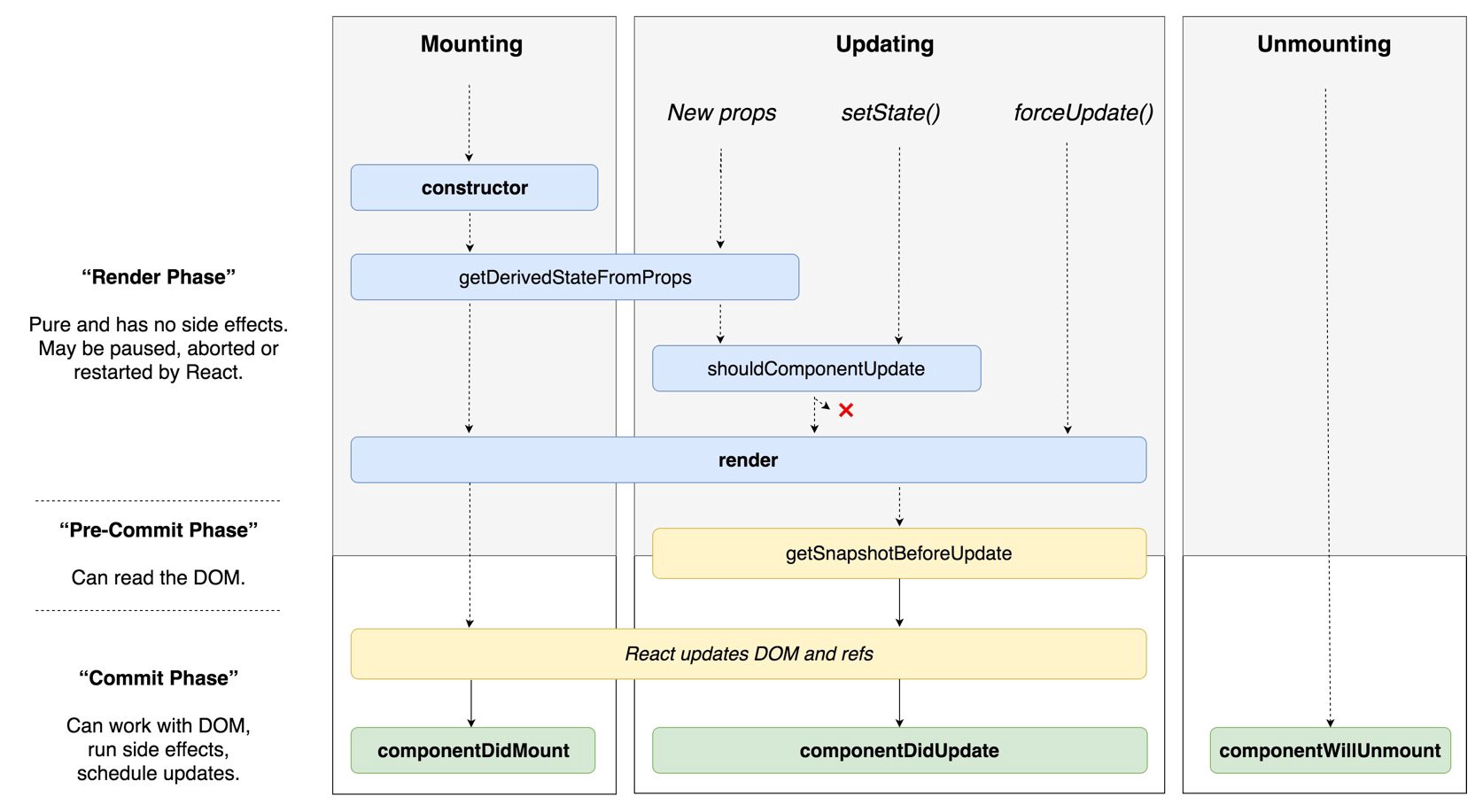
**BÁO CÁO 4: COMPONENT LIFECYCLE, HOOK**

