React : JavaScript library for building user interfaces

When our data changes, React will efficiently update and re-render our components.
Components are the core building block of React apps
react is a component tree - having one root component ("App") and then an potentially infinite amount of nested child components.

JSX is just syntactic sugar for JavaScript, allowing you to write HTMLish code instead of nested React.createElement(...) calls.

Functional components (also referred to as "presentational", "dumb" or "stateless" components)

class-based components (also referred to as "containers", "smart" or "stateful" components)

→ only changes in props and/ or state trigger React to rerender your components and potentially update the DOM in the browser

props allow you to pass data from a parent (wrapping) component to a child (embedded) component.

The `render` method returns a *description* of what you want to see on the screen.

you can put *any* JavaScript expressions within braces inside JSX.

createElement()

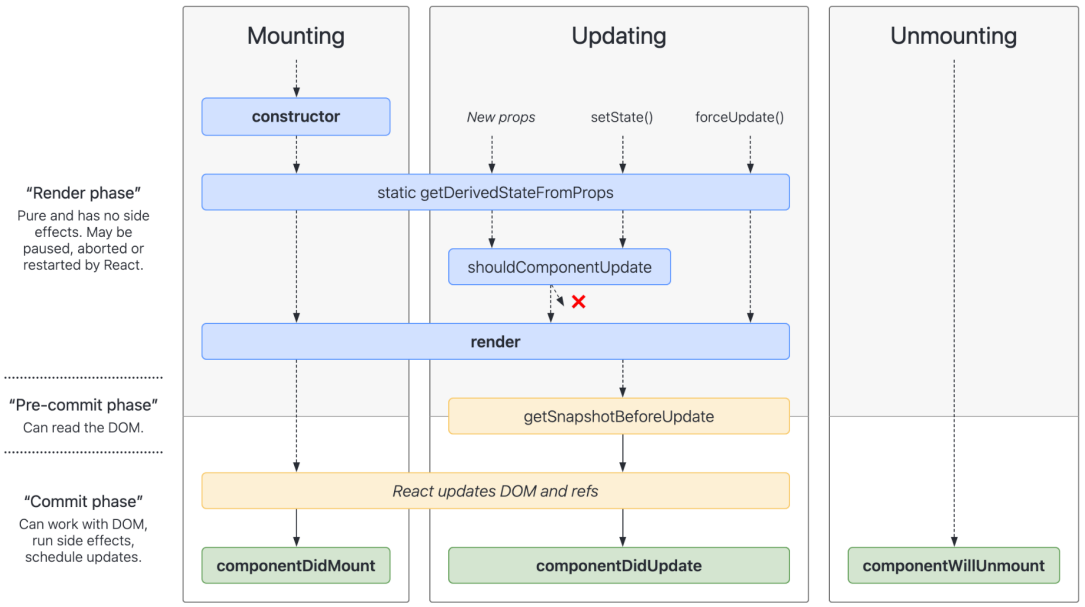
Create and return a new React element of the given type. The type argument can be either a tag name string (such as `'div'` or `'span'`), a React component type (a class or a function), or a React fragment type.

Code written with JSX will be converted to use `React.createElement()`

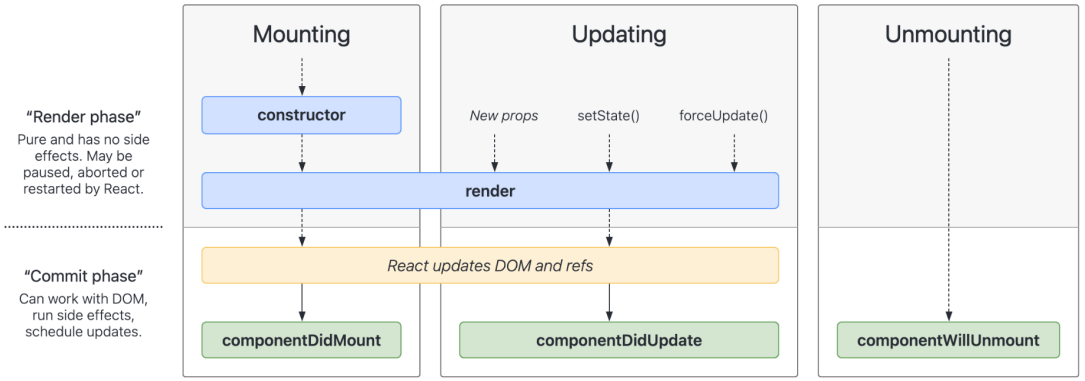
→ `this.state` should be considered as private to a React component that it's defined in

