React.3

🌟 review me

▼ When is a promise used?

A Promise is used whenever some piece of code completes in the future.

▼ What is resolving a promise?

Resolving a promise means performing an action when the promise has completed its work.

▼ How to resolve a promise?

To resolve a Promise, you need to **call .then() on it and pass a callback function**. That callback function will be called when the promise has completed its work successfully.

```
functionThatReturnsPromise().then(result => {
    console.log(result);
});
```

▼ When a promise fails, how do you catch errors?

catch(callback)

```
functionThatReturnsPromise()
.then(result => {
    console.log(result); // when successful
})
.catch(error => {
    console.error(error); // when there's an error
})
```

▼ What is the fetch API?

a browser API (meaning that it's a functionality available in the browser), that lets you make a network request to a server

▼ What is the most common use case of the fetch API?

when building (web) apps, send and receive data from the backend/API

▼ What does the fetch API return?

a promise which is why we have to resolve it

▼ fetch API GET example

1. the `fetch()` call returns a promise, which is why we resolve it with `.then()`.
2. the `response` we get back from `fetch` is a generic response, which is why we want to convert it to a JSON object by calling `response.json()`.
3. `response.json()` also returns a promise, which is why we need to resolve it **again** with `.then()`.

```
fetch(URL)
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

▼ Why use fetch API with React?

use the fetch API inside a React component and create a state variable out of the data that we get back from the backend/API

If you forget to specify the effect dependency, and you have a setState call inside the effect, then you will end up with an infinite loop. Why? setState initiates component re-render.

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

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://website.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data); // keep it for debugging
            // only call setState here if there is a dependencies array as a control else get infinite loop
            setUsers(data);
        });
    }, []);

    return null;
}
```

▼ Why is calling fetch inside a component is considered a side effect?

because fetch goes to the network (**outside of your component**), so it must be inside an effect

▼ How often do you want to call an effect that makes a fetch API call?

most of the time it will be just once when the component mounts aka 1 call

```
useEffect(() => {
        fetch("https://website.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
        });
    }, []);
```

▼ How to handle error: cannot read property X of undefined?

Add an if condition and return a different JSX (conditional rendering) to avoid this issue.

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

function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        fetch("https://react-tutorial-demo.firebaseio.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
            setUsers(data);
        });
    }, []);

    if (!users) {
        return null; // return nothing
    }

    return <h1>There are {users.length} users</h1>
```

▼ How can you return nothing from JSX?

return null

▼ How to handle rendering with JSX when data is undefined using logical && operator?

```
return <>
        <h1>Users</h1>
        <ul>
            {users && users.map(user => <li key={user.id}>{user.name}</li>)}
        </ul>
  </>
```

the expression users && users.map() will **short circuit** at users because users returns undefined

so before users has loaded, the App component will render:

```
<>
    <h1>Users</h1>
    <ul>
    </ul>
</>
```

▼ HTTP status codes 2xx

A status code of 200 (or any number between 200 and 299, often written as 2xx) is considered successful.

▼ HTTP status codes 4xx (client) and 5xx (server)

responses in the range of 400 to 499 (4xx) as well as 500 to 599 (5xx) mean that the fetch request did not complete successfully

▼ How to avoid setting state on an error object, assuming that the API returns an error object when things don't go as expected?

```
.then(data => {
    // only set the users when there's no error
    if (data && !data.error) {
        setUsers(data);
    }
});
```

▼ When do network errors happen?

when the connection breaks or when the user is offline

▼ How to handle network errors in fetch requests?

catch()

```
fetch("https://website.com/users.json")
.then(response => response.json())
.then(data => {
    console.log(data);
})
.catch(error => {
    console.log(error); // or console.error(error)
});
```

▼ How to handle showing a loading icon when fetching data from an API?

