Mateusz Jabłoński

Wprowadzenie do React

10.05.2024

sages

Kim jestem?

Mateusz Jabłoński

mateuszjablonski.com
mail@mateuszjablonski.com

Od początku

Ustalenia

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

Co nas czeka?

- Javascript - powtórzenie elementów istotnych w kontekście biblioteki React
- Rodzaje aplikacji frontendowych
- Architektura komponentowa
- Podstawowe założenia biblioteki React
- Idea Virtual DOM
- Komponenty funkcyjne i klasowe
- JSX
- Algorytm różnicujący i rekoncyliacja
- Właściwości komponentu, czyli props
- useEffect oraz useState
- Zarządzanie stanem komponentu

github.com/matwjablonski/react-stacjait-100524

Rozkład warsztatów

- **17:00 - start**
- 18:00 - 18:15 - przerwa
- 19:45 - 20:00 - przerwa
- **21:00 - koniec**

Projekt

- aplikacja z ciekawymi miejscami do zobaczenia

Javascript

Zmienne i funkcje

Sposoby deklarowania zmiennych

- **var**
  - najstarszy sposób deklarowania zmiennych
  - działa w oparciu o zakres globalny i funkcyjny
  - zezwala na swobodne modyfikowanie wartości bez względu na jej typ
  - po deklaracji w zasięgu globalnym nie można jej usunąć z obiektu globalnego za pomocą **delete**
- **let**
  - wprowadzony w ES2015
  - działa w oparciu o zakres globalny i blokowy
  - zezwala na swobodne modyfikowanie wartości bez względu na jej typ
  - po deklaracji w zasięgu globalnym nie tworzy odrębnej właściwości w obiekcie globalnym
- **const**
  - wprowadzony w ES2015
  - działa w oparciu o zakres globalny i blokowy
  - nie zezwala na modyfikowanie wartości prymitywnych
  - wartość musi zostać przypisana podczas deklaracji
  - pozwala mutować obiekty złożone
  - po deklaracji w zasięgu globalnym nie tworzy odrębnej właściwości w obiekcie globalnym

Arrow functions

- uproszczona wersja tradycyjnej funkcji
- nie powinna być wykorzystywana jako metoda w klasach
- nie zmienia kontekstu **this**
- nie mogą być używane z metodami zmiany kontekstu (**call**, **bind**, **apply**)
- nie ma dostępu do obiektu **arguments**
- nie mogą być używane jako metody konstruktora
- nie mają pełnego wsparcia dla funkcji generatorów

Javascript!

Funkcje tablicowe

```
const data = [
  {
    name: 'Adas',
    age: 26,
  },
  {
    name: 'Monika',
    age: 22,
  },
  {
    name: 'Stanislaw',
    age: 62,
  },
];

data
  .filter((person) => person.age > 25)
  .map((person) => ({
    ...person,
    salary: person.age * 100,
  }))
  .reduce((acc, { salary }) => acc + salary, 0);
```

Javascript

Transpilacja kodu

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

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 polyfiili aby zezwolić na używanie funkcji, które nie są dostępne w standardzie ES5

Babel w React

- Babel jest jednym z podstawowych narzędzi wykorzystywanych w procesie development aplikacji React’owych
- Konfiguracja jest trzymana w pliku **.babelrc**
- Konfiguracja jest podzielona na dwie grupy:
  - presets - reguły transformacji kodu
  - plugins - rozszerzenia dla silnika Babel’a
-