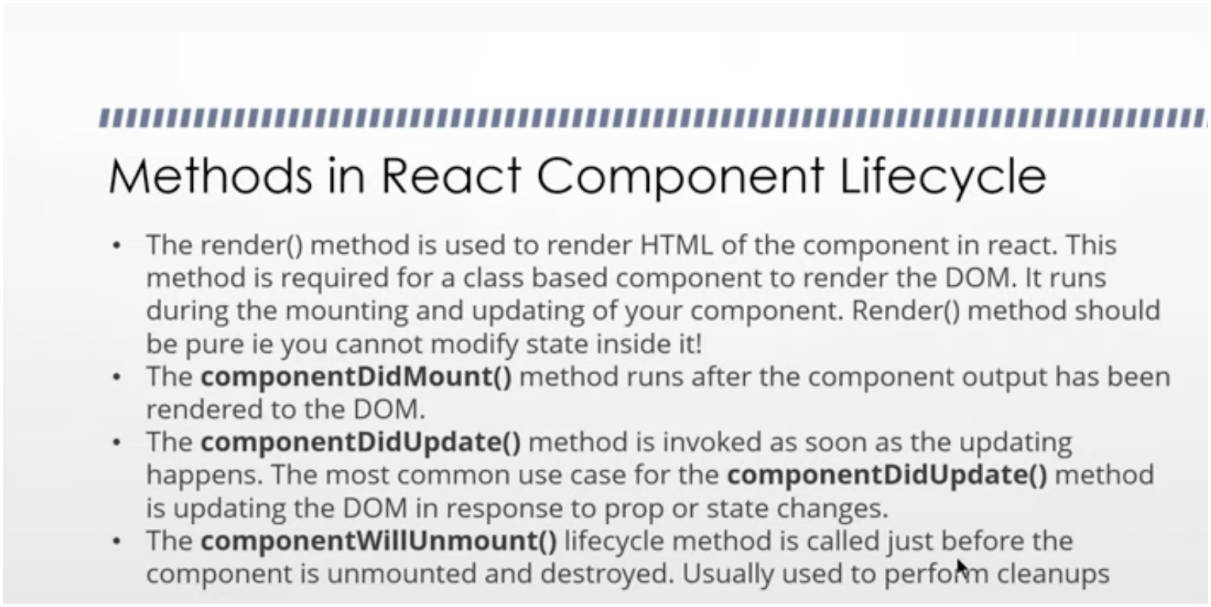
→To collect data from multiple children, or to have two child components communicate with each other, you need to declare the shared state in their parent component instead. The parent component can pass the state back down to the

children by using props; this keeps the child components in sync with each other and with the parent component.



https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/





`ComponentDidMount()` is invoked immediately after a component is mounted (inserted into the tree). Initialization that requires DOM nodes should go here. If you need to load data from a remote endpoint, this is a good place to instantiate the network request.

`componentDidUpdate()` is invoked immediately after updating occurs. This method is not called for the initial render.Use this as an opportunity to operate on the DOM when the component has been updated. This is also a good place to do network requests as long as you compare the current props to previous props

HOOKS

let you use state and other React features without writing a class

Rules of hooks

1. Don’t call Hooks inside loops, conditions, or nested functions

2. Only Call Hooks from React Functions. Don't call Hooks from regular JavaScript functions

Predefined Hooks

1. **`useState`** returns a pair: the *current* state value and a function that lets you update it. You can call this function from an event handler or somewhere else. It's similar to `this.setState`

The only argument to `useState` is the initial state. the state here doesn't have to be an object — although it can be if you want. The initial state argument is only used during the first render.

useState does not merge and update the object automatically while setstate do To merge the object use spread operator.

you can pass previous state to useState to update the value

The Effect Hook, `useEffect`, adds the ability to perform side effects from a function component. It serves the same purpose as `componentDidMount`, `componentDidUpdate`, and `componentWillUnmount` in React classes, but unified into a single API

useEffect run after every render of the component i*ncluding* the first render. Effects are declared inside the component so they have access to its props and state

eg

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

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


 // Similar to componentDidMount and componentDidUpdate:
 useEffect(() => {
   // Update the document title using the browser API
   document.title = `You clicked ${count} times`;
 });
```

