REACT

What is it?

JavaScript library for creating user interfaces
often used to create Single Page Applications

Main assumptions

Declarative

React makes it much easier to create interactive UIs: it takes care of smooth updating and re-rendering of the relevant components.

Based on components

Since component logic is written in JavaScript rather than templates, passing complex data structures and storing the application state outside of the DOM tree becomes easier.

Learn once, use everywhere

React works in isolation from the rest of the technology stack, so you can create new functionalities without having to rewrite existing code.

React

Advantages

- it has high performance as it is based on a virtual domain
- its components can be used multiple times
- it can be used with any framework
- it is stable, there is a large community behind it, it is constantly developing
- uses a one-way data flow
- allows you to build a dynamic interface
- can be used by novice programmers

React

Disadvantages

- hmmm? ;)
- architecture - responsibility falls on us
- JSX - you have to learn it
- additional libraries - to create complex applications you need to learn a lot about them
- the speed of library development - it may turn out to be problematic for someone

React

React allows us to build web applications, don't confuse it with mobile applications. However, a separate, very similar creation was created: React Native, which allows you to create a mobile application using JavaScript.

More info: https://reactnative.dev/

Create React App

is the officially supported way to create a single page application in React.
It offfers modern build settings with no configuration.

Create react app

What's included in it?

- **Package manager** - **yarn** or **npm**. It allows you to use a huge ecosystem of additional packages. It allows you to easily install and update them.

- **Bundler** - **webpack**.It allows you to write modular code and package it into small packages to optimize load times.

- **Compiler** - **babel**. It allows the use of new versions of JavaScript while maintaining compatibility with older browsers.

Create react app

Available scripts:

- **npm start**
  Runs the application in development mode. Opens **http://localhost:3000** to view it in a browser. The page will be reloaded if you make any changes (hot-reload). Any lint errors will also be displayed in the console.

- **npm test**
  Runs application tests using the library **Jest** in watch mode, which means that the tests will run automatically after a change in the code.

Create react app

Available scripts:

- **npm run build**
  Builds the production version of the application in the build folder. It properly connects React in production mode and optimizes the build for the best performance. Compilation is minimized and the filenames contain abbreviations.
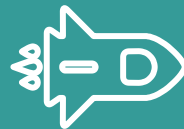
- **npm run eject**
  Note: This is a one-way operation. Once done, it cannot be undone! This command will remove build settings from the project.

Create react app

Files structure:

```
my-app/
  README.md
  node_modules/
  package.json
  public/
    index.html
    favicon.ico
  src/
    App.css
    App.js
    App.test.js
    index.css
    index.js
    logo.svg
```
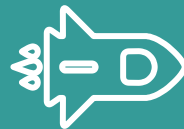
TASK

Create React app with usage of **create-react-app.**
npx create-react-app my-app

Run the applications in developer mode according to the instructions displayed in the console and change the text to see how hot reload works.

TASK

Create a production version of your application with
npm run build

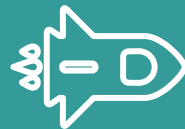Then go to the build folder and open **index.html**.

Create react app

The configuration in create-react-app is made in such a way that webpack will try to read files from the system source. Therefore, we have to show webpack where to read files and for our case it will be the folder in which the application is located.

To do this, just add: **"homepage": "."** to package.json.

Create react app also allows you to easily publish your application to gh-pages: https://github.com/gitname/react-gh-pages

TASK

Add **homepage** to package.json and run **npm run build** again

Then go to the build folder and open **index.html**.

JSX

it is a wrapper on JavaScript that adds the ability to insert code
html (or React components) directly in your code, instead of a string

JSX

JSX element example:
const element = <h1>Hello world!</h1>;

JSX is translated one-to-one into React "elements". JSX is compiled with the Babel to Javascript and the actual HTML elements are created with the createElement function and then added to the DOM tree by the render function.

React does not require JSX, however most programmers find it a useful tool to visualize what is happening in JavaScript code that works with graphical interfaces.

Note: JSX is not HTML!

JSX

Since JSX compiles to Javascript, we can use variables inside it or embed any Javascript expression. For React to know that you want to use a variable, you must enclose the expression in braces.

An example of a JSX element with an expression:
const name = 'Janusz';
const element = <h1>Hello, {name}!</h1>;

In other words: we use the curly braces to "use" javascript in JSX.

JSX

JSX is also an expression, so we can use it as the results of a conditional function for example:

if (statement) { return <h1>Hello, world!</h1>; }

In JSX, we can also pass attributes to HTML elements:

const element = <div tabIndex="0"></div>;

const element = ;

JSX

JSX tags can also contain child elements:

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you.</h2>
  </div>
);
```

JSX

As JSX syntax is closer to JavaScript than to HTML, React DOM uses camelCase notation instead of HTML attribute names for naming arguments and you cannot use reserved words.

For example, in JSX **class** becomes **className** and instead of **tabindex** we use **tabIndex**.

Attention! To use JSX within a given component, React must be imported:

import React from 'react';

This is because JSX is compiled into the React.createElement method.

As of React 17, there is no need to add this import.

JSX

In JSX, one element must always be returned because of how the React.createElement method works. So this type of notation is incorrect:

```
const element = (
    <h1>Hello!</h1>
    <h2>Good to see you.</h2>
);
```

To fix it, you can use an HTML tag or a React fragment:

```
const element = (
    <>
        <h1>Hello!</h1>
        <h2>Good to see you.</h2>
    </>
);
```

RENDER FUNCTION

To render a React element to a DOM node, pass the DOM node
ReactDOM.createRoot() and then on the result run node with React element as
argument:

const element = <h1>Hello, world!</h1>;
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(element);

ReactDOM?
Yes, React creates its own DOM tree. This is called Virtual DOM.

VIRTUAL DOM

React stores the application's entire DOM in memory, after it changes state, it looks up differences between virtual and real DOM and updates changes.

VIRTUAL DOM

ReactDOM compares the element and
its descendants to the previous one,
and applies only those DOM tree
updates necessary to bring it to the
desired state.

Hello, world!

It is 12:26:47 PM.

```
Console   Sources   Network   Timeline
▼<div id="root">
  ▼<div data-reactroot>
      <h1>Hello, world!</h1>
    ▼<h2>
        <!-- react-text: 4 -->
        "It is "
        <!-- /react-text -->
        <!-- react-text: 5 -->
        "12:26:46 PM"
        <!-- /react-text -->
        <!-- react-text: 6 -->
        "."
        <!-- /react-text -->
      </h2>
  </div>
</div>
```

TASK

Let's see this Virtual DOM!
Install React's development tools and check the virtual DOM.
react-developer-tools

Component

Components are independent and reusable pieces of code. They serve it same purpose as JavaScript functions, but run in isolation and can return HTML via a rendering function.
They contain both HTML tags, and the logic associated with them

Components

Let's try to divide the following application into components:

Components

Yellow - FilterableProductTable

Blue - SearchBar

Green - ProductTable

Turquoise - ProductCategoryRow

Red - ProductRow

Components

In React, we distinguish 2 types of components:

functional component

```
function Welcome() {
    return <h1>Cześć</h1>;
}
```

class component

```
class Welcome extends React.Component {
    render() {
        return <h1>Cześć</h1>;
    }
}
```

Components

Component names are always capitalized. This is because in JSX a lowercase letter suggests that it is an HTML tag instead of a component.

To use a given component, you need to import it from the given file and embed it in JSX, just like in the case of HTML tags.

```
import { Welcome } from './Welcome'

<div className="container">
    <Welcome />
</div>
```

Components

Usually, each component is a separate file with the extension js, which contains the definition of the component and its export so that it can be imported later. We distinguish 2 types of exports: export of a specific value and default export.

export function Welcome() { … }

function Welcome() { … }
export default Welcome;

import { Welcome } from ' … ';

import Welcome from ' … ';

Both exports are equivalent, only the way of importing them differs. In the case of default, the name of the import can be anything.

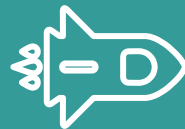# Function component

We call them a regular JavaScript function whose name starts with a capital letter and returns a React element.

```
function Welcome() {
  return <h1>Hello world!</h1>;
}
```

TASK

Create your first function component called **MyName** that displays:

"My name is <YOUR_NAME>"

Add it to the main **<App />** component in the application, don't forget to import.
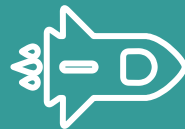
Function component

Each function component takes a single "props" argument (which stands for "properties"), which is a data object.

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

To pass properties to a component, we pass them similarly to attributes in HTML tags:
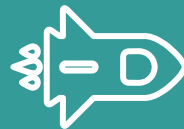
```
<Welcome name="Kamil" />
```

TASK

Change the function component **MyName** to one that will display the name given by props.

Remember to include your name in props where this component is used.

TASK

Add another props containing your last name to the **MyName** component named *surname* and use it in your message by extending it to the version:
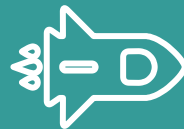
"My name is <YOUR_NAME>, and my surname is <YOUR_SURNAME>"

Function component

Of course, a component can contain some logic, such as a condition:

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;

  if (isLoggedIn) {
      return <UserGreeting />;
  }

  return <GuestGreeting />;
}
```

TASK

Add a condition to the **MyName** component which, depending on whether props 'surname' has been given, will display the text:

My name is … or *My name is … ,and surname is …*

Use the **MyName** component in two different versions in the main App component.

Styling

React's CSS works just like regular HTML, with a slight difference in adding styles. We can add a css file or add inline styles.

In the case of a css file, create it and add styling to it, then import the file in the component where you want to use these styles.

App.css

App.js

```css
.root {
    color: red
}
```

```js
import './App.css';

function App() {
    return <div className='root'>Tekst</div>
}
```
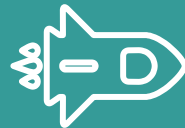
Styling

Caution! Although it may seem that thanks to imports, each component can use only the styles imported to it, but imported styles are available for each component.

