Wybrane elementy praktyki projektowania oprogramowania
Wykład 14/15
React

Wiktor Zychla 2023/2024

1 Spis treści

2 Wprowadzenie 2
3 Tworzenie aplikacji React 3
4 Stan komponentu 4
5 Cykl życia komponentu 5
6 Komunikacja między komponentami 6
7 Komponenty kontrolowane i niekontrolowane 9
7.1 Przykład komponentu niekontrolowanego 10
7.2 Przykład komponentu kontrolowanego 11
8 Złożony stan komponentu 12
9 Komunikacja z serwerem 14
10 Klient i serwer w jednym projekcie 14

2 Wprowadzenie

React to biblioteka znacznie ułatwiająca tworzenie złożonych interfejsów po stronie klienta. Upubliczniona w 2013, a powstała jako wewnętrzne narzędzie w Facebook (obecnie Meta).

W 2015 roku React otrzymuje wersję React-Native przeznaczoną do tworzenia aplikacji natywnych na urządzenia mobilne.

Tworzenie aplikacji w React polega na dzieleniu interfejsu na tzw. komponenty czyli mniejsze części, zarządzające stanem (danymi), komunikujące się z innymi komponentami. Początkowo komponenty pisane były jako klasy JavaScript, zanurzone w hierarchii obiektowej biblioteki.

```
class HelloWorld extends React.Component {
  render() {
    return (
      <div>Hello world />
    );
  }
}
```

W 2019 roku w wersji 16.8 React, ostatecznie dodano szereg funkcjonalności umożliwiających tworzenie tzw. komponentów funkcyjnych.

```
const HelloWorld = () => <div>Hello World</div>;
```

Programowanie przy użyciu obu podejść jest równoważne, zwykle przyjmuje się że styl funkcyjny jest bardziej zwięzły, stąd współcześnie jest wybierany chętniej.

Główne cechy React:

- JSX – rozszerzenie JavaScript/TypeScript umożliwiające osadzanie w kodzie imperatywnym fragmentów deklaratywnych, opisujących widoki. W przypadku JavaScript wymaga zewnętrznego transpilera (np. Babel), w przypadku TypeScript jest wspierane wprost przez kompilator (i silnie typowane!)
- Virtual DOM – modyfikacje widoku nie modyfikują bezpośrednio drzewa DOM w przeglądarce. Zamiast tego React utrzymuje w pamięci kopię struktury DOM, a następnie używa algorytmu tzw. rekoncyliacji do efektywnego wyznaczenia tych fragmentów DOM które należy faktycznie modyfikować
- Wsparcie dla TypeScript

3 Tworzenie aplikacji React

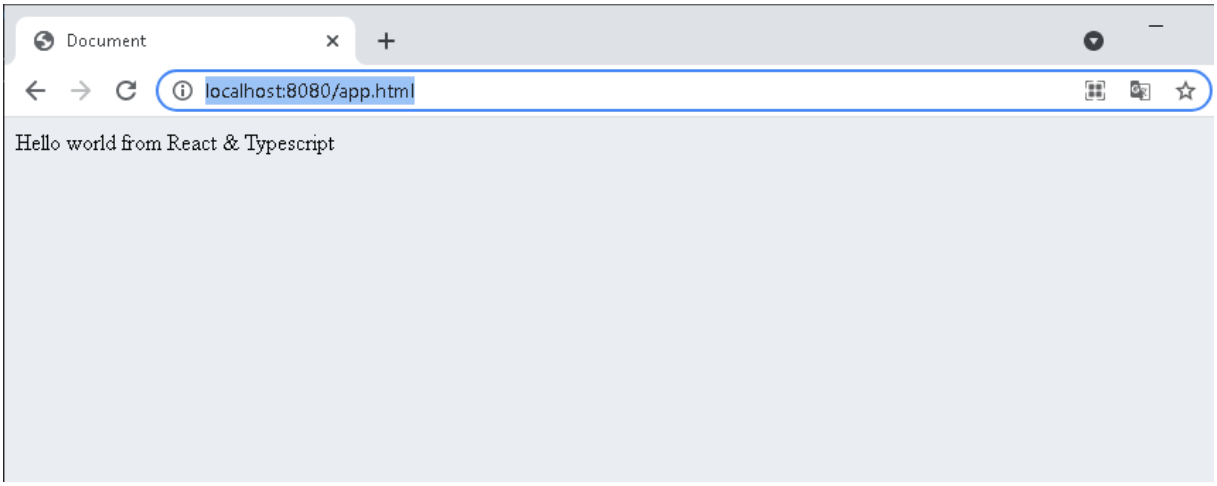
Dokumentacja React sugeruje użycie skryptu **create-react-app**, który należy zainstalować z repozytorium npm.

