funkcji bez zastanowienia się nad możliwymi wyjątkami, warunkami brzegowymi, czy złośliwymi kombinacjami danych wejściowych.

Gigant

- ▶ test zawierający bardzo dużo linii kodu – kilkaset, albo nawet ponad tysiąc
- ▶ test jest tak duży, że nie wiadomo do końca co robi, jego utrzymanie jest bardzo trudne i może on posiadać swoje własne bugi

Pasażer na gapę

- zamiast stworzyć nowy test, dodajemy kolejny cykl Arrange-Act-Assert do istniejącego testu
- w ten sposób istniejące testy się wydłużają, a ich cel się rozmywa
- w końcu taki test przeobraża się w giganta

Przykładowy kod:

```
test("Zbyt długi test – pasażer na gapę", () => {
  const user = createUser("Bob", 25);
  expect(user.name).toBe("Bob");

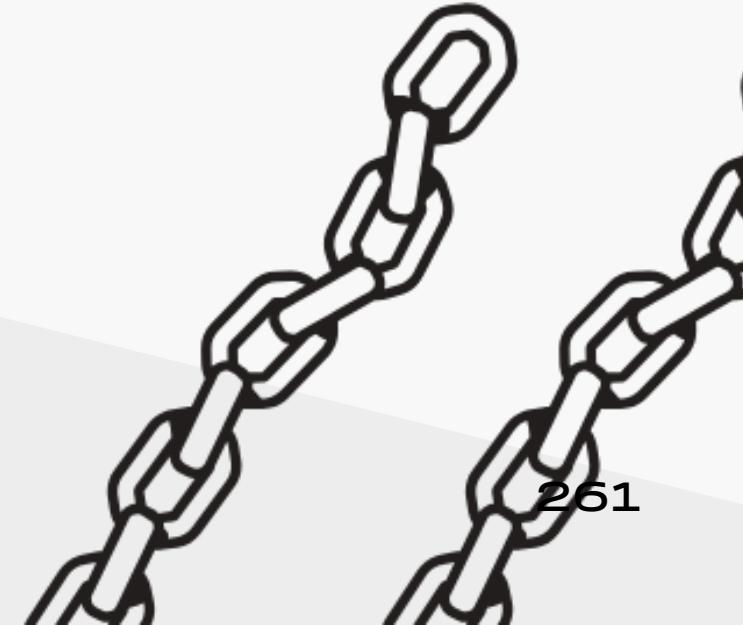
  user.updateAge(26);
  expect(user.age).toBe(26);

  user.deactivate();
  expect(user.isActive).toBe(false);
});
```



chain gang

- ◀ kolejne testy są od siebie zależne i muszą być wykonywane w odpowiedniej kolejności
- ◀ usunięcie lub zmiana jednego testu z łańcucha skutkuje wtedy zepsaniem całej grupy testów



Sobowtór

- ▶ na potrzeby testu mockujemy jakąś zależność, jednak aby wykonać test nie wystarczy nam podstawić zwracaną wartość - zamiast tego w mocku kopujemy zachowanie rzeczywistej funkcji
- ▶ jeżeli będziemy chcieli wprowadzić zmiany, musimy je uwzględnić zarówno w kodzie produkcyjnym, jak i w mocku



Kłamca

- ◀ test jest tak skonstruowany, żeby wchodził w odpowiednie ścieżki w kodzie i nabijał w ten sposób code coverage
- ◀ jednak nie sprawdza on w żaden sposób poprawności wykonywania tych ścieżek



Inspektor

- ◆ aby osiągnąć lepsze pokrycie, test bazuje na konkretnej implementacji i łamie zasady enkapsulacji
- ◆ w ten sposób otrzymujemy skomplikowane testy utrudniające refactor kodu w przyszłości



Flaky test

- ▶ test raz przechodzi, a raz nie – bez zmiany kodu
- ▶ może być zależny od kolejności uruchamiania, środowiska czy nawet czasu
- ▶ powoduje frustrację i zmniejsza zaufanie do testów

Przykładowy kod:

```
test("Flaky test - zależny od czasu", () => {
  const now = new Date().getSeconds();
  expect(now % 2).toBe(0); // Test przejdzie tylko w parzystych sekundach
});
```