1. **Vòng đời của component**

* Trong React, có 3 giai đoạn (phase) sẽ được thực hiện khi chạy chương trình:
  + Render phase: đây là giai đoạn bao gồm các công việc render components và tính toán các thay đổi cần apply. Giai đoạn này có thể tạm ngưng, hủy hoặc chạy lại bởi React
  + Pre-commit phase: Giai đoạn này sẽ đọc DOM
  + Commit phase: Giai đoạn áp dụng các thay đổi vào DOM thật.
* Trong 3 phase này sẽ có các nhóm phương thức như là *Mounting, Updating, Unmounting* và *Error handling*



1. **Mounting**
2. **constructor()**

* Phương thức constructor() luôn luôn được gọi đầu tiên khi component được khởi tạo và nó là nơi để cài đặt khởi tạo state và các giá trị khác.

constructor(props) {

  super(props);

  this.state = {favoritecolor: "red"};

}

1. **getDerivedStateFromProps()**

* Phương thức này được cọi ngay trước khi render các elements trong DOM. Có thể return một object để update state hoặc null
* Rất hiếm khi sử dụng, chỉ khi giá trị của state phụ thuộc vào prop

static getDerivedStateFromProps(props, state) {

  return {favoritecolor: props.favcol };

}

1. **render()**

* Đây là phương thức bắt buộc duy nhất khi tạo ra một component, trả về một trong những giá trị sau:
  + React element
  + Arrays và fragment
  + Portals
  + String và numbers
  + Boolean hoặc null

render() {

  return (

    <h1>This is the content of the Header component</h1>

  );

}

1. **componentDidMount()**

* Đây là phương thức được gọi sau khi render. Được thực hiện để kết nối React với một ứng dụng bên ngoài ví dụ như web API. Ngoài ra, đây cũng là nơi thích hợp để thực hiện các hàm setInterval hoặc setTimeout

componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)

  }

1. **Updating**
2. **getDerivedStateFromProps()**

* Phương thức này sẽ được gọi xuống trước khi render, có thể return được một object để update state hoặc null
* Rất hiếm khi sử dụng, chỉ khi giá trị của state phụ thuộc vào prop
* Nên hạn chế sử dụng method này vì logic hiển thị khá khó hiểu.

class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  static getDerivedStateFromProps(props, state) {

    return {favoritecolor: props.favcol };

  }

  render() {

    return (

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

    );

  }

}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));

1. **shouldComponentUpdate()**

* Đây là phương thức sẽ xác định rằng component có được update hay không. Mặc định sẽ trả về là true. Có thể re-render lại component bằng cách thêm một số ràng buộc điều kiện trong method này.
* Giả sử muốn re-render component khi props thay đổi thì method sẽ nhận nextProps và nextState để giúp chúng ta so sánh với prop hiện tại

shouldComponentUpdate(nextProps, nextState) {

  return this.props.clicks !== nextProps.clicks;

}

1. **componentWillUpdate()**

* Phương thức này sẽ được gọi giữa shouldComponentUpdate() và render(). Nó nhận 2 tham số truyền vào là nextProps và nextState.
* Phương thức này không thể gọi được **this.setState** bởi vì mục tiêu của method này là tương tác với những thứ bên ngoài React

componentWillUpdate(nextProps, nextState) {

  if (nextState.open == true && this.state.open == false) {

    this.props.onWillOpen();

  }

}

1. **getSnapshotBeforeUpdate()**

* Gọi ngay trước khi render xuống DOM, cho phép lấy một số thông tin của DOM, các giá trị return từ hàm này sẽ đưa cho **componentDidUpdate()**

getSnapshotBeforeUpdate(prevProps, prevState) {

  document.getElementById("div1").innerHTML =

  "Before the update, the favorite was" + prevState.favoritecolor;

}

componentDidUpdate() {

  document.getElementById("div2").innerHTML =

  "The updated favorite is " + this.state.favoritecolor;

}

1. **componentDidUpdate()**

* Phương thức này sẽ chỉ được gọi sau khi re-render component. Sau mỗi lần update mới, component sẽ được update vào Dom và phương thức này sẽ được thực hiện. Method này sẽ nhận vào các đối số như prevProps và prevState.

componentDidUpdate() {

  document.getElementById("mydiv").innerHTML =

  "The updated favorite is " + this.state.favoritecolor;

}

1. **componentWillReceiveProps()**

* Phương thức này sẽ được gọi trước khi một component nhận props mới.
* Nếu component cha làm các components con phải re-render lại thì phương thức này sẽ được gọi kể cả khi props không thay đổi

componentWillReceiveProps(nextProps){

  if(nextProps.someValue!==this.props.someValue){

    //Perform some operation

    this.setState({someState: someValue });

    // ....