Można wypróbować ten sposób. Jego zaleta to m.in. integracja serwera developerskiego, który startuje rozwijaną aplikację w przeglądarce. Jego wada to ogromna liczba zależności i przytłaczająca struktura aplikacji, w której nawet konfiguracja poszczególnych elementów jest ukryta. Na słabszej maszynie tworzenie pustej aplikacji w ten sposób może trwać nawet kilkanaście minut!

Można również zacząć od zera i dodać tylko wymagane komponenty, tak aby uzyskać minimalną, działającą aplikację React, z minimalną liczbą zależności i prostymi plikami konfiguracyjnymi. Wymagane będą:

- **webpack** – zainstalowany globalne przez **npm install webpack -g**
- **typescript** – zainstalowany globalnie przez **npm install typescript -g**

Tutorial który wyjaśnia poszczególne kroki znajduje się tu. Należy zwrócić uwagę na to że otrzymana aplikacja jest w pełni debuggowalna z poziomu VS Code.



4 Stan komponentu

Jedną z najważniejszych zalet komponentu jest możliwość przetrzymywania stanu, do zdefiniowania którego użyjemy hooka **useState**. To uproszczony sposób zarządzania stanem, jego uogólnieniem jest **useReducer**, które warto poznać z uwagi na to że podejście wykorzystujące reduktory jest wykorzystywane w bibliotekach do zarządzania stanem (np. Redux).

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

const App = () => {

    const [counter, setCounter] = useState(0);

    const decrementClick = () => {
      setCounter( counter - 1 );
    }

    const incrementClick = () => {
      setCounter( counter + 1 );
    }

    return <div>
      <div>
        Licznik: {counter}
      </div>
      <div>
        <button onClick={decrementClick}>-</button><button
onClick={incrementClick}>+</button>
      </div>
    </div>
  };

export default App;
```

5 Cykl życia komponentu

Najprostsze zarządzane cyklem życia komponentu polega na użyciu hooka **useEffect** dla którego określa się listę zależności (zmiennych stanu), których zmiana powoduje kolejne renderowanie komponentu i wykonanie funkcji przekazanej do hooka.

Dla pustej listy zależności osiąga się efekt wykonania fragmentu kodu przy pierwszym (i tylko pierwszym) renderowaniu. Z kolei funkcja zwracana z wywołania hooka jest wywoływana przy tzw. odmontowywaniu komponentu (czyli wtedy kiedy jest on usuwany z drzewa DOM).

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

const App = () => {

  const [date, setDate] = useState<Date>();

  useEffect(() => {

    const interval = setInterval(() => {
      setDate(new Date());
    }, 1000);

    return () => {
      clearInterval(interval);
    };
  }, []);

  return <>
    <div>bieżąca data {date && date.toISOString()}</div>
  </>
};

export default App;
```

6 Komunikacja między komponentami

Komponent-rodzic przekazuje dane swoim komponentom potomnym za pomocą tzw. właściwości (*props*).

W poniższym przykładzie komponent-rodzic (**App**) przekazuje stan komponentowi potomnemu (**Child**). Każda zmiana w rodzicu wartości zmiennej która jest przekazywana do potomka we właściwości powoduje ponowne renderowanie potomka.

```
// app.tsx
import React, { useState } from 'react';
import Child from './Child';

const App = () => {

    const [counter, setCounter] = useState(0);

    const decrementClick = () => {
      setCounter( counter - 1 );
    }

    const incrementClick = () => {
      setCounter( counter + 1 );
    }

    return <div>
      <Child counter={counter} />
      <div>
        <button onClick={decrementClick}>-</button><button
onClick={incrementClick}>+</button>
      </div>
    </div>
  };

export default App;

// Child.tsx
import React from 'react';

type ChildPropsType = {
    counter: number;
}

const Child = (props: ChildPropsType) => {
    return <div>
        {props.counter}
    </div>
```

```
}

export default Child;
```

