Programowanie Front-end

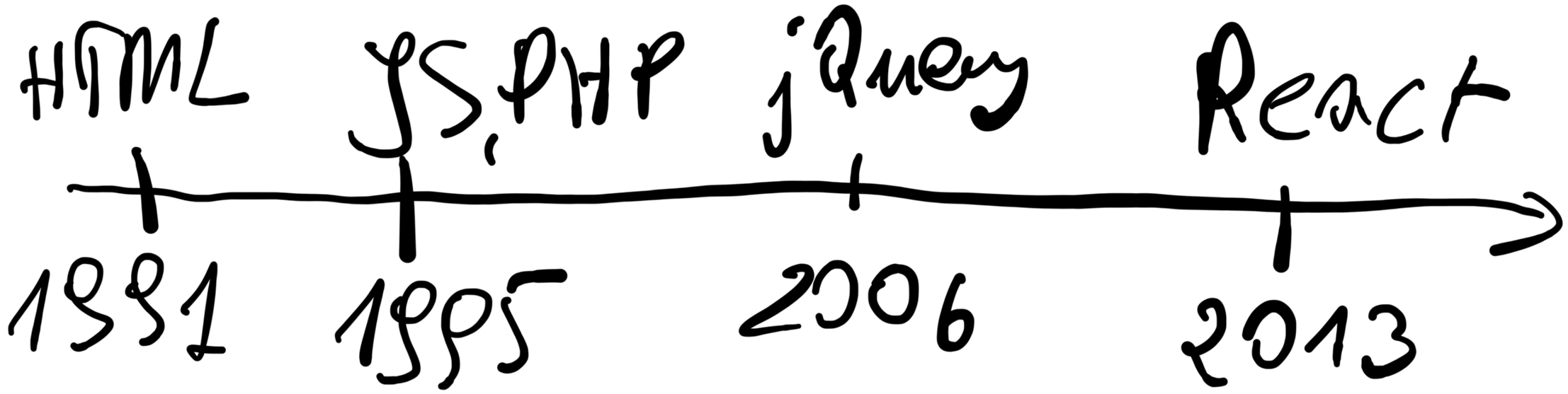
React

mgr inż. Jakub Gogola

Ogłoszenia

- Na Moodle pojawił się już projekt zaliczeniowy na laboratoria

Historia



Historia

HTML

```
<!-- index.html -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Greeting Page</title>
</head>
<body>
  <h1>Hello, World!</h1>
</body>
</html>
```

Historia

PHP

```
<!-- greeting.php -->
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Greeting Page</title>
</head>
<body>
  <?php
    $name = "World";
    echo "<h1>Hello, $name!</h1>";
  ?>
</body>
</html>
```

Historia

JavaScript

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Greeting Page</title>
</head>
<body>
  <h1 id="greeting">Hello, World!</h1>
  <button onclick="updateGreeting()">Change Greeting</button>

  <script>
    function updateGreeting() {
      document.getElementById("greeting").innerText = "Hello, JavaScript!";
    }
  </script>
</body>
</html>
```

Historia

jQuery

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Greeting Page</title>
  <script src="https://code.jquery.com/jquery-3.6.0.min.js"></script>
</head>
<body>
  <h1 id="greeting">Hello, World!</h1>
  <button id="change-greeting">Change Greeting</button>

  <script>
    $("#change-greeting").click(function() {
      $("#greeting").text("Hello, jQuery!");
    });
  </script>
</body>
</html>
```

Historia

React

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

export function Greeting() {
  const [greeting, setGreeting] = useState("Hello, World!");