```
// index.js

import React, {useEffect, useState} from "react";
import {render} from "react-dom";
import Loader from './Loader.js';

function App() {
    const [users, setUsers] = useState();
    const [isLoading, setIsLoading] = useState(true);
```

```
    useEffect(() => {
        fetch("https://website.com/users.json")
        .then(response => response.json())
        .then(data => {
            console.log(data);
            if (data) {
                setUsers(data);
            }
        })
        .catch(error => {
            console.error(error);
        })
        .finally(() => {
            setIsLoading(false); // stop the loader
        });
    }, []);

    return <>
        {isLoading && <Loader />}
        {users && <h1>Users</h1>}
        <ul>
            {users && users.map(user => <li key={user.id}>{user.name}</li>)}
        </ul>
    </>;
}

render (<App />, document.querySelector("#react-root"));
```

```
// Loader.js

import React from "react";

export default function Loader() {
    return <svg className="spinner" width="65px" height="65px" viewBox="0 0 66 66" xmlns="http://www.w3.org/2000/svg">
   <circle className="path" fill="none" strokeWidth="6" strokeLinecap="round" cx="33" cy="33" r="30"></circle>
</svg>;
}
```

▼ Why disable a button while loading?

This is common practice to avoid the user from repeatedly clicking on a button that has already started a fetch request

```
<button disabled={isLoading}>Click me</button>
```

▼ If you need to run a fetch call based on a state value that can change, it's best to run the fetch request inside what?

a useEffect with a dependency on that state variable

```
import React, {useState, useEffect} from "react";
import {render} from "react-dom";

function CurrencyConversion() {
    const [currency, setCurrency] = useState("");
    const [rate, setRate] = useState("");
    const [isLoading, setIsLoading] = useState(false);

    useEffect(() => {
        if (currency) {   // fetch call won't worry if no currency
            setIsLoading(true);
            fetch(`https://website.com/currencies/${currency}.json`)
            .then(response => response.json())
            .then(data => {
                if (data) {
                    setRate(data);
                }
            })
            .catch(error => console.log(error))
            .finally(() => {
                setIsLoading(false);
            });
        }
```

```
    }, [currency]);

    function handleCurrencyChange(event) {
        setCurrency(event.target.value);
    }

    return <>
        <h2>Currency rates</h2>
        <select onChange={handleCurrencyChange} disabled={isLoading}>
            <option value="">Select a currency</option>
            <option value="usd">USD</option>
            <option value="eur">EUR</option>
            <option value="cad">CAD</option>
        </select>
        <h3>1 {currency.toUpperCase()} = {rate} USD</h3>
    </>;
}

render(<CurrencyConversion />, document.querySelector("#react-root"));
```

▼ What is the most common cause of stack overflow error?

Deep or infinite recursion, in which a function calls itself so many times that the space needed to store the variables and information associated with each call is more than can fit on the stack.

▼ How to use async/await (syntactic sugar on top of promises) when using useEffect? (example)

To use await, we need to be inside an async function.

We add an async immediately invoked function expression inside useEffect.

Use try/catch to handle errors

```
function App() {
    const [users, setUsers] = useState();

    useEffect(() => {
        (async () => {
          try {
            const response = await fetch("https://website.com/users.json")
            const data = await response.json()
            setUsers(data);
        } catch (error) {
            console.log(error);
        }
        })();
    }, []);

    return null;
}
```

▼ Async/await with events (example)

```
import React from "react";

function App() {
    async function handleButtonClick() {
        // use fetch with await here
        const response = await fetch("https://website.com/users.json")
        const data = await response.json();
    }

    return <button onClick={handleButtonClick}>Load data</button>;
}
```

▼ fetch POST is used for what?

to send data to an API

▼ fetch POST (example)

You need to specify the method: "POST" and often have to send the body: JSON.stringify(dataObjectHere).

It's also a best practice to set the header "Content-Type": "application/json".

```
fetch("https://website.com/grades.json", {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify({grade: 50}) // converts JSON to string
})
.then(response => response.json())
.then(data => {
    console.log(data);
});
```

▼ fetch POST in React (example)

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

function App() {
    const [number, setNumber] = useState(0);

    function handleFormSubmit(event) {
        event.preventDefault();

        fetch("https://website.com/grades.json", {
            method: "POST",
            headers: {
                "Content-Type": "application/json"
            },
            body: JSON.stringify({grade: number})
        })
        .then(response => response.json())
        .then(data => {
            console.log("Grade added");
            console.log(data);
        });
    }

    return <form onSubmit={handleFormSubmit}>
        <input type="number" value={number} name="grade" onChange={event => setNumber(event.target.value)} placeholder="Enter the grade
        <input type="submit" />
        </form>
    ;
}
```

Notice how {grade: number} is sending the state variable called number which is being set in the onChange handler with the setNumber(event.target.value) call.