Komponent potomny przekazuje dane swojemu rodzicowi za pomocą funkcji zwrotnej. W poniższym przykładzie do komponentu potomnego (**Child**) przeniesiono dodatkowo elementy interfejsu odpowiedzialne za renderowanie przycisków do zmiany stanu, ale sama zmiana stanu zachodzi nadal w komponencie-rodzicu (**App**). Do komunikacji zwrotnej użyte są dwie funkcje, przekazane we właściwościach **decrementHandler** i **incrementHandler.** Sygnatury przekazywanych funkcji mogą być dowolne, byle tylko były zgodne po obu stronach (komponentu przekazującego i komponentu odbierającego).

```
// app.tsx
import React, { useState } from 'react';
import Child from './Child';

const App = () => {

  const [counter, setCounter] = useState(0);

  const decrementClick = () => {
    setCounter( counter - 1 );
  }

  const incrementClick = () => {
    setCounter( counter + 1 );
  }

  return <div>
    <Child counter={counter} decrementHandler={decrementClick}
incrementHandler={incrementClick} />
  </div>
};

export default App;

// child.tsx
import React from 'react';

type ChildPropsType = {
    counter: number;
    incrementHandler: () => void;
    decrementHandler: () => void;
}

const Child = (props: ChildPropsType) => {
```

```
    const decrementClick = () => {
        props.decrementHandler();
    }

    const incrementClick = () => {
        props.incrementHandler();
    }

    return <div>
        {props.counter}
        <button onClick={decrementClick}>-</button><button
onClick={incrementClick}>+</button>
    </div>
}

export default Child;
```

7 Komponenty kontrolowane i niekontrolowane

W przypadku komponentów zawierających elementy interfejsu użytkownika pozwalające na edycję danych, należy zdecydować się na jeden z dwóch modeli architektury.

Komponent niekontrolowany to taki, który dane wprowadzane przez użytkownika obsługuje wyłącznie w DOM (skąd można je odczytać kiedy jest to potrzebne, na przykład do przesłania stanu komponentów na serwer)

Komponent kontrolowany to taki, który dane wprowadzane przez użytkownika odwzorowuje w stanie komponentu (a bieżący stan jest używany do renderowania DOM)

Komponenty niekontrolowane są łatwiejsze do programowania, ale mają szereg ograniczeń, wynikających z braku bezpośredniego dostępu do stanu, m.in.

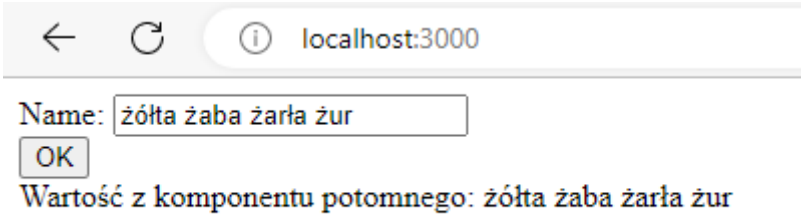
- Trudniejsza walidacja
- Trudniejsze programowanie dynamicznych zależności (np. przycisk nieaktywny dopóki w polu tekstowym nie wprowadzono wartości)
- Trudniejsze programowanie zaawansowanych scenariuszy (np. więcej niż jeden formant dla jednej edytowanej wartości)
- Trudniejsze testowanie

Komponent-rodzic w obu poniższych przykładach będzie ten sam:

```
import React, { useState } from 'react';
import Child from './Child';

const App = () => {

  const [name, setName] = useState('');

  /**
   * Komponent potomny przekaże wartość zwrotnie przez tę funkcję
   */
  const onNameEdited = (name: string) => {
    setName(name);
  }

  return <div>
    <div>
      <Child onNameEdited={onNameEdited}  />
    </div>
    <div>
      Wartość z komponentu potomnego: {name}
    </div>
  </div>
};

export default App;
```

7.1 Przykład komponentu niekontrolowanego

Komponent niekontrolowany używa hooka **useRef** do utworzenia trwałej referencji i przypięcia jej do elementu drzewa DOM. Odczytanie wartości polega na bezpośrednim odwołaniu do referencji (**refInput.current.value**).




