**CÁC KHÁI NIỆM CƠ BẢN TRONG REACTJS**

BÁO CÁO KỸ THUẬT (10–15 TRANG)

Sinh viên: [Tạ Văn Thắng] | MSSV: [2210900063]

# MỤC LỤC

1. Giới thiệu  
2. Virtual DOM  
3. JSX  
4. Components (Thành phần)  
5. Props & State  
6. React Lifecycle (Vòng đời)  
7. Kết luận  
8. Tài liệu tham khảo

# 1. Giới thiệu

ReactJS là một thư viện JavaScript mã nguồn mở dùng để xây dựng giao diện người dùng (UI). Được phát triển bởi Facebook (nay là Meta), React ra đời nhằm giải quyết các bài toán: tách giao diện thành các thành phần nhỏ có thể tái sử dụng, quản lý trạng thái hiệu quả và cập nhật UI nhanh, mượt.  
  
Báo cáo này đi sâu vào 5 khái niệm cốt lõi khi bắt đầu với ReactJS: Virtual DOM, JSX, Components, Props & State, và React Lifecycle. Mỗi mục bao gồm: định nghĩa, cơ chế hoạt động, ví dụ minh hoạ, lưu ý quan trọng, best practices và lỗi thường gặp, giúp bạn nắm vững nền tảng để phát triển ứng dụng React hiện đại.

# 2. Virtual DOM

**2.1. Khái niệm**

Virtual DOM (VDOM) là một biểu diễn (representation) dạng cây của UI trong bộ nhớ. Khi dữ liệu (state/props) thay đổi, React tạo ra một cây VDOM mới, so sánh (diff) với cây VDOM cũ để biết chỗ nào khác biệt, từ đó tính toán một tập các thay đổi tối thiểu (patch) và áp dụng chúng lên DOM thật (real DOM). Cách tiếp cận này giúp giảm số lần thao tác trực tiếp với DOM vốn tốn kém, nâng cao hiệu suất render.

**2.2. Tại sao cần Virtual DOM?**

* DOM thật chậm khi thay đổi nhiều node liên tiếp; VDOM giúp gom và tối ưu các cập nhật.
* Tách biệt logic UI và thao tác DOM, giúp code dễ dự đoán (predictable).
* Là nền tảng cho cơ chế reconciliation và kiến trúc Fiber (React 16+), cải thiện khả năng lên lịch (scheduling).

**2.3. Cách hoạt động (Diffing & Reconciliation)**

1. Tạo VDOM từ JSX/Component tree.
2. Khi state/props thay đổi → tạo VDOM mới.
3. So sánh VDOM mới với VDOM cũ theo từng node (same type, same key).
4. Sinh ra danh sách thay đổi tối thiểu và cập nhật DOM thật.

Heuristics chính: nếu hai phần tử có cùng loại (type) và cùng key, React sẽ giữ lại node DOM cũ và chỉ cập nhật props/con children. Nếu khác type hoặc khác key, React bỏ node cũ và tạo node mới. Với danh sách (list), việc dùng key ổn định là cực kỳ quan trọng.

**2.4. Keys trong danh sách**

* Key giúp React nhận diện phần tử là duy nhất giữa các lần render.
* Không dùng index làm key nếu phần tử có thể thêm/xoá/đổi chỗ → tránh lỗi render sai và mất state.
* Ưu tiên dùng id ổn định từ dữ liệu.

**2.5. Ví dụ minh hoạ diffing (mã giả)**

// Pseudo-code minh hoạ

const oldList = ['A', 'B', 'C']; // key: A, B, C

const newList = ['B', 'C', 'D']; // key: B, C, D

// React nhận ra 'A' bị remove, 'D' được add; 'B','C' giữ nguyên (di chuyển).

**2.6. Best practices & Pitfalls**

* Luôn cung cấp key ổn định cho danh sách.
* Tránh mutate DOM trực tiếp ngoài React (có thể dùng useRef + effect nếu bắt buộc).
* Đo lường trước khi tối ưu (React DevTools Profiler).
* Memo hoá component (React.memo) khi chi phí render cao và props ít đổi.

# 3. JSX

**3.1. Khái niệm**

JSX (JavaScript XML) là cú pháp mở rộng cho phép viết UI bằng cấu trúc giống HTML ngay trong JavaScript. JSX sẽ được biên dịch (thường bởi Babel) thành lời gọi React.createElement(...) tạo nên cấu trúc VDOM.