On the other hand, we can add inline styles also as a JavaScript object, where two-part names are changed to camelCase instead of being separated by a semicolon:

<div style={{color: 'red', backgroundColor: 'white' }}>Text</div>

TASK

Create a new **Contact** component that will display your contact details in accordance with the attached graphics. Contact details should be provided as 1 props object in the following form:

{
  phone: '111 222 333',
  address: { street: 'Słowackiego', city: 'Gdańsk', number: 37 },
  email: 'moj@mail.com'
}

TEL

+48 575 707 887

ADRES

ul. Słowackiego 37
Gdańsk Wrzeszcz
(obok Garnizonu)

EMAIL

kontakt@muzykon.pl

CSS Modules

CSS Modules allows the scoping of CSS by automatically creating a unique className of the format *[filename]_[classname]__[hash]*. This means that it fixes the problem with overriding the classNames by other components and you can use the same name of class for every component.

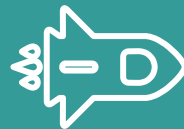Create react app supports CSS modules out of the box using the *[name].module.css* file naming convention.

https://create-react-app.dev/docs/adding-a-css-modules-stylesheet/

App.module.css

```css
.root {
    color: red
}
```

App.js

```js
import styles from './App.module.css';

function App() {
    return <div className={styles.root}>Tekst</div>
}
```

TASK

Change name of Contact.css file to Contact.module.css and change the way the file is imported and use CSS modules.
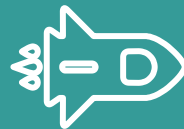
Event handling

React's event handling is very similar to that of the DOM tree. However, there are a few syntax differences:

- React events are written with camelCase, not lowercase.
- in JSX, the event handler is passed as a function, not a string, so it is passed without "()"

```
<button onClick={activateLasers}>
    Activate lasers
</button>
```

TASK

In the newly created **Contact** component, add a button with the text "Send" which, when clicked, will display an alert with the text:

"Thank you! Feel free to come!"

Try to display props data in the alert:
"Thank you! Feel free to visit me at <YOUR_ADDRESS>! "
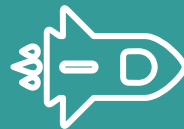
Children

If you pass **children** to our component in JSX, they are available in props under the name children.

```
<Wrapper>
    <div>Children</div>
</Wrapper>

function Wrapper(props) {
    return <div style={{ textAlign: 'center' }}>
        {props.children}
    </div>
}
```

TASK

Create a **Wrapper** component that centers the elements in the middle (using a flexbox) and displays the children given to it.

Use it in the main <App /> component and wrap all the components created so far with it.

Lists and keys

To display the list of elements in React, we need to transform this list of elements into JSX elements using the JavaScript map function, we add the created collection to JSX in braces **{ }**:
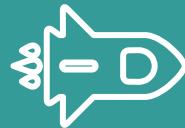
```
function Photos() {
    const photos = ['photo1', 'photo2'];

    return <ul>
        {photos.map(photo => <li>{photo}</li>)}
    </ul>
}
```

Lists and keys

Keys help React identify which items have changed, added or removed. Keys should be given to the elements inside the array so that the elements gain a stable identity.
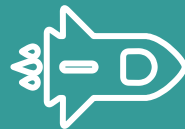
```
function Photos() {
    const photos = [{ id: 1, name: 'photo1'}, { id: 2, name: 'photo2'}];

    return <ul>
        {photos.map(photo => <li key={photo.id}>{photo.name}</li>)}
    </ul>
}
```

TASK

Create a **MyFavouriteDishes** component that will display a list of your favorite dishes (at least 3). Dishes should be provided as props. Remember about unique keys!

TASK

Create a new **AboutMe** component, where you will move the **MyName** and **Contact** components and put **MyFavouriteDishes** in it. Add it to the main **<App />** component in the application.

STYLED COMPONENTS

Styled components

This is a package that gives us a way to improve CSS for styling React component systems.

- keeps track of which components are rendered on the page and injects their styles and nothing else, fully automatically
- no errors in class names: styled-components generates unique class names for your styles
- easier CSS removal: it can be hard to tell if a class name is being used somewhere in your codebase
- simple dynamic styling
- hassle-free maintenance

Styled components

```
const Title = styled.h1`
  font-size: 1.5em;
  text-align: center;
  color: palevioletred;
`


return (
  <Wrapper>
    <Title>
      Hello World!
    </Title>
  </Wrapper>
);
```

```
const Wrapper = styled.section`
  padding: 4em;
  background: papayawhip;
`
```

Styled components

You can pass a function ("interpolations") to a styled component's template literal to customize it based on its props.

```
const Button = styled.button`
  background: ${props => props.primary ? "palevioletred" : "white"};
  color: ${props => props.primary ? "white" : "palevioletred"};
  font-size: 1em;
`;


return (
  <div>
    <Button primary>Primary</Button>
  </div>
);
```
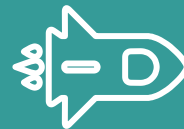
Styled components

To easily create a new component that inherits the style of another, just wrap it in a **styled()** constructor.

```
const TomatoButton = styled(Button)`
  color: tomato;
  border-color: tomato;
`;


return (
  <div>
    <Button>Normal Button</Button>
    <TomatoButton>Tomato Button</TomatoButton>
  </div>
);
```

TASK

Stwórz nowy komponent **ContactStyled**, component that will be an exact copy of the **Contact** component, but made with styled components.

HOOKS

Hooks

What is a hook?

Hook is a special function that allows you to "hook" into internal React mechanisms.
Its name starts with **use***. It only works with function components as of React 16.8.

