React Render CodeEvolution

1. What the series talk about.

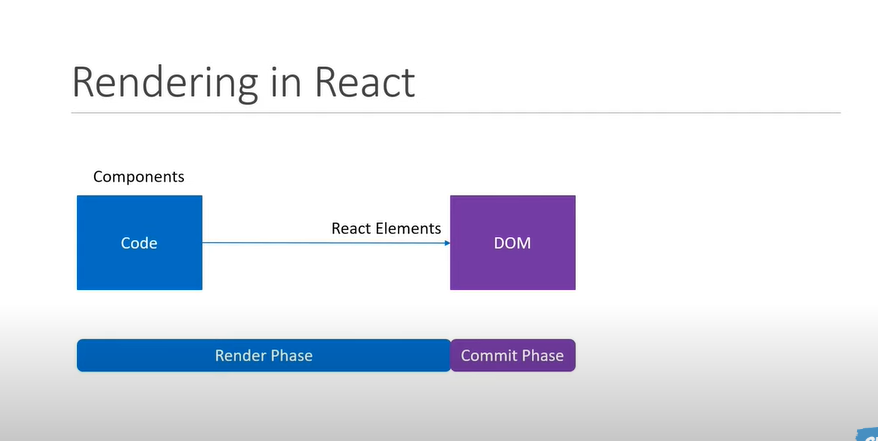
Why render?

Why re-render?