  return (
    <div>
      <h1>{greeting}</h1>
      <button onClick={() => setGreeting("Hello, React!")}>Change Greeting</button>
    </div>
  );
}
```

Server-Side Rendering

Definicja

- zawartość strony (kod HTML) jest *przygotowywany* przez serwer, który pobiera wszystkie niezbędne dane, umieszcza je w treści strony i gotowy kod wysyła do klienta (przeglądarki), który bezpośrednio go wyświetla

Server-Side Rendering

Zalety i wady

- **Zalety**:
  - szybkie ładowanie strony
  - przyjazne dla SEO
  - dobre rozwiązanie dla statycznych stron lub takich, gdzie konieczne jest *szybkie* wyświetlenie zawartości
- **Wady**
  - każdy request (zapytanie o konkretną stronę) wymaga ponownego renderowania po stronie serwera
  - skrypty JavaScript nadal muszę zostać pobrane osobno i wykonane po stronie klienta

Client-Side Rendering

Definicja

- serwer wysyła do klienta (przeglądarki) prosty plik HTML oraz pliki zawierające JavaScript, które następnie są wykonywane po jego stronie i odpowiadają za pobranie niezbędnych danych i *wyrenderowanie* zawartości strony

Client-Side Rendering

Zalety i wady

- **Zalety**
  - wysoka interaktywność i reaktywność
  - zmniejszone obciążenie serwera
  - dobre rozwiązanie do tworzenie *Single-Page Applications*
- **Wady**
  - wolniejsze ładowanie strony
  - rozwiązanie mniej przyjazne dla SEO

Single-Page Application

Definicja

- aplikacja webowa, które ładuje minimalną ilość kodu HTML i dynamicznie renderuje i aktualizuje jej zawartość z możliwością reagowania na interakcje użytkownika
- nie wymaga pełnego *przerenderowania* całej zawartości strony

React

Co to jest?

- popularna biblioteka języka JavaScript *(open source)* stworzona i rozwijana przez **Facebook'a (Meta)**
- stworzona z myślą o budowaniu *single-page applications*
- pozwala na tworzenie interaktywnych i reużywalnych **komponentów**, które mogą zarządzać swoim stanem
- oparta o język JSX *(JavaScript XML)*

React

Cechy

- oparty na reużywalnych fragmentach kodu nazywanych **komponentami**,
- korzysta z **Virtual DOM** do śledzenia zmian i aktualizowania interfejsu
- posiada **deklaratywną składnię** - programista *opisuje* jak powinien wyglądać interfejs w danym stanie, a React *zajmuje się* optymalnym wyrenderowaniem tego w DOM

React

Historia

- stworzony przez **Facebooka (Meta)** w **2011** roku przez **Jordana Walke** i początkowo nazwany FaxJS
- pierwsze oficjalne wydanie - **29 maja 2013**
- **2015** - wydanie **ReactNative** opartego o React'a, dedykowane dla aplikacji mobilnych
- **2017** - wydanie **React Fiber**, czyli zbioru ulepszonych i zoptymalizowanych algorytmów Reacta używanych do renderowania zawartości *(React 16.0)*
- **2020** - duże zmiany w API React'a *(hooks, functional components - React 17.0)*
- **2022** - React 18 ze mechanizmem współbieżnego renderowania, *automatic batching*, *Suspense for SSR*
- **2024/2025** - spodzieane wydanie stabilnej wersji React 19 zawierającej m.in. nowy kompilator, spore usprawnienia w mechanizmie hook'ów

React
JSX

- React jest oparty o język **JSX**, czyli *JavaScript XML*,
- pozwala na pisanie kodu *HTML-like* bezpośrednio w języku JavaScript

```
import React from 'react';

export function Greeting(props) {
  const currentTime = new Date().toLocaleTimeString(); // Get current time

  return (
    <div>
      <h1>Hello, {props.name}!</h1>
      <p>Current time: {currentTime}</p>
    </div>
  );
}
```

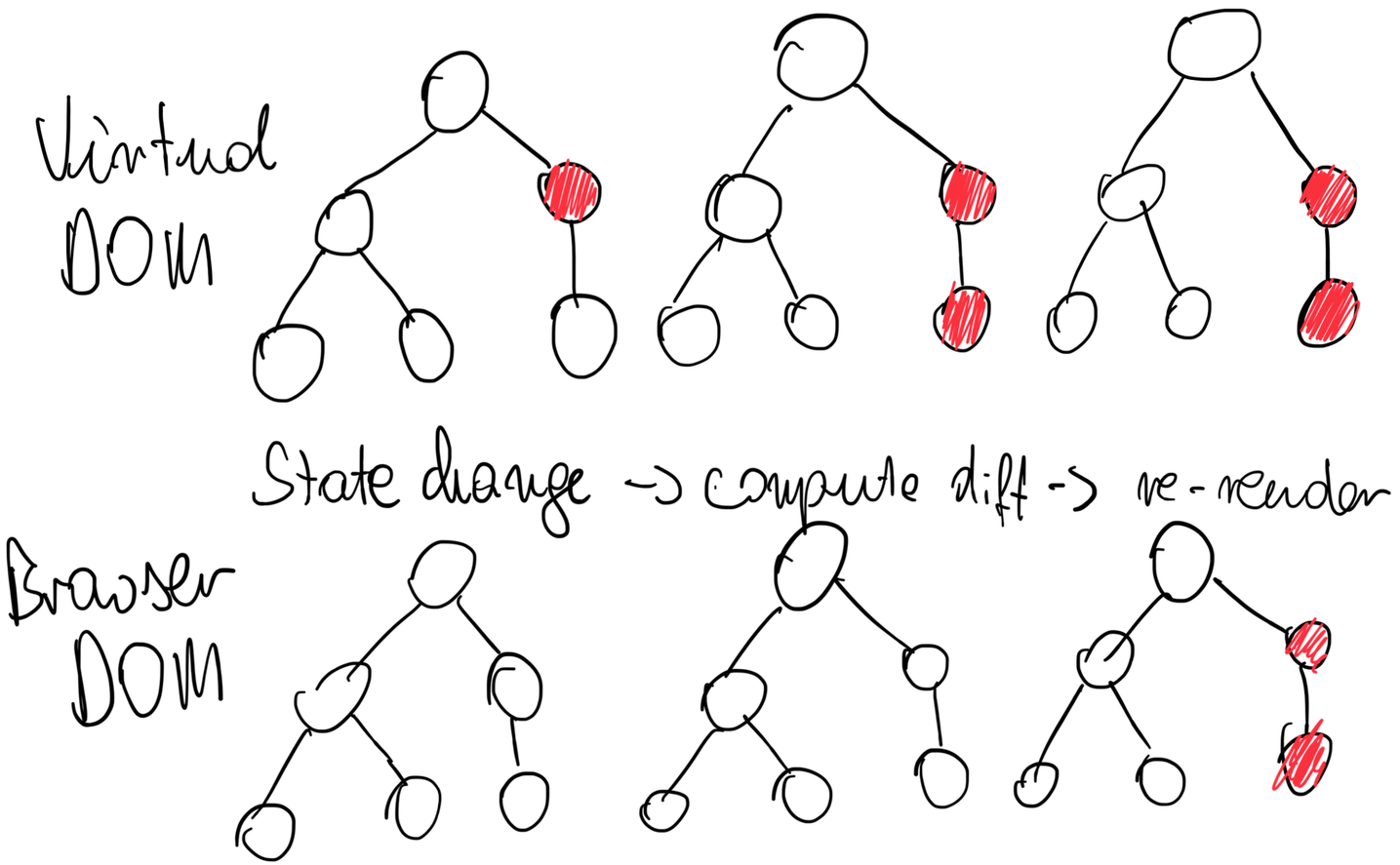
React

Virtual DOM

- *lżejsza* reprezentacja *Document Object Model* stworzona w języku JavaScript
- używana przez biblioteki takie jak React czy Vue.js,
- aktualizowanie struktury VirtualDOM jest szybsze od aktualizacji zawartości standardowego DOM
- React porównuje zaktualizowaną strukturę VirtualDOM z poprzednią i nanosi na DOM tylko niezbędne zmiany

React

Virtual DOM



React

Komponent

- reużywalny, samodzielny fragment kodu, który reprezentuje część interfejsu użytkownika
- pozwala na podzielenie interfejsu na mniejsze, niezależne części, które posiadają własną logikę i funkcje odpowiedzialne za ich wyrenderowanie

React

Komponent