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

React

Podstawy


React

SPA

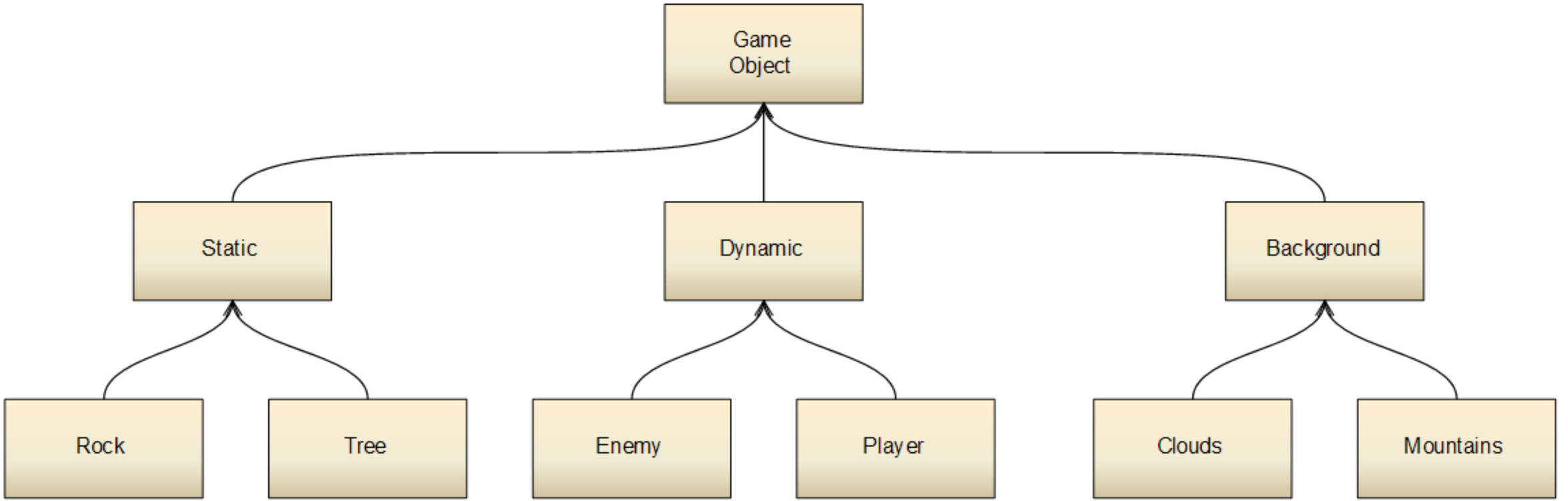
- SPA - Single Page Applications
- SPA jest alternatywą dla MPA (Multi Page Applications)
- Serwer zwraca prosty plik HTML, który zawiera tylko jednego diva i informacje o wymaganych stylach i skryptach
- Odpowiedź z serwera jest bardzo szybka
- Cała logika aplikacji i nawigacji jest przeniesiona na 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)
- Kosz infrastruktury został przeniesiony na klienta końcowego

React

- React jest biblioteką która może być wykorzystana do budowania dynamicznych i złożonych interfejsów użytkownika w sposób deklaratywny i modułowy
- Zdejmuje odpowiedzialność za renderowanie i aktualizację stanu drzewa dom z programisty
- Zezwala na tworzenie struktury widoku i logiki wyświetlania treści w sposób bardziej deklaratywny (bardziej naturalny niż w przypadku programowania zorientowanego obiektowo czy imperatywnego)
- Zezwala na trzymanie struktury i logiki w jednym miejscu (**tylko z wykorzystaniem Javascript** - programista może wykorzystywać wszystkie możliwości języka Javascript, bez konieczności dotykania składni języków strukturalnych)
- React buduje widok w architekturze komponentowej
    - Komponent to element oprogramowania posiadający dobrze wyspecyfikowany interfejs oraz zachowanie

Architektura komponentowa

- Komponenty mogą być wykorzystane w wielu aplikacjach
- Skraca czas budowy nowych aplikacji i obniża koszty projektu - poprzez zastosowanie 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.


Game Object
Static
Dynamic
Background
Rock
Tree
Enemy
Player
Clouds
Mountains

React

- **react** - podstawowa biblioteka obsługująca model kompozycyjny React’a
- **react-dom** - biblioteka udostępniająca specyficzne dla DOM metody
  - **react-dom/client** - moduł, który zezwala na tworzenie i używanie aplikacji React po stronie klienta
  - **react-dom/server** - moduł, który zezwala na budowanie komponentów React w wersji serwerowej

!

React nie renderuje zmian bezpośrednio, ale poprzez Virtual-DOM

```
<!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>
```

React

Virtual DOM

Virtual DOM

- 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

!

Dzięki małej liczbie operacji na strukturze DOM,
React aktualizuje widok błyskawicznie!

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.

React

Atrybuty i klasy

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

React

JSX

```
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'));
```

```
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'));
```

```
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>;

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

React

Komponenty

Komponenty

- 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 Reac'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**)

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

```
// 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>
  }
});
```

```
// ES6 class

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

```tsx
// TS class

interface GreetingsProps {
  name: string;
}

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

```
// 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.

```tsx
// 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>
}
```

React

Algorytm różnicujący

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 z instancjami 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 następnym 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-renderowaniami instancja komponentu pozostaje niezmieniona
- Kiedy w komponencie wykryto zmianę:
  - Komponent zostanie poinformowany o potrzebie wykonania re-renderowania.