**3.2. Quy tắc cú pháp quan trọng**

* Phải có một thẻ gốc bọc ngoài (wrapper) hoặc dùng <>...</> (Fragment).
* Dùng className thay cho class; htmlFor thay cho for.
* Biểu thức JS đặt trong ngoặc nhọn {} (ví dụ: {user.name}).
* Thuộc tính boole có thể viết dạng rút gọn (disabled).
* Style inline nhận object JS: style={{ marginTop: 8 }}.

**3.3. Rendering có điều kiện & danh sách**

function Greeting({ user }) {

return (

<div>

{user ? <h1>Xin chào, {user.name}</h1> : <h1>Xin chào, Khách</h1>}

</div>

);

}

function TodoList({ items }) {

return (

<ul>

{items.map(item => (

<li key={item.id}>{item.text}</li>

))}

</ul>

);

}

**3.4. XSS và JSX**

JSX mặc định escape chuỗi để tránh XSS. Chỉ khi dùng dangerouslySetInnerHTML mới chèn HTML thô. Hạn chế tối đa việc dùng dangerouslySetInnerHTML, và nếu bắt buộc phải dùng, hãy làm sạch (sanitize) dữ liệu.

**3.5. Best practices**

* Tách UI thành component nhỏ, có tên rõ ràng.
* Tránh logic phức tạp trong JSX; đưa vào hàm/tham số trước rồi render kết quả.
* Dùng Fragment <> </> thay vì <div> thừa.
* Giữ JSX thuần (no side-effects).

# 4. Components (Thành phần)

**4.1. Khái niệm**

Component là đơn vị xây dựng UI trong React. Mỗi component nhận vào props, có thể có state (đối với function component dùng hook useState) và trả về JSX mô tả giao diện.

**4.2. Function Component & Class Component**

// Function component (khuyến nghị)

function Profile({ name }) {

return <h2>Xin chào, {name}</h2>;

}

// Class component (di sản, vẫn dùng trong codebase cũ)

class ProfileClass extends React.Component {

render() {

return <h2>Xin chào, {this.props.name}</h2>;

}

}

**4.3. Composition (Thành phần lắp ghép) & Children**

function Card({ children }) {

return <div className="card">{children}</div>;

}

function App() {

return (

<Card>

<h3>Tiêu đề</h3>

<p>Nội dung</p>

</Card>

);

}

**4.4. Controlled vs Uncontrolled Components (Form)**

// Controlled: giá trị do state quản lý

function EmailInput() {

const [email, setEmail] = React.useState('');

return <input value={email} onChange={e => setEmail(e.target.value)} />;

}

// Uncontrolled: truy cập qua ref

function FilePicker() {

const inputRef = React.useRef(null);

const pick = () => inputRef.current && inputRef.current.click();

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

}

**4.5. Tối ưu hoá Components**

* React.memo cho function component để tránh render lại khi props không đổi theo shallow compare.
* useMemo/useCallback để tránh tạo lại giá trị/hàm tốn kém không cần thiết.
* Phân tách component lớn thành các phần nhỏ (split) để nâng cao khả năng tái sử dụng và tối ưu hiệu năng.

# 5. Props & State

**5.1. Props là gì?**

Props (properties) là dữ liệu được truyền từ component cha xuống component con. Props là bất biến (immutable) trong component con; để thay đổi dữ liệu hiển thị, component cha cần truyền props mới.

**5.2. State là gì?**

State là dữ liệu nội bộ của component, đại diện cho tình trạng hiện tại của UI. Thay đổi state sẽ kích hoạt quá trình render lại. Trong function component, dùng hook useState hoặc useReducer để quản lý state.

**5.3. Ví dụ useState & useReducer**

function Counter() {

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

return (

<div>

<p>Count: {count}</p>

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

</div>

);

}

function CounterReducer() {

function reducer(state, action) {

switch (action.type) {

case 'inc': return { value: state.value + 1 };

case 'dec': return { value: state.value - 1 };

default: return state;

}

}

const [state, dispatch] = React.useReducer(reducer, { value: 0 });

return (

<div>

<p>Count: {state.value}</p>

<button onClick={() => dispatch({ type: 'inc' })}>+</button>

<button onClick={() => dispatch({ type: 'dec' })}>-</button>

</div>

);

}

**5.4. Quy tắc cập nhật state**

* Không mutate trực tiếp: setState với giá trị mới (hoặc dùng updater function).
* Nhiều setState trong cùng tick có thể được gộp (batching).
* State cập nhật bất đồng bộ về mặt khái niệm; đừng dựa vào state vừa set trong cùng lần render, hãy dùng hàm cập nhật theo prevState (setCount(c => c + 1)).