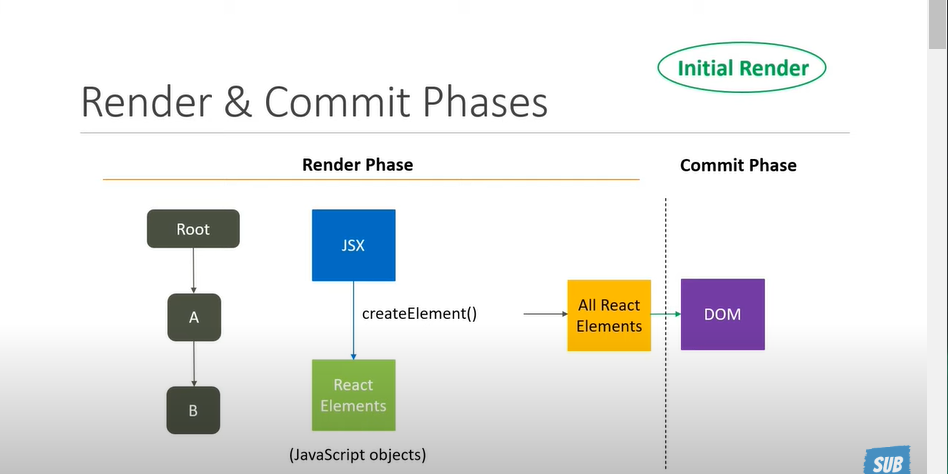
Optimize rendering

Incorrect optimization

1. Rendering and Re-rendering

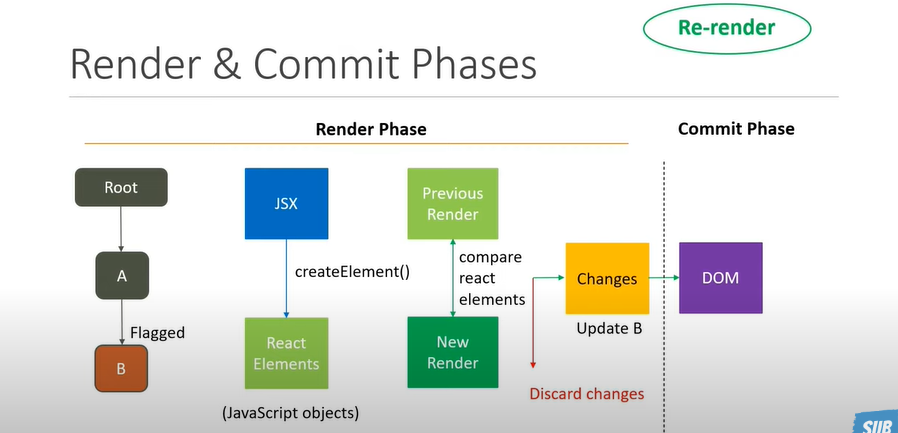


Rendering in React



During the Render phase React will start at the root of the component tree and goes downwards to the leave components while traversing for each the component react invokes the create element method and convert the component jsx into react element and stores that render output ,react lement is basically a js object that describe the structure of the ui on ethe conversion is done all the react elements are handed over to the commit phase, In the commit phase the react element are applied to the dom using the react dom package

Re-rendering in React



During the render phase React will start at the root of the component tree and goes downwards to the leaf components finding all the components that have be flagged as needing updates a component can flag itself for an update by calling the use State setter function or the Use Reducer dispatch function then for each the component react invokes the create element method and convert the component jsx into react element and stores that render output , on ethe conversion is done for all the flagged components react will compare the new set of react elements with the ones that were produces from the last render a list is created with all the changes that need to be made to the dom and handed over the commit phase ,in the commit phase the change are actually applied to the dom . Une image contenant texte

Description générée automatiquement

1. useState an rendering and re-rendering

import React, { useState } from "react";

export const UseState = () => {

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

  console.log('UseState Render')

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

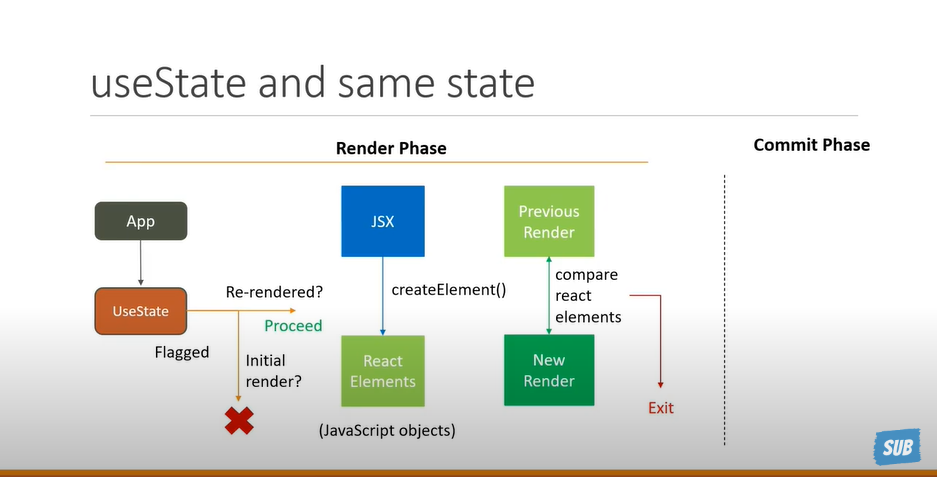
      <button onClick={() => setCount(0)}>Count to 0-{count}</button>

      <button onClick={() => setCount(5)}>Count to 5-{count}</button>

    </div>

  );

};



Une image contenant texte

Description générée automatiquement

1. useReducer an rendering and re-rendering
2. mport React,{useReducer} from 'react'
3. const UseReducer = () => {
4. const initialiseState=0;
5. const reducer=(state,action)=>{
6. switch(action){
7. case 'increment' :return state+1
8. case 'decrement' :return state-1
9. case 'reset' :return initialiseState
10. default :return state
11. }
12. }
13. const [count,dispatch]=useReducer(reducer,initialiseState)
15. console.log("UseReduecer Render")
16. return (
17. <div>
18. <div>{count}</div>
19. <button onClick={() => dispatch("increment")}>Increment</button>
20. <button onClick={() => dispatch("decrement")}>Decrement</button>
21. <button onClick={() => dispatch('reset')}>Reset</button>
22. </div>
23. )
24. }
25. export default UseReducer

The dispatch function from a useReducer hook will cause the component to re-render any time even if the value are the same value after the initial value