▼ Why use PUT method?

to **update** an existing value

▼ What is a custom hook?

A custom hook is a JavaScript function whose name starts with use.

This function is able to call other hooks (for example useEffect, useState, etc.).

The goal of custom hooks is to be able to reuse them in several components.

It's also much cleaner to have them in a separate file.

▼ Why build a custom hook?

- easily reuse code in several components, which makes it easier to share common functionality.
- build abstractions which makes some complicated logic easier to read.

▼ Custom hook (example)

```
import {useEffect} from "react";

function useAnalyticsEvent() {
    useEffect(() => {
        gtag("event", "component_rendered");
    }, []);
}

function App() {

    useAnalyticsEvent(); // use hook for App Component

    return <h1>My App</h1>
}

render(<App />, document.querySelector("#react-root"));
```

This could be useful for sending an analytics event (for example using Google Analytics).

The gtag function is coming from Google Analytics library.

The point of this example is to show you that you can send the analytics event only once after the component has rendered the first time. That's because we're using useEffect(..., []).

▼ Custom hook file and import (example)

```
// useHelloWorld.js
import {useEffect} from "react";

export default function useHelloWorld() {
    useEffect(() => {
        console.log("Hello World!");
    });
}
```

```
import React from "react";
import useHelloWorld from "./useHelloWorld.js";

function App() {
    useHelloWorld();

    return <h1>My App</h1>;
}
```

▼ What does document represent?

The Document interface represents any web page loaded in the browser and serves as an entry point into the web page's content, which is the DOM tree.

The DOM tree includes elements such as <body> and <table>, among many others.

▼ Custom hook with state (example)

For example, for our supermarket web app, we will often need to have a counter that keeps track of how many items the user has added to their basket.

Now this custom hook needs to expose the counter state as well as the increment and decrement functions, so that they can be used by the outside component. How? Through returning an object and using object destructuring.

Why? When you have more than 2 items being returned by a custom hook is that the order won't matter as long as you use the same keys' names. (less error prone)

```
// useProductCounter.js
import {useState} from "react";

export default function useProductCounter() {
    const [counter, setCounter] = useState(0);

    function increment() {
```

```
        setCounter(prevCounter => prevCounter + 1);
    }

    function decrement() {
        setCounter(prevCounter => {
            if (prevCounter > 0) {
                return prevCounter - 1;
            }
            return 0;
        });
    }

    /* we need to return the counter and the 2 functions */
    return {
      counter: counter,
      increment: increment,
      decrement: decrement
    };
}
```

Inside the component (object destructuring).

```
function App() {
    const {counter, increment, decrement} = useProductCounter();

     return <>
        <h2>{counter}</h2>
        <button onClick={increment}>+</button>
        <button onClick={decrement}>-</button>
    </>;
}
```

Now that our custom hook is returning the increment and decrement functions, we need to update the JSX's onClick handlers to use the new increment and decrement functions.