Presentation of Hooks on ReactConf: https://youtu.be/dpw9EHDh2bM?t=686

State

In React components, we can save the local state. In the case of functional components, we use Hooks to achieve this:

const [value, setValue] = useState(0)

value – state value
setValue – setter, sets value state
argument of useState is the initial state value (0 in this case)
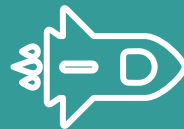
State

Example:

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

const Component = () => {
    const [value, setValue] = useState(0);

    const increase = () => setValue(value + 1);

    return <>
        <h1>{value}</h1>
        <button onClick={increase}>increase</button>
    </>
}
```

TASK

Create Game component that displays:

"*Welcome to game <GAME_NAME>*" where the name of the game is passed by props.

Display information under the greeting:

"*Your number of points is: <POINTS>*",

where the points are provided by the component state and are initially 0.

Add it to the main **<App />** component in the application.

TASK

Modify the **Game** component by adding a button that, when clicked, will increase the number of points by 5 and a button that will decrease this value by 5.

When a player's score is below zero, the score should be red. (try to use styled components with passing prop to it)

Effect

Sometimes we want to run some extra code after React updates the DOM tree. Requests to server, manual modification of the DOM tree or logging. We can do this with another hook: **useEffect**.

By using this hook, you are telling React that your component needs to do something after rendering. React will remember the function that was passed to the Hook (we'll name it "effect"), and then call it once it has updated the DOM tree.

Effect

Example:

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

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

  useEffect(() => {    document.title = `Kliknięto ${count} razy`;  });

  return <div>
     <p>Kliknięto {count} razy</p>
     <button onClick={() => setCount(count + 1)}>
      Kliknij mnie
     </button>
   </div>
}
```

TASK
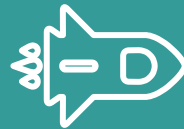
By using the **useEffect** hook in the Game component, show an alert when you reach 50 points:

"Congratulations! You won the game <GAME_NAME>" where the game name is a props value.

Effect

However, sometimes we only want an effect to trigger once - to achieve this we can add a second argument to useEffect, which is a dependency array. What does it mean? If count is 5 and our component is re-rendered with count still 5, React will compare [5] from the previous render and [5] from the next render. Since all the elements in the array are the same (5 === 5), React will ignore the effect.

```
useEffect(() => {
  document.title = `Clicked ${count} times`;
}, [count]);
```

// run effect only when count changes

TASK
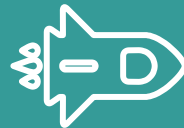
Add dependency array with **points** in **useEffect** hook in the Game component.

Effect

So if we want to perform a given operation only once after the component is rendered for the first time, we have to pass an empty dependency array in **useEffect**:

```
useEffect(() => {
  console.log('Component mounted');
}, []);
```

// run effect only after first render

Downloading data

Downloading data in React is best done after the first render of the component. Why? The user already receives information that the application is working and may change something in the case of a miss click + we can, for example, show the loading status.

```
useEffect(() => {
    fetch('.../posts').then(r => r.json()).then(posts => {
        console.log('Fetched: ' + posts);
    });
}, []);
```

TASK

Create a **Users** component and download it from the address:
https://jsonplaceholder.typicode.com/users .

Display the downloaded data as a list:
"{name} {surname} works in {companyName}".

TASK

Replace the list items in the **Users** component with another component called **User**, which will receive data about the user in the form of props.

Then add logic in it so that the phone number with the area code in italics is also displayed.

Effect

However, sometimes we want to clean up something after a component or we want to perform some action at the unmount phase. In this case, in the function added to the hook, we have to return a new function that will be called at the end of the effect with each subsequent rendering.

```
useEffect(() => {
   ChatAPI.subscribe(props.id, handleStatusChange);

   return () => {
    ChatAPI.unsubscribe(props.id, handleStatusChange);
   };
 });
```

Hooks

You can use each Hook multiple times, for example you can use **useState** more than once to save more than 1 field in a state.

So you can also use multiple effects. This allows you to isolate unrelated logic into separate effects.

Hooks allow code to be broken down into smaller chunks in terms of their responsibility, not the name of the lifecycle method. React will execute each effect used in the component in the order it was added.

Advanced hooks

useCallback returns a remembered callback that will only change if any of the dependencies given in the array as the second argument changes.

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

useMemo returns the cached value, which will be recalculated only if any of the dependencies given in the array as the second argument changes. This optimization avoids costly computations with every render.

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

Class component

Class component

You can also use an ES6 class to define a component. From React's point of view, there is no difference between them. However, they differ from functional components in several syntactic and performance issues.

```
class Welcome extends React.Component {
    render() {
        return <h1>Hello world!</h1>;
    }
}
```

Caution! Class components require a render function which should return a piece of JSX or null.

Class components - props

In the case of a class, we do not have access to function arguments, so we read props in a class component by referring to this.

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

Class component - state

As with a functional component, you can define a local state in a class component. The state in a class component is always an object.

The state can be defined in two ways

- definition in class constructor
- with a shortened notation - by class properties

As in the case of props, the state in a class component is read by reference to this.