1. Rendering and re-rendering state immutability

ObjectUseState

import React ,{useState} from 'react'

const initState={

    fName:'Houssam',

    lName:"Mrabte"

}

export const ObjectUseState = () => {

    const [person,setPerson]=useState(initState)

    const changeName=()=>{

        let prs={...person}

        prs.fName='Abir'

        prs.lName="Lmrabte"

        setPerson(prs)

    }

    console.log("ObjectUseState Render")

  return (

    <div>

        <button onClick={()=>changeName()}>{person.fName}  {person.lName}</button>

    </div>

  )

}

ArrayUseState

import React ,{useState} from 'react'

const initState=['Bruce','Wayne']

export const ArrayUseState = () => {

    const [persons,setPersons]=useState(initState)

    const handleClick=()=>{

        let prs=[...persons]

        prs.push("Clark")

        prs.push("Test")

        setPersons(prs)

    }

    console.log("ArrayUseStateRender")

  return (

    <div>

          <button onClick={handleClick} >Click</button>

          {

            persons.map( (person,i)=>(

                <div key={i}>{person}</div>

            ))

          }

    </div>

  )

}

Une image contenant texte

Description générée automatiquement

1. Rendering with child and parent component

Parent.js

import React, { useState } from "react";

import { Child } from "./Child";

export const Parent = () => {

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

  console.log('Parent Render')

  return (

    <div>

        <Child/>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      <button onClick={() => setCount(0)}>Count to 0-{count}</button>

      <button onClick={() => setCount(5)}>Count to 5-{count}</button>

    </div>

  );

};

Child.js

import React, { useState } from "react";

import { Child } from "./Child";

export const Parent = () => {

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

  console.log('Parent Render')

  return (

    <div>

        <Child/>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      <button onClick={() => setCount(0)}>Count to 0-{count}</button>

      <button onClick={() => setCount(5)}>Count to 5-{count}</button>

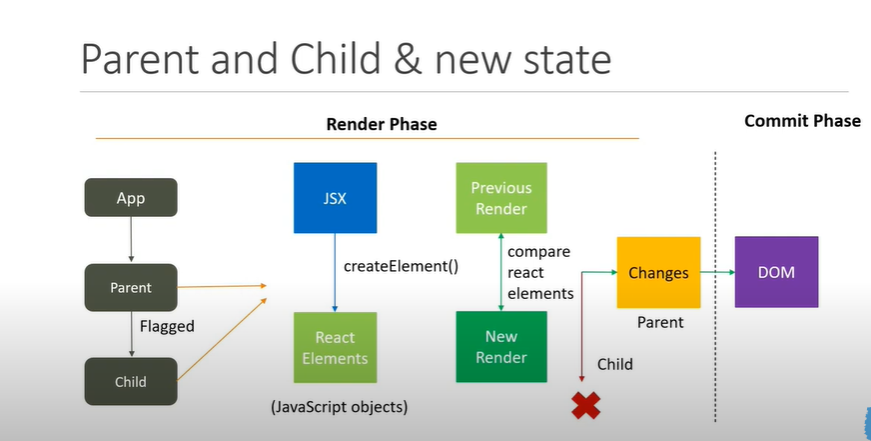
    </div>

  );

};

Une image contenant texte

Description générée automatiquement



Une image contenant texte

Description générée automatiquement

1. Optimization
2. Same Element reference

Une image contenant texte

Description générée automatiquement

Solution:

GrandParents.js

import React,{useState} from 'react'

import { ChildOne } from './ChildOne'

import { ParentOne } from './ParentOne'

export const GrandParents = () => {

    const [newCount,setNewCount]=useState(0)

    console.log("GrandParent Render")

  return (

    <div>

        <button onClick={()=>setNewCount((prev)=>prev+1)}>

        GrandParents - Count :{newCount}

        </button>

        <ParentOne>

            <ChildOne/>

        </ParentOne>

        </div>

  )