▼ custom hook for fetch API (example)

```
// useFetch.js

import { useState } from "react";

export default function useFetch(baseUrl) {
  const [loading, setLoading] = useState(true);

  function get(url) {
    return new Promise((resolve, reject) => {
      fetch(baseUrl + url)
        .then(response => response.json())
        .then(data => {
          if (!data) {
            setLoading(false);
            return reject(data);
          }
          setLoading(false);
          resolve(data);
        })
        .catch(error => {
          setLoading(false);
          reject(error);
        });
    });
  }

  function post(url, body) {
    return new Promise((resolve, reject) => {
      fetch(baseUrl + url, {
          method: "post",
          headers: {
            "Content-Type": "application/json",
          },
          body: JSON.stringify(body)
      })
```

```
        .then(response => response.json())
        .then(data => {
          if (!data) {
            setLoading(false);
            return reject(data);
          }
          setLoading(false);
          resolve(data);
        })
        .catch(error => {
          setLoading(false);
          reject(error);
        });
    });
  }

  return { get, post, loading };
};
```

```
// index.js

import React, {useEffect} from "react";
import useFetch from "./useFetch.js";

function App() {
    const {get, loading} = useFetch("https://api.website.com/");

    useEffect(() => {
        get("users.json").then(data => {
            console.log(data);
        })
        .catch(error => console.error(error));
    }, []);

    return (<>
        <h2>{loading ? "Loading..." : ""}</h2>
    </>);
}
```

```
// POST example

// index.js

import React, {useState, useEffect} from "react";
import {render} from "react-dom";
import useFetch from './useFetch.js';

function GradeForm() {
    const [grade, setGrade] = useState(0);
    const {post} = useFetch("https://api.website.com/");

    function handleFormSubmit(event) {
        event.preventDefault();
        post("grades", {
            grade: grade
        }).then(data => {
            console.log(data);
        }).catch(error => console.error(error));
    }

    return (
        <form onSubmit={handleFormSubmit}>
            <input type="number" value={grade} name="grade" onChange={event => setGrade(event.target.value)} placeholder="Enter the gra
            <input type="submit" />
        </form>
    );
}

render(<GradeForm />, document.querySelector("#react-root"));
```

▼ What is a React ref and why use it?

Think of it as an #id. It holds a **reference** or ref to a rendered DOM element.

Why? Allows you to find a specific element rendered to the DOM with React.

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

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

    return <input ref={inputRef} type="text" />;
}
```

▼ Why should document.querySelector() never be used **inside** React?

It may not work if elements are not yet rendered.

React creates React elements which make up the Virtual DOM and are later on rendered to the browser's real DOM

▼ How to focus an input element (using ref attribute)?

```
import React, {useRef, useEffect} from "react";

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

    useEffect(() => {
        inputRef.current.focus();
    }, []);

    return <input ref={inputRef} type="text" />;
}
```

▼ What is the ref.current key?

When you call useRef(), React will create an object containing the key current.

When you assign the ref to the JSX element, React will assign the current key to the rendered DOM element.

This means that every time you want to **access the element you're referencing**, you have to use the .current key.

```
// object created using useRef
{
    current: <input type="text"/>
}


// later
inputRef.current.focus();
```

▼ inputRef.current will only return the DOM element when?

after the component has been rendered to the DOM

This is why you will often only use the ref inside either an event handler or a useEffect call. That's because both of these methods run after the component has been rendered to the DOM.

▼ When to use refs?

1. Imperative DOM operations. When you find that the only way to make this work, is by **calling a certain DOM method**. For example `.play()` or `.focus()`.
2. **Initializing DOM libraries**. For example, Mapbox, or a carousel plugin.

▼ Using refs (use cases)

- Focusing an element.
- Playing a video (`videoElement.play()`), pausing a video, etc.
- Selecting text from the DOM.

- Initializing DOM libraries (will be covered in the next project).

▼ Imperative vs. Declarative

Imperative: telling the computer **how** to do something

Declarative: telling the computer **what** you would like to happen, and lets the computer figure out how to do it

▼ What does **context** mean in software?

relevant information

▼ What is the use of Context in React?

The ideal use case for Context is when you want to make global data accessible to several components in your app.

For example, the theme of your app, or the locale (language selected), or the chosen currency, or the user profile details (id, image, bio).

▼ How to create context in React? (steps)

Step 1: Create Context

- Create a new Context.

```
import React, {createContext} from "react";

const ThemeContext = createContext();
```

Step 2: Create a Provider

- Every context needs a Provider.
- In our case, we're creating a ThemeContext and we'd like to provide the theme (for example "dark" or "light").
- ThemeProvider is a React component that receives props and renders the props.children wrapped with <ThemeContext.Provider>
- The ThemeContext.Provider receives a value which is the value that all the children will be able to receive.
- In a nutshell, this provider will wrap our <App /> component and give it the value="dark" so that we can read the theme from any component.