```
import React from 'react';

// DEPREACTED
class ClassComponent extends React.Component {
    render() {
        return <h1>Hello, world!</h1>;
    }
}

function FunctionalComponent() {
    return <h1>Hello, world!</h1>;
}

export function Components() {
    return (
        <div>
            This is a class component: <ClassComponent />
            This is a functional component: <FunctionalComponent />
        </div>
    );
}
```

React

Component properties

- mechanizm pozwalający na parametryzacja komponentu i dynamiczne przekazywanie do nich wartości z komponentu-rodzica

```
import PropTypes from 'prop-types';

function Greeting(props) {
    return <h1>Hello, {props.name}!</h1>;
}
Greeting.propTypes = {
    name: PropTypes.string.isRequired,
};

export function ComponentProps() {
    return (
        <div>
            <Greeting name="Alice" />
            <Greeting name="Bob" />
        </div>
    );
}
```

React

Children prop

- Specjalny parametr komponentu, przez który przekazywane są inne, zagnieżdżone w nim komponenty

```
import PropTypes from 'prop-types';

function Card({children}) {
    return <div className="card">{children}</div>;
}

Card.propTypes = {
    children: PropTypes.node.isRequired,
};

export function Children() {
    return (
        <div>
            <Card>
                <h1>Title</h1>
                <p>This is a card with a title and a paragraph.</p>
            </Card>
            <Card>
                <button>Click Me</button>
            </Card>
        </div>
    );
}
```

React

Props - typy

- Dobrą i zalecaną praktyką jest jawne nadawanie typów dla *props'ów*
- Można to osiągnąć przy użyciu języka TypeScrip:
  - wykorzystując go w pełni do implementacji projektu
  - korzystając jedynie z mechanizmu typów

React

Props - typy

- Istnieje alternatywa dla języka TypeScript w postaci biblioteki `prop-types`
  - Link: https://www.npmjs.com/package/prop-types
- Jest dużo prostsza dla początkujących programistów i pozwala zrozumieć koncepcję typów - korzystanie z niej nie jest jednak zalecane i docelowo należy korzystać z języka TypeScipt

React

Props - biblioteka `prop-types`

```jsx
import React from 'react';
import PropTypes from 'prop-types';

function UserProfile({ name, age, onClick }) {
  return (
    <div>
      <h1>{name}</h1>
      <p>Age: {age}</p>
      <button onClick={onClick}>Click me</button>
    </div>
  );
}

// Defining prop types
UserProfile.propTypes = {
  name: PropTypes.string.isRequired,
  age: PropTypes.number.isRequired,
  onClick: PropTypes.func,
};