```
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Second argument of useEffect is condition when useEffect will work
→ to run use efect only once (not every render) pass [] as second argument in useEffect

eg

```
functionHookMoue() {
const [x, setX] = useState(0)
const [y, setY] = useState(0)
const logMousePosition = e => {
console.log('Mouse event')
setX(e.clientX)
setY(e.clientY)
}
useEffect(() => {
console.log('useEffect called')
window.addEventListener('mousemove', logMousePosition)
return () => {
console.log('Component unmounting code')
window.removeEventListener('mousemove', logMousePosition)}
}, [])
return (
<div>
Hooks - X - {x} Y - {y}
</div>
)
}
```

→ use effect can be use for api call using axios which return a promise

Context help to pass data through component tree without passing manually at every level

```jsx
import { useState, createContext, useContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext();

function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
  );
}

function Component3() {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 />
    </>
  );
}

function Component4() {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 />
    </>
  );
}

function Component5() {
```

```
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

useReducer : State management. related to reducer function

useReducer(reducer, initialState)

reduce vs useReducer

reduce in JavaScript	useReducer in React
array.**reduce**(***reducer***, initialValue)	**useReducer**(***reducer***, initialState)
singleValue = ***reducer***(accumulator, itemValue)	newState = ***reducer***(currentState, action)
reduce method returns a single value	**useReducer**returns a pair of values. [newState, dispatch]

```
import { useState, createContext, useContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext();
```

```jsx
function Component1() {
  const [user, setUser] = useState("Jesse Hall");

  return (
    <UserContext.Provider value={user}>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 />
    </UserContext.Provider>
  );
}

function Component2() {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 />
    </>
  );
}

function Component3() {
  return (
    <>
      <h1>Component 3</h1>
      <Component4 />
    </>
  );
}

function Component4() {
  return (
    <>
      <h1>Component 4</h1>
      <Component5 />
    </>
  );
}

function Component5() {
  const user = useContext(UserContext);

  return (
    <>
```

```
      <h1>Component 5</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Component1 />);
```

useCallBack :A hook that will return a memorised version of callback function that only changes if one of dependencies has changed.

It is used when passing callback to optimised child component that rely on reference equality to prevent unnecessary renders

```
import { useState, useCallback } from "react";
import ReactDOM from "react-dom/client";
import Todos from "./Todos";

const App = () => {
  const [count, setCount] = useState(0);
  const [todos, setTodos] = useState([]);

  const increment = () => {
    setCount((c) => c + 1);
  };
  const addTodo = useCallback(() => {
    setTodos((t) => [...t, "New Todo"]);
  }, [todos]);

  return (
    <>
      <Todos todos={todos} addTodo={addTodo} />
      <hr />
      <div>
        Count: {count}
        <button onClick={increment}>+</button>
      </div>
    </>
  );
};
```

useMemo

The main difference is that `useMemo` returns a memoized value and `useCallback` returns a memoized function.

Controlled input : accept its current value as prop and callback function to changes its value

useRef

The `useRef` Hook allows you to persist values between renders.

It can be used to store a mutable value that does not cause a re-render when updated.

The useRef returns a mutable ref object. This object has a property called .current. The value is persisted in the refContainer.current property. These values are accessed from the current property of the returned object. The .current property could be initialised to the passed argument initialValue e.g. useRef(initialValue). The object can persist a value for a full lifetime of the component.

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

function App() {

// Creating a ref object using useRef hook
const focusPoint = useRef(null);
const onClickHandler = () => {
```

```
      focusPoint.current.value =
      "The quick brown fox jumps over the lazy dog";
      focusPoint.current.focus();
};
return (
      <Fragment>
      <div>
            <button onClick={onClickHandler}>
            ACTION
            </button>
      </div>
      <label>
      Click on the action button to
      focus and populate the text.
      </label><br/>
      <textarea ref={focusPoint} />
      </Fragment>
);
};
```

CustomHook: a js function whose name start with use . it can call other hooks as well.