Class component - state

definition in constructor:

```
class Score extends React.Component {
    constructor(props) {
        super(props);
        this.state = { value: 10 }
    }
    render() {
        return <h2>Score: {this.state.value}.</h2>
    }
}
```

Class component - state

with the shortened version:

```
class Score extends React.Component {
    state = { value: 10 }

    render() {
        return <h2>Score: {this.state.value}.</h2>
    }
}
```

Class component - state

The state can be changed using the **this.setState** method, which takes as an argument or an object that will be merged with the current state (or a function that takes state and props as arguments and returns the changed state).

this.setState({comment: 'Witam'});

Class component - state

Example

```
class Score extends React.Component {
    state = { value: 10 }
    increase = () => this.setState({ value: this.state.value + 1 });

    render() {
        return <>
            <h2>Wynik: {this.state.value}.</h2>
            <button onClick={this.increase}>increase</button>
        </>
    }
}
```

Note that increase is a method of the class and must be preceded with **this** to refer to it.

Class component - state
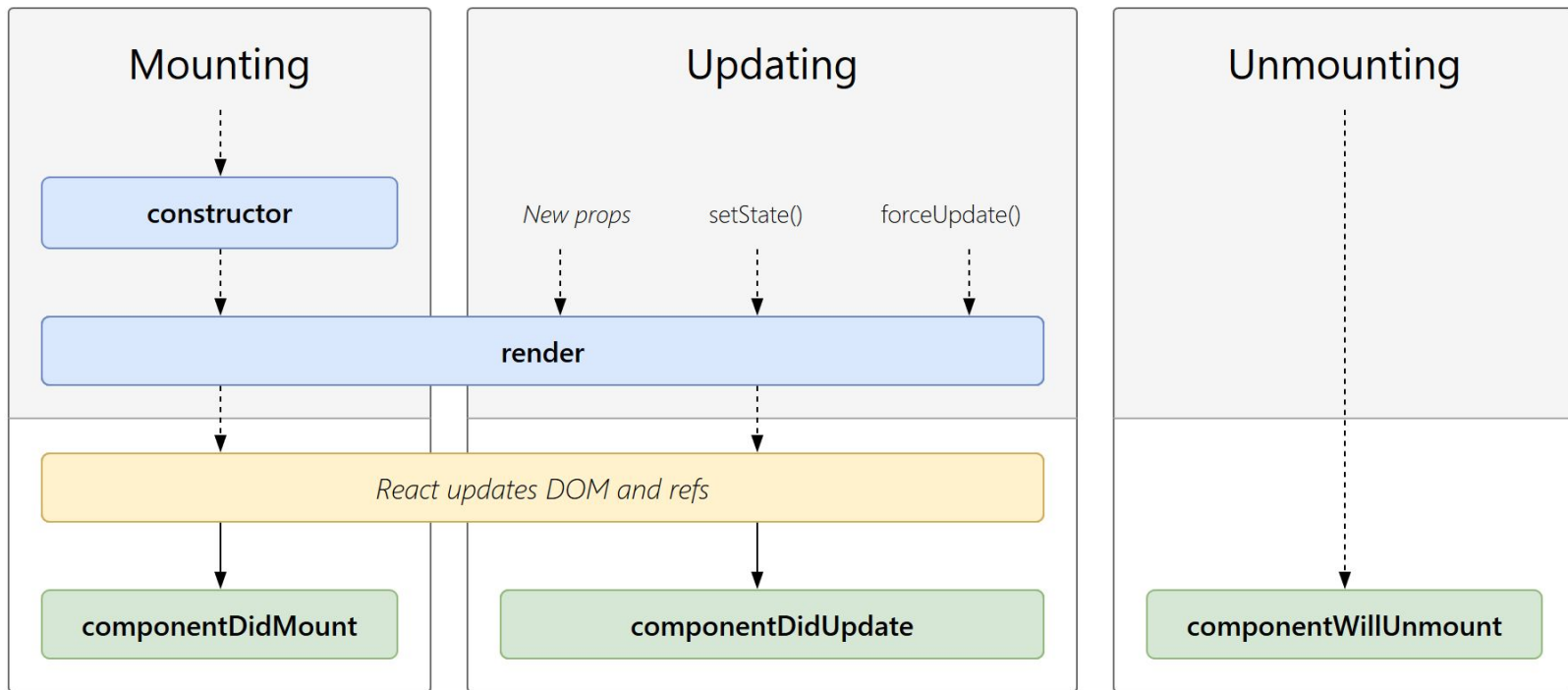
State and method notes:

1. Do not modify the state directly as such a change will not re-render the component.

2. The **this.setState** operation is asynchronous - so it will not be executed immediately, to be sure that something will be executed after the state change, we can add a function like a second argument that will be executed immediately after the state change:
    this.setState({comment: 'Hello'}, () => { console.log(this.state) });
    More here: https://pl.reactjs.org/docs/faq-state.html

3. Methods in a class component can be written as arrow function to avoid writing bind to that method in the constructor. More here:
    https://pl.reactjs.org/docs/faq-functions.html

Life cycle

Mounting

constructor

↓

Updating

New props   setState()   forceUpdate()

↓   ↓   ↓

render

↓   ↓

React updates DOM and refs

↓   ↓

componentDidMount   **componentDidUpdate**

Unmounting

↓

componentWillUnmount

Life cycle

Life cycle methods can be used in such ways:

```
class Clock extends React.Component {
    state = { date: new Date() }

    componentDidMount() { }

    componentDidUpdate() { }

    componentWillUnmount() { }

    render() {
        return <h2>Time: {this.state.date.toLocaleTimeString()}.</h2>
    }
}
```