}

ParentOne.js

import React, { useState } from "react";

export const ParentOne = ({ children }) => {

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

  console.log("ParentOne Render");

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      {children}

    </div>

  );

};

ChildOne.js

import React from "react";

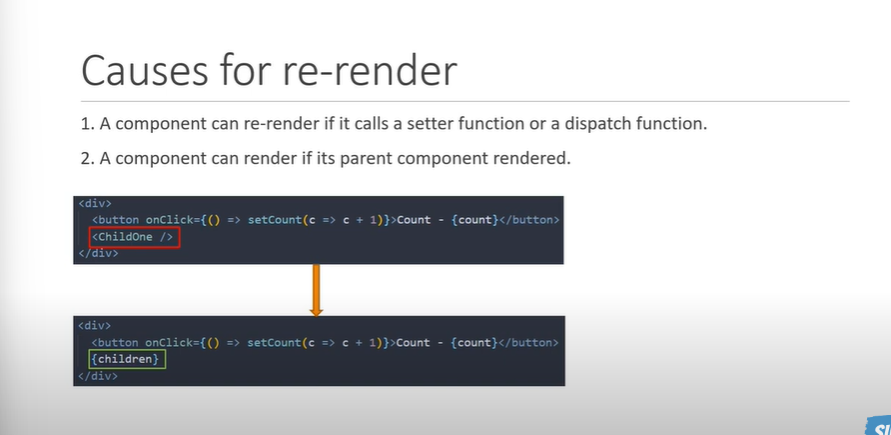
export const ChildOne = () => {

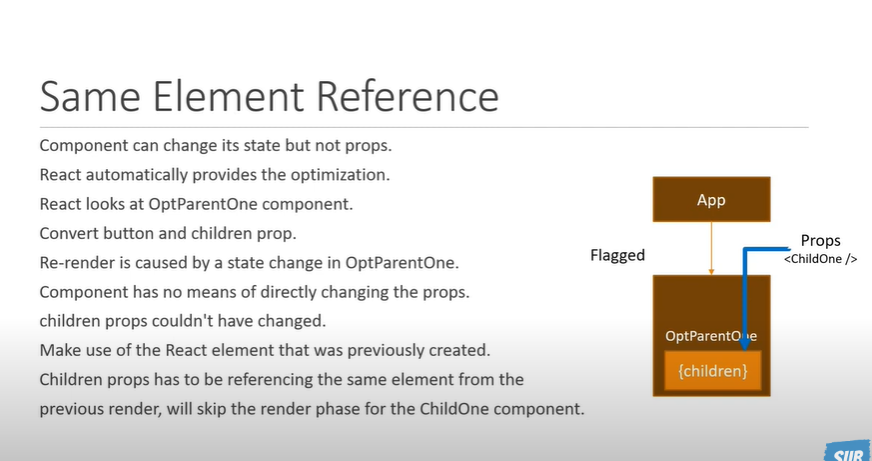
  console.log("ChildOne Render");

  return <div>ChildOne components</div>;

};

Explication:





Une image contenant texte

Description générée automatiquement

`

1. React Memo

ParentTwo.js

import React, { useState } from "react";

import { MemozidChildTwo } from "./ChildTwo";

export const ParentTwo = () => {

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

  const [name, setName] = useState("Houssam");

  console.log("ParentTwo Render");

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      <button onClick={() => setName("CodeEvolution")}>Name-{name}</button>

      <MemozidChildTwo name={name}/>

    </div>

  );

};

ChildTwo.js

import React,{memo} from "react";

export const ChildTwo = () => {

  console.log("ChildTwo Render");

  return <div>ChildTwo components</div>;