```
import React, {createContext} from "react";

const ThemeContext = createContext();

function ThemeProvider(props) {  // THIS IS A COMPONENT
  const theme =  "dark";

  return (
    <ThemeContext.Provider value={theme}>  // provider for the defined context
      {props.children}
    </ThemeContext.Provider>
  );
};
```

Step 3: Export the Context & the Provider

```
export { ThemeContext, ThemeProvider };
```

▼ What is a Provider?

A provider will provide the data to the context.

The provider will wrap our components and provide them with the value (some global data).

```
<MyContext.Provider value={data_to_provide_to_context}>
</MyContext.Provider>
```

▼ How to use Context (steps)?

Step 1. Wrap components with the Provider

- To make the context available in your app, you have to wrap the components with the Provider.

```
// index.js
import React from "react";
import {ThemeProvider} from "./ThemeContext.js";

function App() {
    return (<ThemeProvider>
        <Nav />
    </ThemeProvider>);
}
```

Step 2. Use the context inside components

- Any component nested inside the ThemeProvider can now access the context by importing the ThemeContext and passing it to the useContext() hook.
- Notice how we import the useContext hook and we pass to it the ThemeContext. This will allow you to consume the value that we passed to the provider in the previous step, which in this example is "dark".

```
// Button.js
import React, {useContext} from "react";
import {ThemeContext} from "./ThemeContext.js";

function Button(props) {
    const theme = useContext(ThemeContext);
    console.log(theme); // "dark"
    return <button>{props.children}</button>;
}
```

▼ What does using Context in React help you avoid?

The benefit of this is that we don't have to pass the props from the top-most component down to all the children component.

▼ You can also pass an object not just a string to a Provider value (example)

The state update caused by context.toggleTheme() will end up notifying (re-rendering) all the components that use this ThemeContext.

```
import React, { createContext, useState } from "react";

const ThemeContext = createContext();

function ThemeProvider(props) {
  const [theme, setTheme] = useState("dark");

  function toggleTheme() {
      if (theme === "dark") {
        setTheme("light") // state update
      } else {
        setTheme("dark") // state update
      }
  }

  const value = {
    theme: theme,
    toggleTheme: toggleTheme
  }
```

```
  return (
    <ThemeContext.Provider value={value}>
      {props.children}
    </ThemeContext.Provider>
  );
};

export { ThemeContext, ThemeProvider };
```

With that context we can now update the theme from anywhere in the app, as long as we use this ThemeContext.

```
import React, {useContext} from "react";
import {ThemeContext} from "./ThemeContext.js";

function App() {
    const context = useContext(ThemeContext);
    console.log(context); // {theme: "dark", toggleTheme: Function}

    return <button onClick={() => context.toggleTheme()}>Toggle Theme</button>;
}
```

▼ Example of context and provider and toggling data passed to provider

```
// index.js
import React, {useContext} from "react";
import {render} from "react-dom";
import Navbar from "./Navbar.js";
import Main from "./Main.js";
import Button from "./Button.js";
import Footer from "./Footer.js";
import { ThemeProvider, ThemeContext } from './ThemeContext.js'

// App component is wrapped with <ThemeProvider /> (check render() call below)
function App() {
    const context = useContext(ThemeContext);

    return (
        <>
            <Button onClick={context.toggleTheme}>Toggle theme</Button>
            <hr />
            <Navbar />
            <Main />
            <Footer/>
        </>
    );
}

render(<ThemeProvider ><App /></ThemeProvider>, document.querySelector("#react-root"));
```

```
// Button.js
import React, {useContext} from "react";
import clsx from "clsx";
import { ThemeContext } from './ThemeContext';

export default function Button(props) {
    const context = useContext(ThemeContext);

    const classes = clsx({
        dark: context.theme === "dark"
    });

    return <button className={classes} onClick={props.onClick}>{props.children}</button>;
}
```

▼ What is clsx?

clsx is a tiny utility for constructing className strings conditionally, out of an object with keys being the class strings, and values being booleans.

▼ Why should you wrap an <App /> component?

When you attempt to use the context in the same component where you render the provider, you won't be able to access the value of the context because the current component is NOT wrapped by that provider.

Solution: use an App wrapper where you wrap your <App /> component with the provider.