// Default props
UserProfile.defaultProps = {
  age: 18, // default age if none provided
};

export default UserProfile;
```

React

Props - TypeScript

```tsx
import React from 'react';

type UserProfileProps = {
  name: string;
  age: number;
  onClick?: () => void; // Optional function prop
};

const UserProfile: React.FC<UserProfileProps> = ({ name, age, onClick }) => {
  return (
    <div>
      <h1>{name}</h1>
      <p>Age: {age}</p>
      <button onClick={onClick}>Click me</button>
    </div>
  );
};

export default UserProfile;
```

React

Hooks

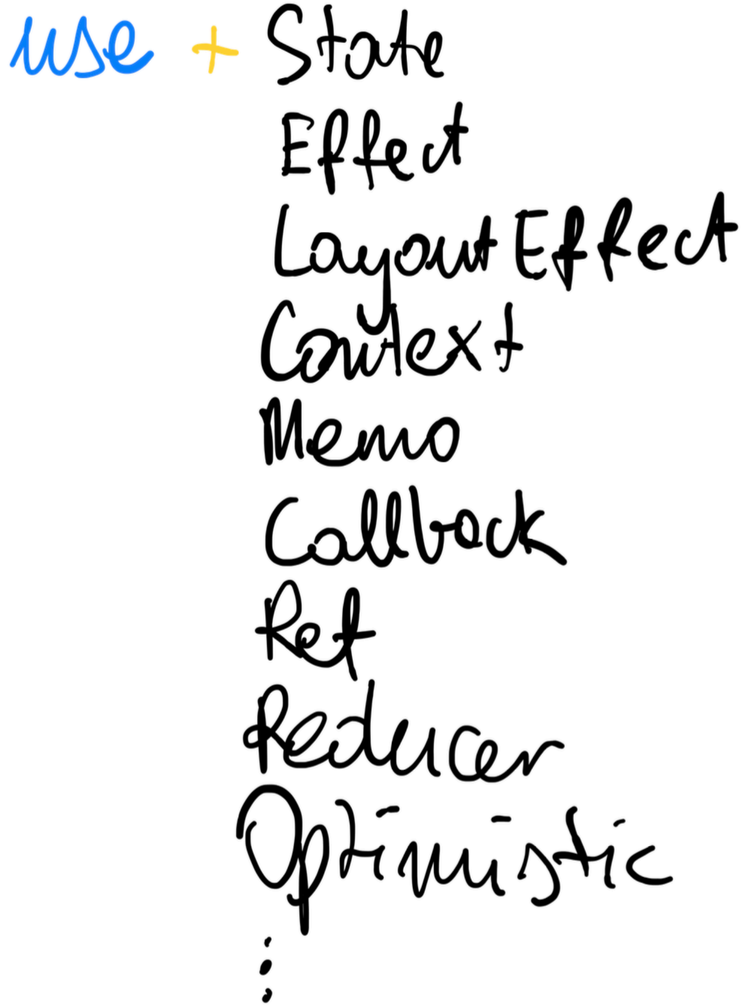
- Mechanizmem mocno powiązany z cyklem życia komponentu są **hooki** *(hooks)* - rodzaj specjalnych funkcji, które zostały wprowadzone w wersji 16.8 biblioteki

- Pozwalają zarządzać stanem komponentu, poszczególnymi etapami *lifecycle*, reagować na event'y i *side effects*, obsługiwać zdarzenia asynchroniczne, konteksty itd.

React

Hooks

- Nazwa funkcji będącej hook'iem zaczyna się od przedrostka *use-*
- React posiada w swoim API zbiór natywnych hook'ów
- Hooki również można ze sobą komponować i tworzyć własne
- W definicji komponentu wszystkie hook'i powinni zostać wywołane przed częścią definiującą renderowaną przez komponent treść



React

Hooks

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

export function HooksOrder() {
    const [hidden, setHidden] = useState(false);

    return (
        <div>
            <h1>Today is {hidden ? '???' : new Date().toLocaleDateString()}</h1>
            <button onClick={() => setHidden(prevState => !prevState)}>
                {hidden ? 'Reveal' : 'Hide'}
            </button>
        </div>
    );
}
```

React
useState

- `useState` pozwala zarządzać stanem komponentu

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

export function UseState() {
    const [count, setCount] = useState(0);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(prevState => ++prevState)}>
                Increase
            </button>
            <button onClick={() => setCount(prevState => --prevState)}>
                Decrease
            </button>
            <button onClick={() => setCount(0)}>Reset</button>
        </div>
    );
}
```

React

useEffect

- `useEffect` pozwala obsłużyć *side effects* takie jak subskrypcje event'ów, asynchroniczne pobieranie danych, update'y DOM
- Jest uruchamiany po wyrenderowaniu komponentu
- Zawiera tablicę zależności, które powinny powodować jego ponowne przeliczenia

React

useEffect

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

export function UseEffect() {
    const [seconds, setSeconds] = useState(0);

    useEffect(() => {
        const interval = setInterval(() => {
            setSeconds(s => ++s);
        }, 1000);

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

    return <p>Elapsed time: {seconds} second(s)</p>;
}
```