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.

React

Create React App

```
{
  "name": "test2",
  "version": "0.1.0",
  "private": true,
  "dependencies": {
    "@testing-library/jest-dom": "^5.16.5",
    "@testing-library/react": "^13.4.0",
    "@testing-library/user-event": "^13.5.0",
    "react": "^18.2.0",
    "react-dom": "^18.2.0",
    "react-scripts": "5.0.1",
    "web-vitals": "^2.1.4"
  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },
  "eslintConfig": {
    "extends": [
      "react-app",
      "react-app/jest"
    ]
  },
  "browserslist": {
    "production": [
      ">0.2%",
      "not dead",
      "not op_mini all"
    ],
    "development": [
      "last 1 chrome version",
      "last 1 firefox version",
      "last 1 safari version"
    ]
  }
}
```

React

Stan i propsy

Stan i props’y

- 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**)
  - 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)

```
import React from 'react';

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

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

```
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” />)
  }
}
```

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

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

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

```
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() }
  }
}
```

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.

- **Podejście klasowe:**
  - 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

- **Podejście funkcyjne (hook):**
  - Zmiany stanu nadpisują poprzedni stan (w przeciwieństwie do komponentów klasowych)

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

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

```
class Comp extends React.Component {

}

Comp.propTypes = {
  name: PropTypes.string,
}
```

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

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

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

```
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} />
    )
  }
}
```

Zagnieżdżanie komponentów

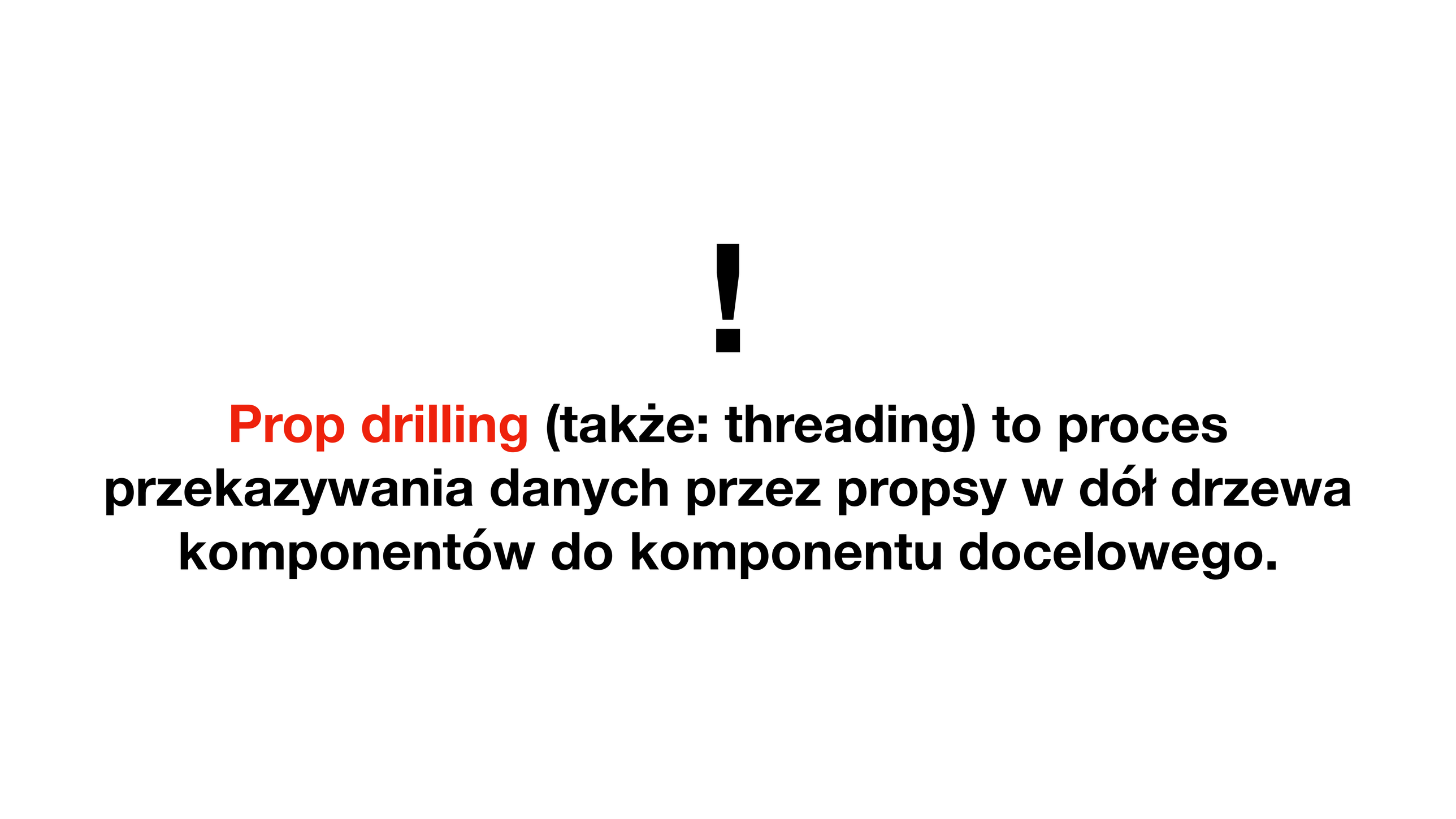
- Komponenty mogą być zagnieżdżane
- Każdy kod JSX, który jest przekazany pomiędzy znacznikiem otwarcia i zamknięcia komponentu, będzie dostępny w tym komponencie jako props **children**
- Jako dzieci komponentu możemy zagnieżdżać stringi, tablice elementów JSX i wszystko co wypełnia typ **ReactNode**