```
import React, {useContext} from "react";
import {SomeContext, SomeProvider} from "./SomeContext.js";
import Blog from "./Blog.js";

function App() {
    const context = useContext(SomeContext);
    console.log(context);

    return <Blog />
}

function AppWrapper() {
    // Wrap the <App /> component with the provider
    return (<SomeProvider>
        <App />
    </SomeProvider>);
}

render(<AppWrapper />, document.querySelector("#react-root"));
```

▼ What does JSX convert a component into?

a React.createElement(Component, {}) call

▼ How to create different types of the same element? (JSX Dot Notation example)

It's used in UI Libraries where the library would export all the buttons under a single object, for example, called Buttons and then you can choose the type of button by using Buttons.Default or Buttons.Outline.

```
const Buttons = {
    Default: function(props) {
        return <button className="btn-default">{props.children}</button>;
    },
    Outline: function(props) {
        return <button className="btn-outline">{props.children}</button>;
    }
}
```

```
// Then you can use the components like so:

<Buttons.Default>Login</Buttons.Default>
<Buttons.Outline>Register</Buttons.Outline>
```

```


// The reason why this works is because they will be converted to
// React.createElement calls as following:

React.createElement(Buttons.Default, {}, "Login");
React.createElement(Buttons.Outline, {}, "Register");
```

▼ Why is it that whenever you have an expression in JSX, React will escape all the HTML entities?

to prevent XSS (cross-site scripting) injection

```


import React from "react";

function Footer() {
```

```
    const text = "<strong>All rights reserved &copy;</strong>";

    return <div>{text}</div>;
}

// won't be All rights reserved ©
// instead will be All rights reserved &copy;
```

▼ What is Cross-Site Scripting (XSS) Injection?

a security vulnerability when the user is able to inject your page with their own code (HTML, CSS, and/or JavaScript)

▼ XSS example

If a website allows you to write your own scripts that get rendered on the page, this will be a massive security breach as you will be able to read people's cookies, act on their behalf, redirect them to other websites, and do so many other things.

```
// given
<script>alert("You got hacked!")</script>

// React escapes HTML elements
// the "comment" above becomes:
&lt;script&gt;alert("You got hacked!")&lt;/script&gt;.
```

By escaping the < and > characters, the browser is able to render the comment as text rather than a script.

▼ How to never trust user input?

If the expression you're embedding is coming from the end-user (for example, a comment, a first name, a last name, an email, etc.), you should never dangerously set inner HTML.

▼ How to set inner HTML when the input comes from trusted source aka your company?

We removed the {text} expression and instead added a prop called dangerouslySetInnerHTML which accepts an object {__html: text} which is wrapped with an expression {} (that's why we have {{}} one for the object and one for the expression).

Inside this object, you have to set the key __html (that's 2 underscores).

```
import React from "react";

function Footer() {
    const text = "<strong>All rights reserved &copy;</strong>";

    return <div dangerouslySetInnerHTML={{__html: text}}></div>;
}
```

▼ What is the <React.StrictMode /> component?

it only activates additional checks and warnings for its children when React is in development mode

React.StrictMode can be quite useful in detecting legacy code that might break in a future version of React, especially when you have an old React project that you're currently maintaining.

```
import React from "react";
import {render} from "react-dom";

render(<React.StrictMode>
    <App />
</React.StrictMode>, document.getElementById('root'));
```

▼ Typical/vanilla way of writing CSS in an app?

having an index.css file where you create classes and then use those classes in the className prop

▼ What is styled components?

a package that uses a technique called CSS-in-JS. Where the CSS is written in JavaScript.

https://styled-components.com/

▼ What are CSS modules?

a package that allows you to import your styles as an object of class names which you can then use in your React component

https://github.com/gajus/babel-plugin-react-css-modules

▼ UI Library: Material-UI package for React components

https://material-ui.com/

▼ What is a synthetic event?

A synthetic event contains the original/native browser event and adds some additional functionalities (e.g. event.currentTarget() and event.target)

When you use event handlers in React, such as onClick={}, onChange={}, etc. **React will wrap the native browser event with a synthetic event**.

▼ How to access the native browser event?

If you need to access the original event, you can do so by accessing the nativeEvent property on the event object.