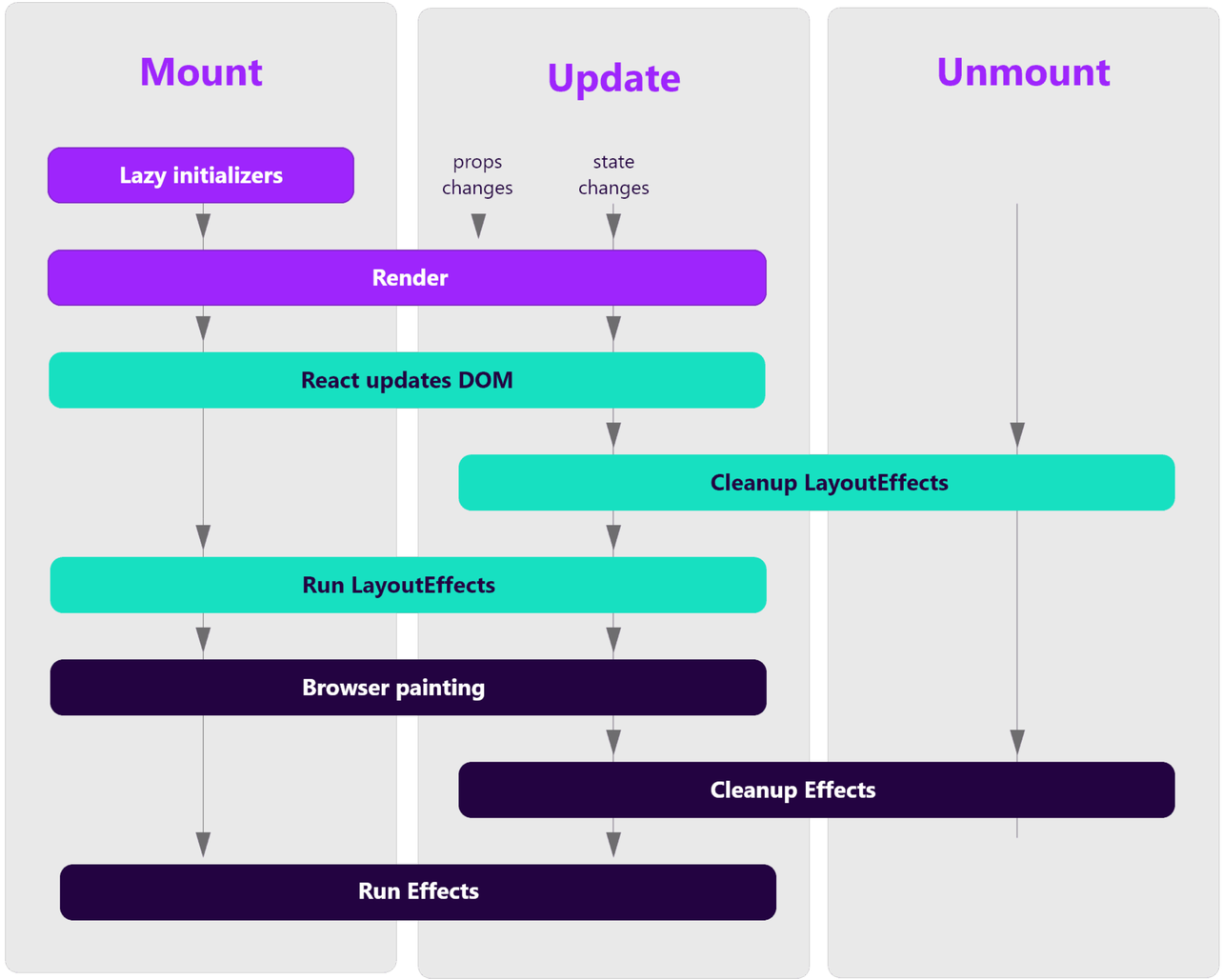
React

Component lifecycle

- Każdy komponent ma swój *cykl życia (lifecycle)*
  - **mounting** - komponent jest renderowany i osadzany w DOM
  - **updating** - stan lub parametry *(props)* komponentu ulegają zmianie i jest on aktualizowany i ponownie renderowany,
  - **unmounting** - komponent jest usuwany z DOM

React

Component lifecycle



React

Component lifecycle

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

export function Lifecycle() {
    const [count, setCount] = useState(0);

    // Runs once after the component mounts
    useEffect(() => {
        console.log('Component mounted');

        return () => {
            console.log('Component will unmount'); // Cleanup before unmounting
        };
    }, []); // Empty dependency array triggers it once, like componentDidMount

    // Runs on every render or when 'count' changes
    useEffect(() => {
        console.log('Component updated: count is now', count);
    }, [count]);

    return (
        <div>
            <p>Count: {count}</p>
            <button onClick={() => setCount(prevState => ++prevState)}>
                Increase Count
            </button>
        </div>
    );
}
```

React

useLayoutEffect

- Hook `useLayoutEffect` działa w podobny sposób jak `useEffect` z następującymi różnicami:

`useLayoutEffect`	`useEffect`
synchroniczny	asynchroniczny
uruchamiany po zaktualizowaniu DOM, ale przed wyrenderowaniem zmian przez przeglądarkę	uruchamiany po zaktualizowaniu DOM i po wyrenderowaniu zmian przez przeglądarkę

React

useLayoutEffect