  }

}

1. **Unmounting**
2. **componentWillUnmount()**

* Có thể sử dụng để remove các listener, các hàm setInterval, cancel network request và nó sẽ được gọi trước khi unmount component

componentWillUnmount() {

  document.removeEventListener("click", this.closeMenu);

}

1. **Error handling**
2. **componentDidCatch()**

* Nếu một component nào đó bị lỗi nó sẽ không chết nguyên cái app nữa mà sẽ bắt lỗi ở đây

componentDidCatch(error, errorInfo) {

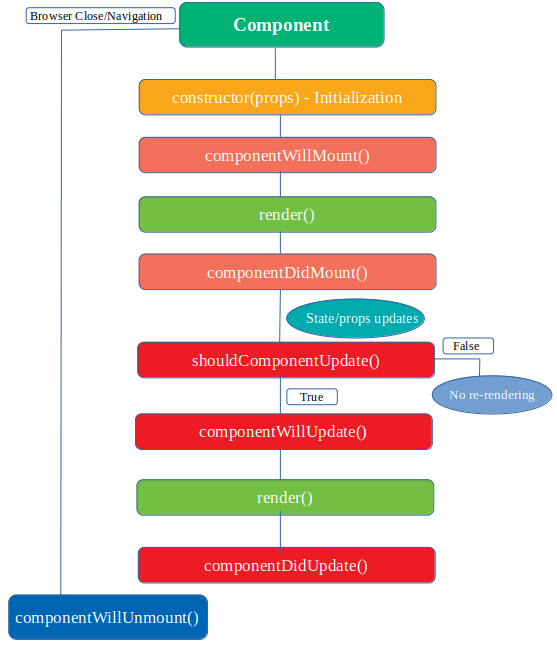
  // You can also log the error to an error reporting service

  logErrorToMyService(error, errorInfo);

}

1. **Tổng kết về lifecycle component**

* Đây là sơ đồ thứ tự thực hiện của các method trong một lifecycle



1. **Hook**

Với mục đích sử dụng state và các chức năng của một class component mà không cần phải tạo class nên hook đã ra đời

1. **useState**

* Giúp sử dụng state trong functional componnet
* Truyền vào là initialState (giá trị hoặc function)
* Trả về một mảng có 2 phần tương ứng là state và setState. **state:** định nghĩa tên của state nó có thể là đơn giá trị hoặc object,.. (là tham số của useState), **setState**: định nghĩa tên function dùng cho việc update state (là tham số của useState)
* Ví dụ:

const [count, setCount] = useState(0)

const handleClick = () => setCount(age + 1)

1. **useEffect**

* **useEffect** cho phép chúng ta sử lý logic trong lifecycle methods.
* Đây là sự kết hợp của componentDidMount, componentDidUpdate và componentWillMount
* Nếu muốn dùng như componentDidUpdate thì ta dùng như sau: useEffect(function)

const [count, setCount] = useState(0)

const handleClick = () => setCount(count + 1)

useEffect(() => {

  document.title = 'Count is: ' + count

})

* Còn nếu muốn dùng như componentDidMount thì ta truyền vô thêm một mảng rỗng :

const [count, setCount] = useState(0)

const handleClick = () => setCount(count + 1)

useEffect(() => {

  document.title = 'Count is: ' + count

}, [])

* Còn với componentWillUnmount thì ta sẽ trả về một function và function này sẽ được gọi ngay trước khi component unmount

useEffect(() => {

  const clickWindow = () => console.log('1')

  window.addEventListener('click', clickWindow)

  //return 1 function, sẽ được gọi ngay trước khi component unmount

  return () => {

    window.removeEventListener('click', clicked)

  }

}, [])

1. **useContext**

* **useContext** là một hooks trong React Hooks cho phép chúng ta có thể làm việc với React Context trong một functional component.

const value = useContext(AppContext);

1. **useLayoutEffect**

* Method này khác với useEffect ở chỗ là thời điểm gọi (không hiểu lắm)

useLayoutEffect(() => {

  //Do something and either return undefined or a cleanup function

  return () => {

    //Do some cleanup here

  };