```
import React from "react";

function Button() {
    function handleButtonClick(event) {
        console.log(event.nativeEvent);
    }

    return <button onClick={handleButtonClick}>Click me</button>;
}
```

▼ What is event pooling (not in latest React - in React 16)?

React creates this synthetic event for performance reasons. **If you click a button 10 times, instead of creating 10 different event objects, React will only create one event and reuse it (with updated values) the next time you click on it.**

Since React is re-using this event object, if you access any property or method of the event variable in a non-synchronous way, you will get null and a warning.

A non-synchronous way means that you try to access it after the event has occurred. For example, inside a setTimeout() or inside the .then() of a fetch call.

▼ How to access an event in a non-synchronous way?

you will need to persist the synthetic event

You can: Store the property in a variable

- As long as you store the property in a synchronous way (meaning, before you have the setTimeout or fetch call), you can access that value again in a non-synchronous way.

```
import React from "react";

function Button() {
    function handleButtonClick(event) {
        const target = event.target;
        setTimeout(() => {
            console.log(target);
        }, 1);
    }

    return <button onClick={handleButtonClick}>Click me</button>;
}
```

You can: call the .persist() method on the Synthetic event object

- This method is provided by React and it allows you to remove the synthetic event from the pool which means that you will be able to access it in a non-synchronous way.

```
import React from "react";

function Button() {
    function handleButtonClick(event) {
        event.persist(); // persist the event
        setTimeout(() => {
            console.log(event.target);
        }, 1);
    }

    return <button onClick={handleButtonClick}>Click me</button>;
}
```

▼ What is React Router do?

it lets us declaratively define routes that are also linked to a certain page URL

▼ What is React Router?

a library that makes creating routes in React easier

The react-router-dom package uses named exports to export its functionality.

▼ React Router (example)

```
// index.js

import React from "react";
import {BrowserRouter, Switch, Route, Link} from "react-router-dom";
import {render} from "react-dom";
import About from "./About.js";
import Home from "./Home.js";

function App() {
  return (
    <BrowserRouter>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
          </ul>
        </nav>

        <Switch>
          <Route exact path="/about">
            <About />
          </Route>
          <Route exact path="/">
            <Home />
          </Route>
        </Switch>
    </BrowserRouter>
  );
}

render(<App />, document.querySelector("#react-root"));
```

▼ What is BrowserRouter?

The <BrowserRouter /> component is a router that uses the HTML5 history API.

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

function App() {
    return <BrowserRouter>
        <div>The rest of your app goes here</div>
    </BrowserRouter>;
}
```

▼ What does BrowserRouter keep your User Interface in sync with?

the URL, which means that the URL will always reflect the current route that is rendered

▼ What is the pushState method from the HTML5 history API?

it allows changing the URL of the page without triggering a browser refresh

▼ What is Route?

The <Route /> component will conditionally render a component when the path prop matches the current URL in the browser.

Don't forget the exact prop on the <Route /> component.

```
// The <About /> component will only render when the browser URL
// matches the path /about.

<Route exact path="/about">
    <About />
</Route>
```

▼ What is npx?

Run a command from a local or remote npm package.

▼ What is a package manager?

A package manager, such as Yarn or npm. It lets you take advantage of a vast ecosystem of third-party packages, and easily install or update them.

▼ What is a bundler?

A bundler, such as **webpack** or **Parcel**. **It lets you write modular code and bundle it together into small packages to optimize load time.**

▼ What is a compiler?

a program that converts instructions into a machine-code or lower-level form

A compiler such as Babel. It lets you write modern JavaScript code that still works in older browsers.

▼ What is the best way to start building a new single-page application in React?

Create React App

It sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production. You'll need to have Node >= 10.16 and npm >= 5.6 on your machine.

```
npx create-react-app my-app
cd my-app
npm start
```

▼ What are error boundaries in React?

Error boundaries are React components that catch JavaScript errors anywhere in their child component tree, log those errors, and display a fallback UI instead of the component tree that crashed.