```
import {useState, useRef, useEffect, useLayoutEffect}
from 'react';

export function UseLayoutEffect() {
    const [width, setWidth] = useState(0);
    const boxRef = useRef(null);

    useLayoutEffect(() => {
        // Measure the width of the box element after it
renders
        if (boxRef.current) {
            setWidth(boxRef.current.offsetWidth);
        }
    }, []); // Dependency array is empty, so this runs
only on the initial render

    useEffect(() => {
        console.log('Width changed:', width);
    }, [width]);

    return (
        <div>
            <div
                ref={boxRef}
                style={{
                    width: '50%',
                    height: '100px',
                    backgroundColor: 'lightcoral',
                    margin: '20px 0',
                }}
            >
                Resize me!
            </div>
            <p>Box width: {width}px</p>
        </div>
    );
}
```

React

useMemo

- Hook `useMemo` pozwala *memoizować* daną wartość i przeliczać ją tylko wtedy, gdy zmienia się jedna lub więcej zależności, od których zależy

React

useMemo

```
import {useState, useMemo} from 'react';
import PropTypes from 'prop-types';

export function ExpensiveCalculationComponent({number}) {
    const [count, setCount] = useState(0);

    // Memoize the result of an expensive calculation
    const expensiveResult = useMemo(() => {
        console.log(`Calculating expensive result for
number: ${number}`);
        return number ** 2;
    }, [number]); // Recalculates only when `number`
changes

    return (
        <div>
            <p>Expensive Calculation Result:
{expensiveResult}</p>
            <p>Count: {count}</p>
            <button onClick={() => setCount(count + 1)}
>Increment</button>
        </div>
    );
}
```

```
ExpensiveCalculationComponent.propTypes = {
    number: PropTypes.number.isRequired,
};

export function UseMemo() {
    return (
        <div>
            <h1>useMemo</h1>
            <ExpensiveCalculationComponent number={5} />
        </div>
    );
}
```

React

useCallback

- `useCallback` działa bardzo podobnie do hook'a `useMemo`
- przelicza funkcję jeżeli zmianie ulegnie jedna lub więcej wartości, od których zależy

React

useCallback

```
import {useState, useCallback} from 'react';
import PropTypes from 'prop-types';

function Item({item, onDelete}) {
    console.log(`Rendering item: ${item}`);
    return (
        <div>
            <span>{item}</span>
            <button onClick={() => onDelete(item)}
>Delete</button>
        </div>
    );
}
Item.propTypes = {
    item: PropTypes.string.isRequired,
    onDelete: PropTypes.func.isRequired,
};
```

```
export function UseCallback() {
    const [items, setItems] = useState(['Apple',
'Banana', 'Cherry']);

    // Memoize the delete function to prevent re-
creation on each render
    const handleDelete = useCallback(itemToDelete => {
        setItems(prevItems => prevItems.filter(item =>
item !== itemToDelete));
    }, []); // No dependencies, so it's created only
once

    return (
        <div>
            <h1>My Item List</h1>
            {items.map(item => (
                <Item key={item} item={item}
onDelete={handleDelete} />
            ))}
        </div>
    );
}
```

React

useContext

- Hook `useContext` pozwala współdzielić dane pomiędzy komponentami bez konieczności podawania ich za pomocą *props'ów*
- Pozwala uniknąć *props drilling*

React

useContext

```
import {createContext, useContext} from 'react';
import {PropTypes} from 'prop-types';

const ToDoContext = createContext();

function ToDoProvider({children}) {
    const todos = [
        {id: 1, text: 'Learn React', done: true},
        {id: 2, text: 'Learn TypeScript', done: false},
        {id: 3, text: 'Learn GraphQL', done: false},
    ];

    return (
        <ToDoContext.Provider value={todos}>{children}</
ToDoContext.Provider>
    );
}
ToDoProvider.propTypes = {
    children: PropTypes.node.isRequired,
};
```

```
function ToDoList() {
    const todos = useContext(ToDoContext);

    return (
        <ul>
            {todos.map(todo => (
                <li key={todo.id}>
                    {todo.done ? '✅' : '❌'}
                    {todo.text}
                </li>
            ))}
        </ul>
    );
}

export function UseContext() {
    return (
        <ToDoProvider>
            <ToDoList />
        </ToDoProvider>
    );
}
```

React

useReducer

- `useReducer` pozwala zarządzać stanem w sposób bardziej złożony
- zasada działania jest bardzo zbliżona do koncepcji wprowadzonej przez bibliotekę Redux - korzysta z *action-based approach*

React

useReducer

```
import {useReducer} from 'react';

const initialState = {count: 0};

function reducer(state, action) {
    switch (action.type) {
        case 'increment':
            return {count: state.count + 1};
        case 'decrement':
            return {count: state.count - 1};
        case 'reset':
            return {count: 0};
        default:
            return state;
    }
}
```