Mr. Tester

- ▶ test, który testuje wszystko naraz
- ▶ zamiast jednego konkretnego przypadku sprawdza kilka rzeczy jednocześnie
- ▶ trudny do zrozumienia i poprawienia, bo nie wiadomo, co dokładnie się zepsuło

Przykładowy kod:

```
test("Sprawdza wszystko na raz", () => {
  const user = createUser("Alice", 30);

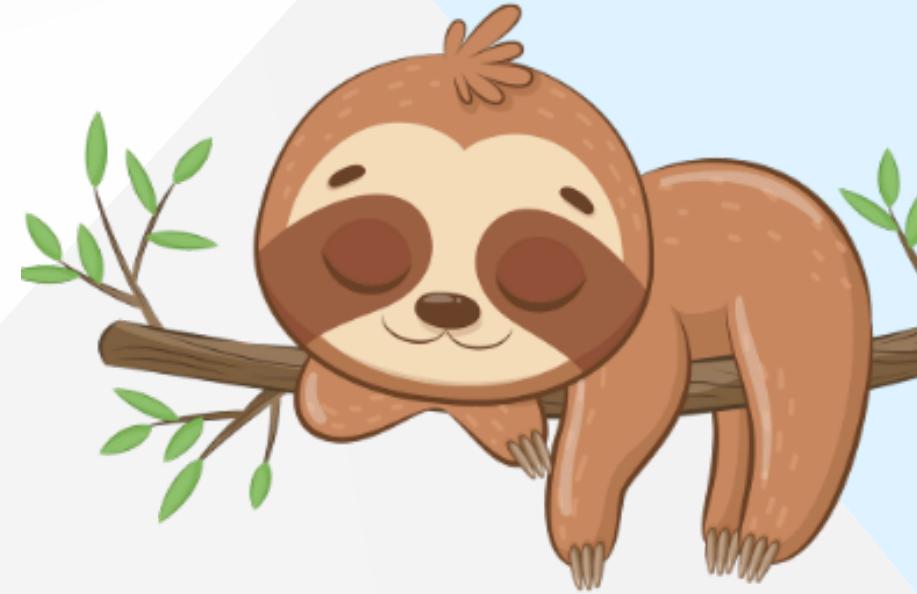
  expect(user.name).toBe("Alice");
  expect(user.age).toBe(30);
  expect(user.isActive).toBe(true);
  expect(user.permissions.length).toBeGreaterThan(0);
});
```

Śpiący test

- ◆ używa `sleep()` lub innych sztucznych opóźnień, zamiast prawidłowego oczekiwania na zakończenie operacji asynchronicznych
- ◆ spowalnia cały proces testowania
- ◆ czasami testy przechodzą, a czasami nie, zależnie od tego, jak wolno działa system

Przykładowy kod:

```
test("Sleepy test", (done) => {
  setTimeout(() => {
    expect(true).toBe(true);
    done();
  }, 5000); // Sztuczne czekanie 5 sekund!
});
```



Martwy kod testowy

- ▶ testy, których nikt nie uruchamia
- ▶ testy, które są zawsze wykomentowane lub ignorowane
- ▶ testy, które nie mogą się nie powieść (np. `assert(true)`)



Co powinniśmy testować na FE?

jednostkowo

- ◆ pojedyncze funkcje i komponenty w izolacji
- ◆ logikę biznesową (np. validację formularzy, obliczenia, konwersje)
- ◆ metody i efekty uboczne w komponentach (np. `useEffect`, eventy)

integracyjnie

- ◆ kluczowe ścieżki aplikacji (np. logowanie, wyszukiwanie, formularze)
- ◆ rzeczywiste interakcje - unikać mockowania wszystkiego
- ◆ komunikację z API - mockując zapytania HTTP

e2e

- ◆ kluczowe ścieżki użytkownika
- ◆ rzeczywiste interakcje w przeglądarce

Domyślne pliki testów

- ❖ `.js` w katalogu `__tests__`
- ❖ `.test.js`
- ❖ `.spec.js`

Jest

- ◆ Javascript Testing Framework
- ◆ zezwala na dostęp do struktury DOM za pomocą biblioteki `jsdom`
- ◆ `jsdom` jest tylko symulacją przeglądarki - pokrywa większość przypadków, w którymi napotkamy się testując aplikacje frontendowe
- ◆ w Jest możemy mockować moduły (takie jak serwisy) czy timery

Jest: Matchers

Jest udostępnia **matchers**, aby ułatwić porównywanie wartości na wiele różnych sposobów.