When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.

```
import { useState, useEffect } from "react";
const useFetch = (url) => {
  const [data, setData] = useState(null);
 useEffect(() => {
fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);
```

```
  return [data];
};
```

//index.js

```
import { useState, useEffect } from "react";
const useFetch = (url) => {
  const [data, setData] = useState(null);
 useEffect(() => {
fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};
```

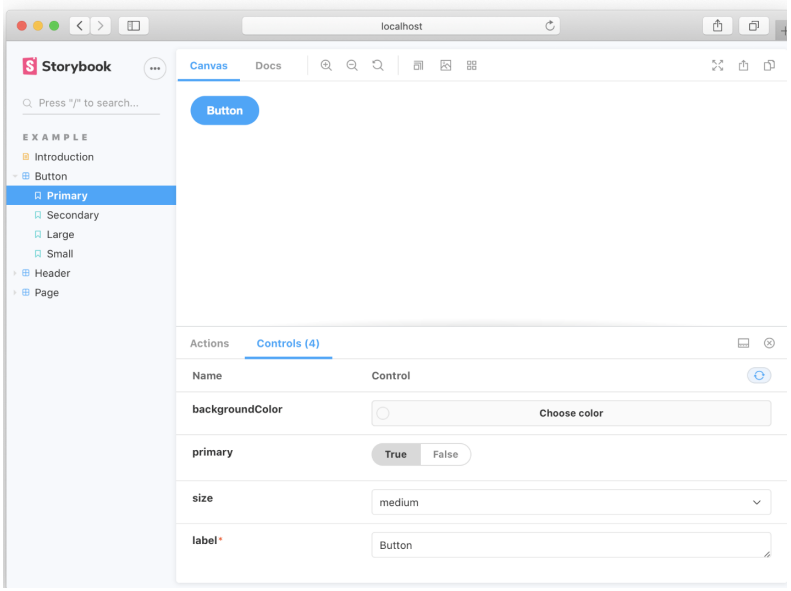
HOC: a higher-order component is a function that takes a component and returns a new component.

StoryBook

Storybook is a tool for UI development. It makes development faster and easier by isolating components. This allows you to work on one component at a time. You can develop entire UIs without needing to start up a complex dev stack, force certain data into your database, or navigate around your application.

```
import React from 'react';

import { Button } from './Button';

export default {
  /* 👇 The title prop is optional.
  * See
https://storybook.js.org/docs/react/configure/overview#configure-story-loadi
ng
  * to learn how to generate automatic titles
  */
  title: 'Button',
  component: Button,
```

```
};

//👇 We create a "template" of how args map to rendering
const Template = (args) => <Button {...args} />;

//👇 Each story then reuses that template
export const Primary = Template.bind({});
Primary.args = {
   primary: true,
   label: 'Button',
};
```

output 

A `*.stories.js` file defines all the stories for a component

Addons are plugins that extend Storybook's core functionality. You can find them in the addons panel, a reserved place in the Storybook UI below the Canvas. Each tab shows the generated metadata, logs, or static analysis for the selected story by the addon

A **decorator** is a way to wrap a story in extra “rendering” functionality. Many addons define decorators to augment your stories with extra rendering or gather details about how your story renders.

```
import React from 'react';

import { ThemeProvider } from 'styled-components';
```

```
export const decorators = [
  (Story) => (
    <ThemeProvider theme="default">
      {Story()}
    </ThemeProvider>
  ),
];
```

Theming
Storybook is theme-able using a lightweight theming API.
The easiest way to customize Storybook is to generate a new theme using the `create()` function from `storybook/theming`.

```
// .storybook/YourTheme.js

import { create } from '@storybook/theming';

export default create({
  base: 'light',
  brandTitle: 'My custom storybook',
  brandUrl: 'https://example.com',
  brandImage: 'https://place-hold.it/350x150',
  brandTarget: '_self',
});
```

```
// .storybook/manager.js

import { addons } from '@storybook/addons';
import yourTheme from './YourTheme';

addons.setConfig({
  theme: yourTheme,
});
```

Hook Form

register function : The hook won't be able to validate a field unless it gets registered

and with it, the validation options

List of validation rules supported:

- required
- min
- max
- minLength
- maxLength
- pattern
- validate

eg

```
import React from "react";
import { useForm } from "react-hook-form";

export default function App() {
  const { register, handleSubmit } = useForm();
  const onSubmit = data => console.log(data);

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register("firstName", { required: true,
maxLength: 20 })} />
      <input {...register("lastName", { pattern:
/^[A-Za-z]+$/i })} />
      <input type="number" {...register("age", { min: 18,
max: 99 })} />
      <input type="submit" />
    </form>
  );
}
```