```
export function UseReducer() {
    const [state, dispatch] = useReducer(reducer,
initialState);

    return (
        <div>
            <p>Count: {state.count}</p>
            <button onClick={() => dispatch({type:
'increment'})}>
                Increment
            </button>
            <button onClick={() => dispatch({type:
'decrement'})}>
                Decrement
            </button>
            <button onClick={() => dispatch({type:
'reset'})}>Reset</button>
            <button onClick={() => dispatch({type:
'unkown-action'})}>
                Unknown action
            </button>
        </div>
    );
}
```

React

useRef

- Hook `useRef` pozwala na dostęp i przechowywanie bezpośredniej referencji do elementów DOM
- Jego wartość pozostaje stała i niezależna od *re-renderów* komponentu
- Może zostać wykorzystany do przechowywania wartości, które nie powinny ulegać zmianie podczas kolejnych cyklów życia komponentu

React

useRef

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

export function UseRef() {
    const inputRef = useRef(null);

    const focusInput = () => {
        inputRef.current.focus();
    };

    return (
        <div>
            <input ref={inputRef} type="text" />
            <button onClick={focusInput}>Focus Input</button>
        </div>
    );
}
```

Rect

Event handling

- React pozwala, podobnie jak *Vanilla JS*, na obsługę różnych event'ów użytkownika
- Elementy w języku JSX posiadają atrybuty, które pozwalają nasłuchiwać na poszczególne event'y:
  - `onClick`
  - `onSubmit`
  - `onChange`
  - `onFocus`
  - `onBlur`
  - …

React

Event handling

```jsx
import {useState} from 'react';

export function EventsHandling() {
    const [inputValue, setInputValue] = useState('');
    const [submitted, setSubmitted] = useState(false);

    // Handle click event
    const handleClick = () => {
        alert('Button clicked!');
    };

    // Handle input change event
    const handleChange = event => {
        if (submitted) {
            setSubmitted(false);
        }

        setInputValue(event.target.value);
    };

    // Handle form submission event
    const handleSubmit = event => {
        event.preventDefault(); // Prevent page refresh
        setSubmitted(true);
        console.log('Form submitted with value:', inputValue);
    };

    return (
        <div>
            {/* onClick event example */}
            <button onClick={handleClick}>Click Me</button>

            {/* onChange event example */}
            <input
                type="text"
                value={inputValue}
                onChange={handleChange}
                placeholder="Type something"
            />

            {/* onSubmit event example */}
            <form onSubmit={handleSubmit}>
                <button type="submit">Submit</button>
            </form>

            {/* Display submitted data */}
            {submitted && <p>Submitted value: {inputValue}</p>}
        </div>
    );
}
```

React

Conditional rendering

- Renderowanie warunkowe, czyli *conditional rendering*, pozwala uzależnić wyświetlaną zawartość od warunków logicznych, które muszą zostać spełnione do jej wyświetlenia

React

Conditional rendering

```
import PropTypes from 'prop-types';

function TernaryOperator({loggedIn = false}) {
    return (
        <div>{loggedIn ? <h1>Welcome User</h1> :
<h1>Welcome Guest</h1>}</div>
    );
}
TernaryOperator.propTypes = {
    loggedIn: PropTypes.bool
};

function ShortCircuitOperator({hasNewMessages = false})
{
    return (
        <div>
            <h1>Welcom to Messenger Application!</h1>
            {hasNewMessages && <p>You have new
messages!</p>}
        </div>
    );
}
ShortCircuitOperator.propTypes = {
    hasNewMessages: PropTypes.bool
};
```

```
function IfStatement({loading = false}) {
    if (loading) {
        return <h1>Loading...</h1>;
    }
    return <h1>Loaded!</h1>;
}
IfStatement.propTypes = {
    loading: PropTypes.bool,
};

export function ConditionalRendering() {
    return (
        <div>
            <TernaryOperator loggedIn />
            <TernaryOperator />
            <ShortCircuitOperator hasNewMessages />
            <ShortCircuitOperator />
            <IfStatement loading />
            <IfStatement />
        </div>
    );
}
```

React

Listy

- Czasem zachodzi potrzeba wyświetlenia listy elementów o podobnej strukturze, różniących się jednak treścią
- React pozwala na bezpośrednie renderowanie tablic elementów/ komponentów

```
export function Lists() {
    const items = ['Apple', 'Banana', 'Cherry', 'Date'];

    return (
        <ul>
            {items.map((item, index) => (
                <li key={`item-${index.toString()}`}>{item}</li>
            ))}
        </ul>
    );
}
```

React

Listy

- Każdy z elementów/komponentów listy powinien posiadać unikalny identyfikator zwany **kluczem** nadawany za pomocą atrybutu `key`
- W przypadku iterowacyjnego renderowania elementów (np. przy pomocy metody `map`) powinno **unikać się** bezpośredniego stosowania jako identyfikatorów indeksów poszczególnych elementów

React

Listy - klucze

- Klucze (unikalne identyfikatory) pozwalają bibliotece React rozróżnić poszczególne elementy należące do listy i śledzić:
  - zmianę stanu
  - usunięcie elementu z listy
  - dodanie nowego elementu do listy