Life cycle methods in useEffect

```
class Example extends React.Component {
  componentDidMount() {
    console.log("I am mounted!");
  }
  render() {
    return null;
  }
}
```

```
function Example() {
  useEffect(() => {
    console.log("mounted");
  }, []);
  return null;
}
```

Life cycle methods in useEffect

```
class Example extends React.Component {
  componentDidMount() {
    console.log("I am here!");
  }

  componentDidUpdate() {
    console.log("I am here!");
  }

  render() {
    return null;
  }
}
```
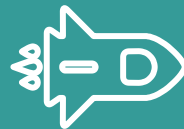
```
const Example = () => {
  useEffect(() => {
    console.log("I am here!")
  })
}
```

Life cycle methods in useEffect

```
class Example extends React.Component {
  componentWillUnmount() {
    console.log("Bye, bye...");
  }

  render() {
    return null;
  }
}
```

```
const Example = () => {
  useEffect(() => {
    return () => {
      console.log("Bye, bye...");
    };
  });
};
```

TASK

Rewrite the Game component into a class component.

Docs: https://pl.reactjs.org/docs/state-and-lifecycle.html

Forms

In React, HTML form elements work a little differently than other DOM elements. This is because the form elements maintain their internal state by themselves.

In React, however, the variable state of a component is usually stored in the component's state property. It is only updated using the setter functions.
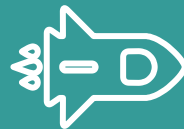
Forms

It is possible to combine the two solutions by establishing the React state as the "one source of truth". Then the React rendering component of the form also controls what happens inside the form when the user fills in the fields.

The input element of a form, controlled in this way by React, is called a "controlled component".

Forms

```
const NameForm = () => {
 const [name, setName] = useState('');

 const handleChange = (event) => {    setName(event.target.value);  }
 const handleSubmit = (event) => {
  event.preventDefault();
  alert('Given name: ' + name);
 }

 return (
   <form onSubmit={handleSubmit}>
    <label> Name:
     <input type="text" value={name} onChange={handleChange} />
    </label>
    <input type="submit" value="Send" />
   </form>
 );
}
```

TASK

Create a component with a form called **MyForm**, in which the user will be able to save data such as: name, age, select gender from the drop-down list, enter a comment.

After pressing the "send" button, display the provided information in the alert. The page should not be reloaded.

Forms

There are many external packages in React to help you work with forms:

- formik (https://formik.org/)
- react hook form (https://react-hook-form.com/)

They simplify the daily work with the forms, making it easier to manage the values in the form, validation and its internal state. Thanks to the library, you can easily:

- attach custom validations that we can attach as simply as native validations,
- attach asynchronous validations
- signal to the user which fields have already been filled in by him
- handle arrays and objects as form fields

TASK

Create a component from the previous job using one of the selected packages named **MyBoostedForm**.

Remember to install the package first.
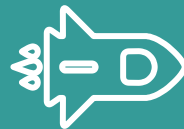
REACT UI COMPONENTS

UI Components

Working at Reack, we rarely write our own styling. We use UI Components - components that are responsible only for the appearance. There are many free packages available on the web with such components:

- material ui
- semantic ui
- react bootstrap

and many others:

https://www.codeinwp.com/blog/react-ui-component-libraries-frameworks/

TASK

Using one of the packages, change the **AboutMe** components (without Contact) and **Game** so that it uses components with previously prepared styles.

Remember to install the package of the given components first.

REACT ROUTER

React router

Server-side routing is a very common method of handling routing. However, it is not part of the React Router. How it's working?

1. The user clicks a link on the website.
2. A completely different page appears on the screen. The URL path is being updated.
3. Server-side routing causes the page to refresh because we make another request to the server, and it gives us completely new content to display.

If the new page still has the header and footer information on the screen, why should we reload it? However, the server doesn't know this, so it will reload them even though it doesn't have to.
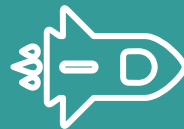
React router

Client-side routing relies on JS's internal handling of messages, which is rendered on the front end. How it's working?

1. The user clicks a link on the website.
2. The URL path is being updated. Only the required part of the page is updated on the screen.
3. Routing on the client side causes that the page does not refresh, but only performs subsequent queries to the server.

Thanks to this, we get a SPA, where we do not have to load many pages. Instead, we load the initial request with our initial HTML, CSS, and JS files from the server. We use client side routing to keep all the advantages of routing.

TASK

Using one of the UI components packages provided, display the navigation for our website, in which you add buttons: Home, About me, Forms, Game, Users.

React router

We have two types of packages for the react-router:

- react-router
- react-router-dom

react-router-dom is a superset of a react-router with components ready for use in the browser

Router

I have two types of routers:

- BrowserRouter - for dynamic queries
- HashRouter - for static requests

In most cases, we are interested in BrowserRouter due to SPA applications.

BrowserRouter is a component that must "wrap" (surround) our application or its part in which we want to use routing.