}, [dependencies]);

1. **useReducer**

* Đây là một phiên bản nâng cao của useState, dùng trong trường hợp local state của component phức tạp, có nhiều action làm thay đổi state đó.
* Hàm có 2 tham số là state, action và trả về new state sau khi thực hiện một action với cú pháp như sau (state, action) => newState

1. **useCallback**

* useCallback có chức năng tránh re-render ở component con, ta xét ví dụ sau:

function Parent({ ... }) {

  const [a, setA] = useState(0);

  const onChangeHandler = useCallback(() => {

    doSomething(a);

  }, [a]);

  ...

  return (

    ...

    //Pure là component con có sử dụng React.memo

    <Pure onChange={onChangeHandler} />

  );

}

* Mỗi lần component **Parent** re-render thì hàm **onChangeHandler** sẽ không cần phải tạo mới nữa mà nó chỉ tạo mới khi biến **a** thay đổi.

1. **useMemo**

* Tương tự như useCallback, useMemo cũng có chức năng tránh re-render, ngoài ra cũng tránh được những tính toán nặng. Ta xét ví dụ sau:

const componentA = () => {

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

  const getArray = useMemo(() => {

    // tưởng tượng một hàm phức tạp, filter,

    // sort một mảng 100 phần tử, tốn 2s để chạy

    const result = filterAndSortAndDoSomething(...);

    return result;

  }, []);

  return (

    <div>

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

      <div>mảng phức tạp: {getArray}</div>

    </div>

  )

}

* Khi bấm nút và tăng biến **count** thì hàm **getArray** sẽ trả ra giá trị **result** trước đó mà không cần mất 2 giây để tính toán

1. **useRef**

* Công dụng của useRef là cố định dữ liệu giữa các lần re-render, truy xuất giá trị đó qua thuộc tính current
* Một ứng dụng rất phổ biến của useRef là truy xuất đến DOM node. Thí dụ để set focus của input

function Form() {

  const nameRef = React.useRef();

  const emailRef = React.useRef();

  const passwordRef = React.useRef();

  const handleSubmit = (e) => {

    e.preventDefault();

    const name = nameRef.current.value;

    const email = emailRef.current.value;

    const password = passwordRef.current.value;

    console.log(name, email, password);

  };

  return (

    <React.Fragment>

      <label>

        Name:

        <input placeholder="name" type="text" ref={nameRef} />

      </label>

      <label>

        Email:

        <input placeholder="email" type="text" ref={emailRef} />

      </label>

      <label>

        Password:

        <input placeholder="password" type="text" ref={passwordRef} />

      </label>

      <button onClick={() => nameRef.current.focus()}>Focus Name Input</button>

      <button onClick={() => emailRef.current.focus()}>

        Focus Email Input

      </button>

      <button onClick={() => passwordRef.current.focus()}>

        Focus Password Input

      </button>

      <button onClick={handleSubmit}>Submit</button>

    </React.Fragment>

  );

}

1. **Nguồn tham khảo**

[Chi tiết về lifecycle method trong ReactJS - CodersX (coders-x.com)](https://coders-x.com/chi-tiet-ve-lifecycle-method-trong-reactjs/)

[React Lifecycle (w3schools.com)](https://www.w3schools.com/react/react_lifecycle.asp)

[Giải thích React Component Lifecycle | TopDev](https://topdev.vn/blog/react-component-lifecycle/#static-getderivedstatefrompropsprops-state)

[Error Boundaries – React (reactjs.org)](https://reactjs.org/docs/error-boundaries.html)

[ReactJS: useState trong React Hooks. (viblo.asia)](https://viblo.asia/p/reactjs-usestate-trong-react-hooks-yMnKMbzQZ7P)

[Đôi chút hiểu về useEffect trong reactJS Hook. (viblo.asia)](https://viblo.asia/p/doi-chut-hieu-ve-useeffect-trong-reactjs-hook-RQqKLkA057z)

[useContext trong React Hook - Freetuts](https://freetuts.net/usecontext-trong-react-hook-2611.html)

[Sử dụng userLayoutoutEffect trong ReactJs (viblo.asia)](https://viblo.asia/p/su-dung-userlayoutouteffect-trong-reactjs-naQZRLxX5vx)