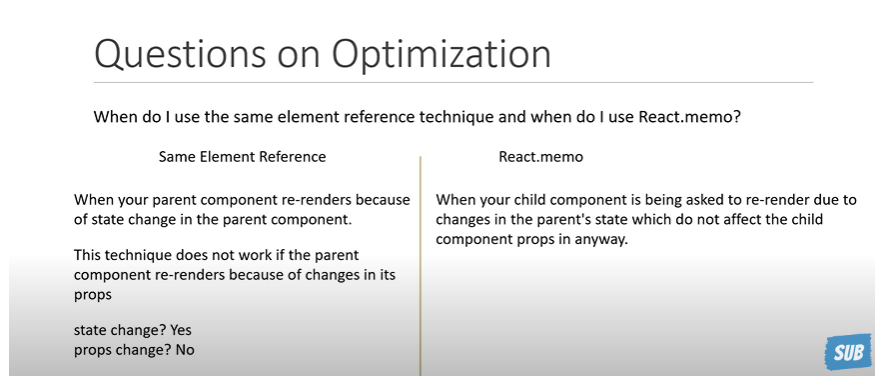
};

export const MemozidChildTwo =memo(ChildTwo)

Une image contenant texte

Description générée automatiquement

1. Questions on Optimization



Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

Optimized render because a react memo

So when we wrapping everything in react.memo can actually be detrimental to the performance of your app ther for it is always a good idea to memorize only expensive components where the props hardly change

Une image contenant texte

Description générée automatiquement

1. Incorrect Memo

Some times we can using React memo in some incorrect situation it’s not really going to help.

* 1. Incorrect Memo with children

ParentThree.js

import { MemozidChildThree } from "./ChildThree";

export const ParentThree = () => {

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

  const [name, setName] = useState("Houssam");

  console.log("ParentThree Render");

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      <button onClick={() => setName("CodeEvolution")}>Name-{name}</button>

      <MemozidChildThree name={name}>

        <strong> Hello</strong>

      </MemozidChildThree>

    </div>

  );

};

ChildThree.js

export const ChildThree = ({ children, name }) => {

  console.log("ChildTwo Three");

  return (

    <div>

      {children} {name}

    </div>

  );

};

export const MemozidChildThree = memo(ChildThree);

when we click on the count button to our surprise we see that both the components re-render even the name value did not change the child component re-render this because of the children props , In React props.children is always a new reference wich will cause the child component to always render ,so there is no need to wrap your child component with React memo if the child component itself has children element.

* 1. Incorrect Memo with Impure Component

ParentThree.js

import React, { useState } from "react";

import { MemozidChildFour } from "./ChildFour";

import { MemozidChildThree } from "./ChildThree";

export const ParentThree = () => {

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

  const [name, setName] = useState("Houssam");

  console.log("ParentThree Render");

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      <button onClick={() => setName("CodeEvolution")}>Name-{name}</button>

      <MemozidChildFour name={name}/>

      {/\* <MemozidChildThree name={name}>

        <strong> Hello</strong>

      </MemozidChildThree> \*/}

    </div>

  );

};

ChildFour.js

mport React, { memo } from "react";

export const ChildFour = ({  name }) => {

  console.log("ChildFour render");

  const date=new Date()

  return (

    <div>

      Hello {name}. it is currently {date.getHours()}:{date.getMinutes()}:{date.getSeconds()}

    </div>

  );

};

export const MemozidChildFour = memo(ChildFour);

**So when the name prop changes the time also updated however with change in count props the time never update because of memorization ,so when we dealing with impure components make sure you’re aware of the consequences when using react.Memo .using randomness is another example we can have the same scenario like dates.**

* 1. **Incorrect memo with props Reference**

**ParentFour.js**

import React, { useState } from "react";

import { MemozidChildFive } from "./ChildFive";

export const ParentFour = () => {

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

  const [name, setName] = useState("Houssam");

  const person={

    fName:'Houssam',

    lName:'Wayne'

  }

 const person1=()=>{

 }

  console.log("ParentFour Render");

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      <button onClick={() => setName("CodeEvolution")}>Name-{name}</button>

      <MemozidChildFive name={name} person={person}/>

    </div>

  );

};

**ChildFive.js**

import React, { memo } from 'react'

export const ChildFive = ({name,person}) => {

    console.log("ChildFive render")

  return (

    <div>

        Hello {name} {person.fName} {person.lName}

    </div>

  )