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

type ChildPropsType = {
    onNameEdited: (name: string) => void;
}

/**
 * Komponent niekontrolowany
 */
const Child = (props: ChildPropsType) => {

    const refInput = useRef<HTMLInputElement>();

    const onClickHandler = () => {
        props.onNameEdited( refInput.current.value );
    }

    return <div>
        <div>
            Name: <input type='text' ref={refInput} />
        </div>
        <div>
            <button onClick={onClickHandler}>OK</button>
        </div>
    </div>
}

export default Child;
```

7.2 Przykład komponentu kontrolowanego

Komponent kontrolowany w pełni panuje nad stanem – to stan jest źródłem wiedzy dla komponentów interfejsu. Każda zmiana wartości w komponencie musi być odwzorowana w stanie.

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

type ChildPropsType = {
    onNameEdited: (name: string) => void;
}

/**
 * Komponent kontrolowany
 */
const Child = (props: ChildPropsType) => {

    // stan jest źródłem wiedzy dla interfejsu
    const [name, setName] = useState('');

    // ale każdą zmianę trzeba odczytać i odwzorować w stanie
    const nameOnChange = (e: React.ChangeEvent<HTMLInputElement>) => {
        setName(e.target.value);
    }

    const onClickHandler = () => {
        props.onNameEdited( name );
    }

    return <div>
        <div>
            Name: <input type='text' value={name} onChange={nameOnChange} />
        </div>
        <div>
            <button onClick={onClickHandler}>OK</button>
        </div>
    </div>
}

export default Child;
```

8 Złożony stan komponentu

Stan komponentu bywa złożony – odwzorowuje np. wiele formantów zbierających dane itp. W takich przypadkach, zamiast **useState** można wesprzeć się hookiem **useReducer** który pozwala na pisanie logiki tzw. redukcji stanu (czyli zmiany stanu).

Logika wyrażona jest za pomocą funkcji tzw. reduktora. Reduktor to funkcja która ma prostą sygnaturę

(stan, akcja) => nowyStan

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

type ComponentState = {
  name?: string,
  surname?: string,
  isLoading: boolean
}

type ComponentAction = ComponentLoadAction | ComponentUpdateAction;

type ComponentLoadAction = {
  type: 'LOAD_DATA'
}

type ComponentUpdateAction = {
  type: 'UPDATE_DATA',
  name: string,
  surname: string
}

const componentReducer =
  (state: ComponentState, action: ComponentAction): ComponentState => {
  switch ( action.type ) {
    case "LOAD_DATA" :
      return { isLoading: true };
    case "UPDATE_DATA" :
      return { ...action, isLoading: false }
  }
}

const App = () => {

  const [state, dispatch] = useReducer(componentReducer, { isLoading: false
});

  useEffect( () => {

    dispatch( { type: 'LOAD_DATA' } );
```

```
    setTimeout( () => {
      dispatch( { type: 'UPDATE_DATA', name: 'Jan', surname: 'Kowalski' } );
    }, 2000 );

  }, [] );

  return <>
    { state.isLoading &&
      <div>
        trwa ładowanie danych
      </div>
    }
    { !state.isLoading &&
      <div>
        <div>Name: {state.name}</div>
        <div>Surname: {state.surname}</div>
      </div>
    }

  </>
};

export default App;
```

9 Komunikacja z serwerem

Komunikacja z serwerem w React wymaga zewnętrznych funkcji, można użyć wbudowanego w przeglądarkę **fetch** albo którejś z bibliotek dodających pewne ułatwienia, np. **axios**.

Logika pobierania danych w prostym scenariuszu może znaleźć się wewnątrz funkcji przekazanej do **useEffect**. W ten sposób komponent pobierze dane podczas montowania (pierwszego renderowania). W przypadku takiego podejścia należy zwracać uwagę na problem pojawiający się w sytuacji w której komponent zostanie usunięty z DOM zanim zakończy się żądanie do serwera. W takich przypadkach należy pamiętać o przerywaniu żądań do serwera za pomocą obiektu **AbortController**.

10 Klient i serwer w jednym projekcie

Dla wygody zarówno część serwerowa (usługi API) jak i aplikacja React mogą znajdować się w jednym folderze. Aplikacja kliencka w takim scenariuszu znajduje się w jakimś podfolderze i jest osobno kompilowana przez webpack.

W VSC przydaje się funkcja duplikowania workspace (**File** -> **Duplicate workspace**) dzięki której w dwóch osobnych kartach VSC można uruchomić niezależnie część serwerową (uruchamianą przez node.js) i część kliencką (uruchomioną w przeglądarce). Obie części projektu są wtedy możliwe do równoległego debugowania.