```jsx
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;
```


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

React

Hooks

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 są funkcjami, które mogą być umieszczane wewnątrz komponentu, ale tylko na najwyższym poziomie (top scope)
- Hooki dodają do komponentów funkcyjnych możliwości używania referencji, obsługi stanu i wywoływania efektów ubocznych
- Hooki mogą być wywoływane tylko w komponentach Reactowych lub w innych własnych hookach
- Możemy tworzyć własne hooki (nazwa funkcji hook’a powinna rozpoczynać się od słowa kluczowego **use**)

!

Kolejność wywoływania Hooków jest ważna!

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

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

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

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 - pierwsza wartość do zmienna przechowująca bieżący stan, druga to funkcja zmieniająca stan).
- Domyślny stan może być również funkcją. Wówczas podczas tworzenia domyślnego stanu domyślna wartość będzie wartością zwróconą z wywołania tej funkcji.

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

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

```
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()

```
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

- 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!)**

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

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

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

Inne hooki

- useContext
- useRef
- useReducer
- useLayoutEffect
- **useMemo** - memoizacja wartości
- **useCallback** - memoizacja funkcji
- **useImperativeHandle** - dostosowuje wartość instancji, która została przykazana do rodzica komponent podczas użycia referencji, zastosowanie podobne do useRef, ale daje większą kontrolę nad wartością która jest zwracana i zezwala na zastąpienie natywnych funkcji takich jak: blur, focus własnymi ich implementacjami
- **useDebugValue** - może być użyty do wyświetlenia etykiet dla własnych Hooków w React DevTools
- **useId** - generuje unikalne ID, które jest stabilne pomiędzy każdym renderem, przydatne gdy musimy uspójnić komponent renderowany po stronie klienta i po stronie serwera
- **useTransition** - pozwala nam ustawić priorytet na zmianie stanu, tak aby wykonała się w pierwszej kolejności
- **useDefferedValue** - pozwala na pokazanie starej wartości, dopóki nowa nie będzie gotowa (przydatne gdy widok jest zależny od wartości pochodzących spoza komponentu, tj. biblioteki, inne komponenty)

React

Kompozycja

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}
```

So What About Inheritance? At Facebook, we use React in thousands of components, and we haven't found any use cases where we would recommend creating component inheritance hierarchies.

React

Strict Mode

<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)

Narzędzia

- **ESLint** - narzędzie do analizy statycznej kodu
- **Prettier**, **Editorconfig** - narzędzia automatyzujące poprawianie kodu stylistycznie podczas zapisu oraz commitowania
- **Visual Studio Code**, **Webstorm** - IDE
- **Live Share** dla VSC - narzędzie do wspólnego pisania kodu
- **Snyk** - narzędzia do analizy podatności w zależnościach w projekcie
- **Playgroundy**:
  - https://www.typescriptlang.org/play
  - https://www.sassmeister.com/
  - https://jsbin.com/?html,output
  - https://jsconsole.com/
  - https://rxmarbles.com/

Dodatkowe pytania?

Ankieta:

bit.ly/4aa0FWO

mail@mateuszjablonski.com
mateuszjablonski.com
linkedin.com/in/mateusz-jablonski/

Dziękuję za uwagę

STACJA.IT