}

export const MemozidChildFive = memo(ChildFive);

**when we click on the count button to our surprise we see that both the components re-render even the name value did not change the child component re-render because our component has objects or functions and you wrapping one of this child components with ReactMemo it is an incorrect usage of memozation.**

**!!!!!!!!!! the solution is in the bottom**

1. **UseMemo and useCallback**

In th pervious video we learned that having objects or functions as part of a component can nullify the optimization that react memo provides so in this paragraph we have the solutions

ParentFour.js

import React, { useCallback, useMemo, useState } from "react";

import { MemozidChildFive } from "./ChildFive";

export const ParentFour = () => {

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

  const [name, setName] = useState("Houssam");

  const person = {

    fName: "Houssam",

    lName: "Wayne",

  };

  const handleClick = () => {};

  const memorizedMerson = useMemo(() => person, []);

  const memorizedHandleClick = useCallback(() => handleClick, []);

  console.log("ParentFour Render");

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count-{count}</button>

      <button onClick={() => setName("CodeEvolution")}>Name-{name}</button>

      <MemozidChildFive

        name={name}

        person={memorizedMerson}

        handleClick={memorizedHandleClick}

      />

    </div>

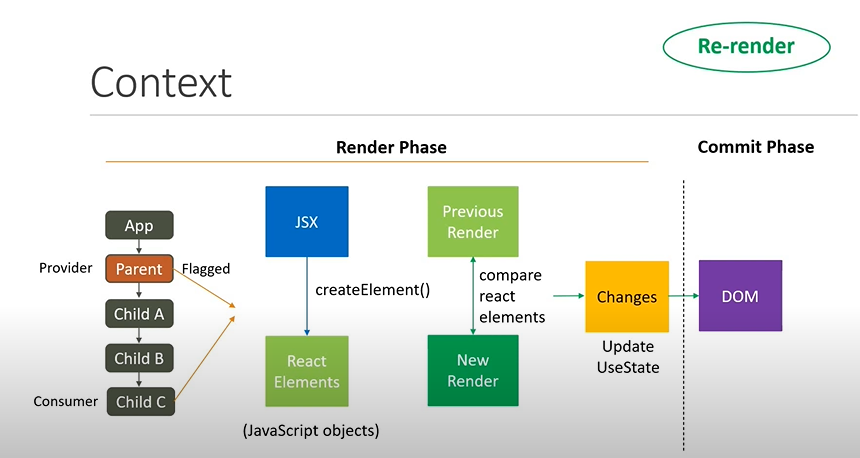
  );

};

**So useMemo and useCallback help us to optimize the react memo provide to optimize component rendering when we using object or function**

1. Context And Memo And Same element reference
   1. Context

Context helps you slove the problem of props drilling so you don’t have to specify props to each Nested component.



ContextParent.js

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

import { ChildA } from "./ContextChildren";

export const CountContext = createContext();

const CountProvider = CountContext.Provider;

export const ContextParent = () => {

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

  console.log("ContextParent Render");

  return (

    <div>

      <button onClick={() => setCount((c) => c + 1)}>Count: {count}</button>

      <CountProvider value={count}>

        <ChildA />

      </CountProvider>

    </div>

  );

};

CountextChildren.js

// I decided that insted of creating three separate files

//It is simpler to have all three children components in the same file

import { useContext } from "react"

import { CountContext } from "./ContextParent"

export const ChildA=()=>{

       console.log("Child A Render")

    return(

        <>

          <div>Child A</div>

          <ChildB/>

        </>

    )

}

export const ChildB=()=>{

    console.log("Child B Render")

 return(

     <>

       <div>Child B</div>

       <ChildC/>

     </>

 )

}

export const ChildC=()=>{

    const count=useContext(CountContext)

    console.log("Child C Render")

 return(

     <>

       <div>Child C  Count = {count}</div>

     </>

 )

}

When the context provider is in the parent component and the parent component's state updates, every child component re-renders and not just the component consuming the context value