Matcher	Opis
toBe	używa Object.is do testowania dopasowania ścisłego
toEqual	rekursively sprawdza każde pole w obiekcie lub tablicy
not	
toBeNull, toBeUndefined, toBeDefined, toBeTruthy, toBeFalsy	dla sprawdzenia prawdziwości

Matcher	Opis
toBeGreaterThan, toBeLessThan, toBeGreaterThanOrEqual, toBeLessThanOrEqual, toBeCloseTo	dla liczb
toMatch	dla stringów
toContain	dla tablic
toThrow	dla wyjątków

Przykładowy kod:

```
test('two plus two is four', () => {
  expect(2 + 2).toBe(4);
});

test('object assignment', () => {
  const data = {one: 1};
  data['two'] = 2;
  expect(data).toEqual({one: 1, two: 2});
});

test('adding positive numbers is not zero', () => {
  for (let a = 1; a < 10; a++) {
    for (let b = 1; b < 10; b++) {
      expect(a + b).not.toBe(0);
    }
  }
});
```

Async

- ▶ kod JavaScript uruchamiany jest zazwyczaj asynchronicznie (zdarzenia, zapytania itp)
- ▶ zwrócenie obietnicy z testu spowoduje, że Jest będzie oczekiwał aż obietnica zostanie rozwiązana
- ▶ Jest posiada również matchery do kodu asynchronicznego: `resolves` i `rejects`

Przykładowy kod:

```
test('the data is peanut butter', () => {
  return fetchData().then(data => {
    expect(data).toBe('peanut butter');
  });
});

test('the data is peanut butter', async () => {
  const data = await fetchData();
  expect(data).toBe('peanut butter');
});

test('the fetch fails with an error', async () => {
  expect.assertions(1);
  try {
    await fetchData();
  } catch (e) {
    expect(e).toMatch('error');
  }
});

test('the data is peanut butter', async () => {
  await expect(fetchData()).resolves.toBe('peanut butter');
});
```

Przykładowy kod:

```
import {describe, expect, test} from '@jest/globals';
import {sum} from './sum';

describe('sum module', () => {

  describe('when sth happen', () => {

    test('adds 1 + 2 to equal 3', () => {
      expect(sum(1, 2)).toBe(3);
    });
  });
});
```

Setup testów

- ↳ jeśli musimy wykonać operacje wielokrotnie przed i po uruchomieniu jakiegoś testu, możemy skorzystać z Hooków: `beforeEach` i `afterEach`.
- ↳ w niektórych przypadkach potrzebujesz wykonać jakąś operację raz przed / po wszystkimi testami z danej grupy, dla takich przypadków Jest udostępnia hooki: `beforeAll` i `afterAll`
- ↳ jeśli musimy uruchomić tylko jeden test (ponieważ nam nie przechodzi), tymczasowo możemy zastąpić funkcję `test` za pomocą funkcji `test.only`

Przykładowy kod:

```
beforeEach(() => {
  return initializeCityDatabase();
});

test('city database has Vienna', () => {
  expect(isCity('Vienna')).toBeTruthy();
});
```

Przykładowy kod:

```
test.only('this will be the only test that runs', () => {
  expect(true).toBe(false);
});

test('this test will not run', () => {
  expect('A').toBe('A');
});
```

Mockowanie funkcji

- ◀ mockowanie funkcji zezwala na testowanie połączeń pomiędzy różnymi częściami kodu za pomocą usunięcia aktualnej implementacji funkcji, przechwyceniem jej uruchomienia (wraz z parametrami), przychwytcie instancji konstruktora funkcji, kiedy zostały zainicjowane przez operator `new`
- ◀ pozwala ponadto na zwrócenie własnej wartości z mockowanej funkcji
- ◀ wszystkie mockowane funkcje mają specjalną właściwość `mock`, gdzie możemy znaleźć informacje na temat ilości i sposobu wywołania danej funkcji
- ◀ możemy wykorzystać mockowanie implementacji całych modułów
- ◀ po zakończeniu testu warto wywołać `mockRestore()` na mocku, aby zapobiec przeciekaniu mocków pomiędzy testami