React

Listy - klucze

- Stosowanie kluczy dla każdego z elementów listy jest ważne z następujących powodów
  - **optymalizacja wydajności** - klucze pomagają React’owi aktualizować tylko te elementy, które się zmieniły aktualizowania całej listy
  - **utrzymanie porządku na liście** - porządek elementów na liście nie zmienia się nawet wtedy, gdy któreś z nich zostają zaktualizowane, usunięte lub dodane

React

Listy - klucze

- Zasady poprawnego tworzenia kluczy
  - klucze **powinny** być różne pomiędzy poszczególnymi elementami listy
  - klucze **nie powinny** być indeksami poszczególnych elementów,
  - klucze **nie muszą** być unikalne globalnie w skali aplikacji, a jedynie w obrębie danej listy

React

Listy - klucze

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

export function ListKeys() {
    const [items, setItems] = useState([
        {id: 1, name: 'Apple'},
        {id: 2, name: 'Banana'},
    ]);

    // Add a new item to the list
    const addItem = () => {
        const newItem = {
            id: items.length + 1,
            name: `Fruit ${items.length + 1}`,
        };
        setItems([...items, newItem]);
    };

    // Remove an item from the list
    const removeItem = id => {
        setItems(items.filter(item => item.id !== id));
    };

    return (
        <div>
            <ul>
                {items.map(item => (
                    <li key={item.id}>
                        {item.name}
                        <button onClick={() =>
removeItem(item.id)}>
                            Remove
                        </button>
                    </li>
                ))}
            </ul>
            <button onClick={addItem}>Add Item</button>
        </div>
    );
}
```

React

Controlled components

- **Controlled component** to taki, w którym wartości poszczególnych elementów są ściśle powiązane ze stanem komponentu
- Jest to sposób obsługi wszystkich wartości wprowadzanych przez użytkownika *(inputs)*
- Zmiana wartości przez użytkownika ma swoje natychmiastowe odzwierciedlenie w stanie komponentu; możliwa jest jej walidacja

React

Formularze

- *Controlled components* pozwalają na tworzenie i obsługę formularzy
- Wykorzystanie tego mechanizmu pozwala na bieżąco odczytywać, walido wać i przetwarzać (np. wysłać do serwera) dane wprowadzane przez użytkownika

React

Formularze

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

import './styles.css';

export function ControlledForm() {
    const [name, setName] = useState('');
    const [email, setEmail] = useState('');

    const handleNameChange = event => {
        setName(event.target.value);
    };

    const handleEmailChange = event => {
        setEmail(event.target.value);
    };

    // Handle form submission
    const handleSubmit = event => {
        event.preventDefault(); // Prevents page refresh
        console.log('Form Submitted:', {name, email});
        // Additional processing, such as API calls, could
go here
    };

    return (
        <form className="controlled-form"
onSubmit={handleSubmit}>
            <label>
                Name:
                <input type="text" value={name}
onChange={handleNameChange} />
            </label>
            <label>
                Email:
                <input
                    type="email"
                    value={email}
                    onChange={handleEmailChange}
                />
            </label>
            <button className="form-submit-button"
type="submit">
                Submit
            </button>
        </form>
    );
}
```

React

Asynchroniczne pobieranie danych

- Jednym ze wspominanych wcześniej zastosowań hook'a `useEffect` jest obsługa asynchronicznych zdarzeń takich jak subskrypcje event'ów czy pobieranie danych np. z zewnętrznego serwera
- Jednym z głównych zadań aplikacji webowych jest prezentacja danych jak i wysyłanie ich do serwera
- Dobry interfejs użytkownika powinien zapewnić jak najlepszy *user experience*

React

Asynchroniczne pobieranie danych

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

const fetchData = async () => {
    await new Promise(resolve => setTimeout(resolve, 3000));

    const response = await fetch('https://api.github.com/
users');
    const data = await response.json();
    return data;
};

export function DataFetching() {
    const [data, setData] = useState(null);
    const [error, setError] = useState(null);

    useEffect(() => {
        fetchData()
            .then(data => {
                setData(data);
            })
            .catch(error => {
                setError(error);
            });
    }, []);

    if (!data && !error) {
        return <div>Loading...</div>;
    }

    return (
        <div>
            <h1>GitHub Users</h1>
            {error && <div>{error.message}</div>}
            {data && (
                <ul>
                    {data.map(user => (
                        <li key={user.id}>{user.login}</li>
                    ))}
                </ul>
            )}
        </div>
    );
}
```

Literatura

- Dokumentacja biblioteki React - https://react.dev/
- Przykłady kodu prezentowane na wykładzie - https://github.com/JakubGogola-IDENTT/dsw-frontend-lecture-2024/tree/main/lecture-3
- React on MDN - https://developer.mozilla.org/en-US/docs/Learn/Tools_and_testing/Client-side_JavaScript_frameworks/React_getting_started

Dziękuję za uwagę!