Routes, Route, Link, NavLink

To create a routing with react-router-dom, we need to know a few other components:

- **Routes** - changes the view based on the url, finds the appropriate Route component among its children and renders it
- **Route** - Takes a path as props and contains the content to be displayed when the path matches the url
- **Link** - creates a link in the application (<a> tag) that can change the url
- **NavLink** - a special Link component that can take appropriate styles as the path it leads to is active

React router dom

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";
function App() {
  return <Router>
      <nav> <ul>
         <li> <Link to="/">Home</Link> </li>
         <li> <Link to="/about">About</Link> </li>
         <li> <Link to="/users">Users</Link> </li>
      </ul> </nav>
      <Routes>
        <Route path="/about" element={<About />} />
        <Route path="/users" element={<Users />} />
        <Route path="/" element={<Home />} />
      </Routes>
   </Router>
}
```
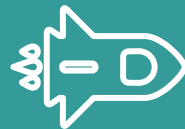
TASK
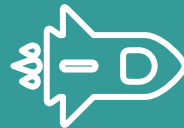
Add **BrowserRouter** to our application - "wrap" the application in index.js.

Add **Routes** in App component with the appropriate **Route**(s) that will lead to Home, About me, Game, Forms, Users components. Under the path Home, display the greeting "Welcome to Our Site!".

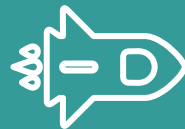Check if the view changes after entering the appropriate path.

TASK

Use the **Link** component in the navigation to lead us to the appropriate url after clicking on its element.

Check if clicking on the navigation element changes the url path and the view of the page.

TASK

Convert **Link** to **NavLink** and use **isActive** props in it.
https://reactrouter.com/en/6.4.5/components/nav-link

Check how the navigation changes after entering the path manually or after clicking on the navigation.
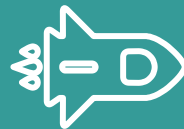
React router props

React router provides us with props that we can use in our application:
- params - url parameters
- location - represents the current url
- history - browser history that can be used for page navigation

To access them, we need to use the appropriate Hooks:
- useParams
- useLocation
- useNavigate

TASK

Create a new **Sign** component, use it in routing under the path */sign*.

Add a login form in it: email and password with the login button.

After clicking Login, redirect to the home page using **useNavigate** hook.

React router props

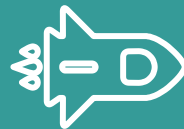We can also store data in the url. To read them, we need to properly define the path:

```
<Routes>
  <Route path="/user/:username" element={<User />} />
</Routes>
```

And then we can easily read those parameters with usage of hook.

```
function User(props) {
  const params = useParams();

  return <h1>Hello {params.username}!</h1>;
}
```

TASK

Create a new **UserDetails** component that will render to path: path = "*/users /:id*". It will display the following information based on the path:
"Hello user with id: <ID_FROM_URL>".

Start by creating a component and displaying it under the given path, then try to display the id given in the url.

TASK

Add in **UserDetails** downloading user data based on the id read from the hook. Display his full name.

You can download the data from the following address:
https://jsonplaceholder.typicode.com/users/1 , where id is the last digit.

TASK
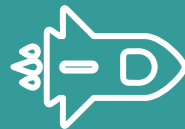
In the **Users** component, change the list items into clickable items that will be links to the path where **UserDetails** is located under the appropriate url.

Use the **Link** component from react-router-dom for this.

CODE SPLITTING

CODE SPLITTING

As the code volume of your application increases, so does its volume, and thus the loading time. To avoid the problem of too large a package, it is worth thinking about it at the beginning and starting to "split" your package. Code splitting is a feature supported by tools such as Webpack, which can create multiple dynamically reloading packages at the time of application code execution.

Sharing your application code will make it easier for you to use lazy loading to load only those resources that are currently required by the resource user, which can greatly improve the performance of your application. Although this way you will not reduce the total amount of code, you will avoid loading functionalities unnecessary for the user at a given moment, which will translate into a smaller amount of code to be downloaded at the start of the application.

IMPORT

The easiest way to introduce code breakdown into your application is to use the dynamic variant of the **import()** function.

```
import { add } from './math';



console.log(add(16, 26));
```

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

When Webpack comes across such a syntax, it automatically starts splitting the code in your application.

REACT LAZY

The React.lazy feature lets you render dynamically imported components like regular components.

Before:
import OtherComponent from './OtherComponent';

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

The above code will automatically load a package containing OtherComponent when a component is first rendered. React lazy takes a function as an argument that calls dynamic **import()**.

REACT SUSPENSE

The "lazy" component should be rendered inside **Suspense**, which allows us to display a replacement component (eg a load indicator) while loading, via the props fallback of a component that accepts any React element.

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

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

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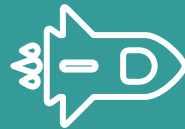
REACT SUSPENSE

Deciding where to split your application code can be confusing. You care about places that you choose places that will evenly distribute your packets, but you don't want to spoil the user experience.

Routes in your application are a good starting point. Most people who use the Internet are used to it taking some time to navigate between pages. Additionally, typically a large portion of the screen is re-rendered during such a transition. It can therefore be assumed that the user will not perform any actions on the interface while loading.

TASK

Take advantage of lazy loading by using React lazy and Suspense when loading components on a new path in the react router.

Add **Spinner** while the page is loading in the center of the screen.

CONTEXT

Context

Context allows data to be passed inside the component tree without having to be passed through the properties of each component.

In a typical React application, data is passed top-down (from parent to child) through properties. However, this may turn out to be too burdensome for some data and would have an impact on the application performance due to too many reloads. To solve this, data sharing contexts were created that can be considered "global" for the component tree, such as logged in user information, color scheme or preferred language.

Context

The context can be created using the method: React.createContext. This method creates a context object. When React renders a component that has subscribed to this context, it will pass the current value to it from the closest "Provider" above in the tree.

const MyContext = React.createContext(defaultValue);
The created context object consists of two elements: Provider ("supplier") and Consumer ("consumer").

Note! The defaultValue argument is only used when the component that reads from context has no provider over it.

Context

Each context object has its own Provider component, which allows reading components to subscribe to changes in this context.

<MyContext.Provider value={/* some value */}>

The value passed by the provider in the value property will go to the "consumers" of that context below in the tree. One provider may be connected to multiple consumers.

All consumers below the provider will be re-rendered whenever the value property changes.

Context

Example:

```
const TextColorContext = React.createContext('black');

const App = () => {
    const [textColor, setTextColor] = useState('black');

    return <>
            <TextColorContext.Provider value={textColor}>
                <Content />
            </TextColorContext.Provider>
            <button onClick={() => setTextColor('red')}>Red color</button>
            <button onClick={() => setTextColor(black)}>Black color</button>
        </>
}
```

Context

createContext also generates a **Consumer** component that subscribes to changes in context, that is, listens for changes in context.

A function must be a children of **Consumer** component. This function gets the current value from the context and returns a React node. The value argument passed to this function will be equal to the value property of the closest provider of this context above in the tree.

Context instead of using the **Consumer** component can also be read using the **useContext** hook.

If there is no provider above the component, the defaultValue value passed to **createContext**() is used.

Context

Example with hook:

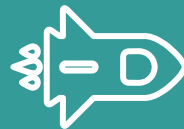```
const ThemedText = (props) => {
    const textColor = useContext(TextColorContext);

    return (

            This text has color from context

    );
}
```

# Context

Example with Consumer component:

```
const ThemedText = (props) => (
    <TextColorContext.Consumer>
      {value => <span style={{color: value}}>This text has color from context</span>}
    </TextColorContext.Consumer>
)
```

TASK

In our application, create a context called **LanguageContext** that will provide two languages - Polish and English (or other). There should be two buttons in the (below) navigation for changing the language, and each component in the application should change the text accordingly.

REACT

Final Battle!

Final battle

We will create a new application using all the knowledge we have known so far and we will try to build the appropriate application architecture.

The new application will consist in creating a website for a burger restaurant, where the user will be able to log in and check the menu.

Note! We will use the fetch method for all queries to the Firebase database. Firebase provides us REST api to read and manipulate data. To get to the data via HTTP protocol, we must always add **.json** to the end.

Example:
https://rest-api-b6410.firebaseio.com/burgers.json

Burgers menu

Create a table which will show menu of the restaurant.

1. Generate table with usage of UI component.
2. Fetch data and log it into the console. Convert data to array.
3. Use converted data in table.

Firebase Burgers

Name	Ingredients	Price
Hot	beef,bread,BBQ,jalapeno	22
Hawai	beef,bread,pineapple	20
Specjal	beef,bread,BBQ	15

Burgers details

Create a detail page, when the details of burger and its photo will be shown.

1. Create a new component under proper path.
2. Change burger name in menu component to Link and make a redirect to detailed page.
3. In detailed page fetch burger details and remember about loading and error states.

Firebase Burgers

Name	Ingredients	Price
Hot	beef,bread,BBQ,jalapeno	22
Hawai	beef,bread,pineapple	20
Specjal	beef,bread,BBQ	15

Burgers admin

Create admin panel.

1. Create the same table as it is for menu (we can just copy paste it).

2. Render edit and delete buttons.

3. At the bottom of table add button with plus (+), which after click opens modal with a form to add new burger.

Firebase Burgers

Name	Ingredients	Price		
Hot	beef,bread,BBQ,jalapeno	22	EDIT	DELETE
Hawai	beef,bread,pineapple	20	EDIT	DELETE
Specjal	beef,bread,BBQ	15	EDIT	DELETE

Burgers admin

Let's add interaction now!

1. Send POST request to add new burger in modal after submit.
2. Send DELETE request after click on button in table.
3. Remember to refresh the table data after each action.

Think about how we can achieve Edit with assumption that it needs to be inline.

Firebase Burgers

Name	Ingredients	Price		
Hot	beef,bread,BBQ,jalapeno	22	EDIT	DELETE
Hawai	beef,bread,pineapple	20	EDIT	DELETE
Specjal	beef,bread,BBQ	15	EDIT	DELETE

Sign in

Create sign in and sign up views!

1. Together let's firstly initialised firebase in app.
2. Create form to sign in.
3. Together let's add methods from firebase to form and generate sign up component.
4. Create context to store user data. Together we will add subscription to firebase.
5. Use the data in context to hide admin panel button on navigation and to switch between sign in and log out.

Thanks!

You can find me:

https://www.linkedin.com/in/kamil-richert/

https://github.com/krichert