Przykładowy kod:

```
function forEach(items, callback) {  
  for (let index = 0; index < items.length; index++) {  
    callback(items[index]);  
  }  
}  
const mockCallback = jest.fn(x => 42 + x);  
forEach([0, 1], mockCallback);  
// The mock function is called twice  
expect(mockCallback.mock.calls.length).toBe(2);
```

Przykładowy kod:

```
import axios from 'axios';
import Users from './users';

jest.mock('axios');

test('should fetch users', () => {
  const users = [{name: 'Bob'}];
  const resp = {data: users};
  axios.get.mockResolvedValue(resp);

  // or you could use the following depending on your use case:
  // axios.get.mockImplementation(() => Promise.resolve(resp))

  return Users.all().then(data => expect(data).toEqual(users));
});
```

Timers

- natywne funkcje timerów (np. `setTimeout`, `setInterval`, `clearTimeout`, `clearInterval`) są mniej niż idealne dla testowania na środowiskach testerskich, ponieważ są zależne od prawdziwego upływu czasu
- w naszym teście możemy włączyć `fake timers` poprzez wywołanie `jest.useFakeTimers()`

Przykładowy kod:

```
jest.useFakeTimers();
jest.spyOn(global, 'setTimeout');

test('waits 1 second before ending the game', () => {
  const timerGame = require('../timerGame');
  timerGame();

  expect(setTimeout).toHaveBeenCalledWith(expect.any(Function), 1000);
  expect(setTimeout).toHaveBeenCalledTimes(1);
});
```

Optymalizacja testów z Jest

Testy równoległe

Domyślnie Jest uruchamia testy równolegle, co może znacząco skrócić czas trwania testów, szczególnie w przypadku dużych projektów. Możesz kontrolować liczbę równocześnie uruchamianych wątków przy pomocy opcji `--maxWorkers`. Domyślnie Jest ustala tę liczbę na liczbę rdzeni CPU.

Przykładowy kod:

```
jest --maxWorkers=4
```

Testowanie tylko zmienionych plików

Używając flagi `--onlyChanged` lub `--watch`, można uruchomić testy tylko dla zmodyfikowanych plików. To przydatne w dużych projektach, gdzie codziennie testujesz tylko część aplikacji.

Przykładowy kod:

```
jest --onlyChanged
```

React Testing Library

- ▶ jest to biblioteka która pomaga nam pisać scenariusze testowe
- ▶ zawiera zestaw funkcji pomocniczych dla testowania komponentów React bez szczegółów implementacyjnych
- ▶ nie zezwala na renderowanie komponentów w sposób płytka (shallow) - czyli bez potomków
- ▶ React Testing Library nie jest alternatywą dla Jest'a, obie biblioteki potrzebują siebie nawzajem i każda z nich ma inne zadanie do wykonania
- ▶ RTL jest zamiennikiem dla Enzyme
- ▶ React Testing Library nie jest specyficzne dla żadnego framework'a do testowania
- ▶ RTL jest zbudowany na DOM Testing Library

Przykładowy kod:

```
import {render, screen} from '@testing-library/react'
import userEvent from '@testing-library/user-event'
import '@testing-library/jest-dom'
import Fetch from './fetch'

test('loads and displays greeting', async () => {
  // ARRANGE
  render(<Fetch url="/greeting" />

  // ACT
  await userEvent.click(screen.getByText('Load Greeting'))
  await screen.findByRole('heading')

  // ASSERT
  expect(screen.getByRole('heading')).toHaveTextContent('hello there')
  expect(screen.getByRole('button')).toBeDisabled()
})
```

	Brak dopasowania	1 dopasowanie	Więcej dopasowań	Await?
getBy	throw	return	throw	brak
findBy	throw	return	throw	tak
queryBy	null	return	throw	brak
getAllBy	throw	array	array	brak
findAllBy	throw	array	array	tak
queryAllBy	[]	array	array	brak

25 minut

zadanie nr 23

- ▼ wykorzystując RTL przetestuj jeden wybrany komponent oraz jedną funkcję z utils/



287

Dodatkowe pytania?





KAHoot

Dołącz na kahoot.it
lub przez aplikację [Kahoot!](#)

XXX

Ankieta

czyli jak mi poszło?

<https://sages.link/049925>





Dziękuję za uwagę

Mateusz Jabłoński

mail@mateuszjablonski.com

mateuszjablonski.com

STACJA.IT

JABŁOŃSKI

sages