**5.5. Lifting State Up (Nâng state lên cha)**

Khi nhiều component con cần dùng chung dữ liệu hoặc cần đồng bộ, hãy đưa state lên component cha chung để làm nguồn dữ liệu duy nhất (single source of truth).

**5.6. Prop Drilling & Context API**

Prop drilling xảy ra khi phải truyền props qua nhiều tầng không cần thiết. Context API cho phép chia sẻ dữ liệu xuống sâu mà không cần truyền qua từng cấp. Tuy nhiên, lạm dụng Context có thể khiến nhiều component render lại; cân nhắc tách context hoặc kết hợp memo hoá.

**5.7. Bảng so sánh nhanh Props vs State**

|  |  |  |
| --- | --- | --- |
| Tiêu chí | Props | State |
| Nguồn dữ liệu | Từ component cha | Nội bộ component |
| Khả năng thay đổi | Không (immutable trong con) | Có (thông qua setState/useState) |
| Mục đích | Cấu hình/đầu vào cho component | Theo dõi tình trạng, dữ liệu thay đổi theo thời gian |
| Cập nhật kích hoạt render | Khi cha truyền props mới | Khi setState/useState được gọi |

# 6. React Lifecycle (Vòng đời)

**6.1. Tổng quan**

Vòng đời mô tả các giai đoạn của component từ khi được tạo (mount), cập nhật (update) đến khi gỡ bỏ (unmount). Với class component có các phương thức vòng đời; với function component dùng các hook như useEffect/useLayoutEffect để mô phỏng hành vi tương ứng.

**6.2. Lifecycle trong Class Component (điển hình)**

* constructor(props) → khởi tạo state, bind method (không side-effects).
* render() → mô tả UI (pure).
* componentDidMount() → chạy một lần sau khi mount (fetch data, subscribe).
* shouldComponentUpdate(nextProps, nextState) → tối ưu render có điều kiện.
* componentDidUpdate(prevProps, prevState) → chạy sau mỗi lần update (sync DOM, network).
* getSnapshotBeforeUpdate(prevProps, prevState) → chụp thông tin trước khi DOM cập nhật (hiếm dùng).
* componentWillUnmount() → dọn dẹp (unsubscribe, clear timer).
* componentDidCatch(error, info) → bắt lỗi con (Error Boundary).

**6.3. Tương quan sang Function Component (Hooks)**

|  |  |
| --- | --- |
| Class lifecycle | Hooks tương ứng (Function) |
| constructor | Khởi tạo useState/useRef trong thân component |
| render | Phần return của function component (không side-effects) |
| componentDidMount | useEffect(() => {...}, []) |
| shouldComponentUpdate | React.memo, useMemo/useCallback (tối ưu hoá) |
| componentDidUpdate | useEffect(() => {...}, [deps]) |
| getSnapshotBeforeUpdate | useLayoutEffect (trước khi browser paint) |
| componentWillUnmount | cleanup trong useEffect: return () => {...} |
| componentDidCatch | Error Boundary (chỉ class) hoặc thư viện hỗ trợ |

**6.4. useEffect & useLayoutEffect**

* useEffect chạy sau khi render commit; thích hợp cho side-effect không chặn vẽ (fetch, event listeners).
* useLayoutEffect chạy đồng bộ sau khi DOM cập nhật nhưng trước khi browser paint; dùng khi cần đo đạc/layout.
* Luôn dọn dẹp (cleanup) để tránh rò rỉ tài nguyên: return () => unsubscribe/clearInterval...
* Quản lý deps đúng để tránh lặp vô hạn; cân nhắc tách effect theo mục tiêu rõ ràng.

**6.5. Ví dụ lifecycle với useEffect**

function ChatRoom({ roomId }) {

const [messages, setMessages] = React.useState([]);

React.useEffect(() => {

const unsub = subscribe(roomId, msg => setMessages(m => [...m, msg]));

return () => unsub(); // cleanup khi roomId đổi hoặc unmount

}, [roomId]);

return <MessageList data={messages} />;

}

**6.6. Concurrent features (React 18+)**

* Tự động batching setState trong nhiều trường hợp (kể cả promise/timers).
* Transitions (startTransition) để phân biệt cập nhật ưu tiên thấp, cho UI mượt mà hơn.
* Suspense tốt hơn cho phân tách code và data fetching (với router/libs phù hợp).

# 7. Kết luận

Nắm vững 5 khái niệm cốt lõi—Virtual DOM, JSX, Components, Props & State, và Lifecycle—giúp bạn viết ứng dụng React đúng chuẩn, dễ mở rộng, và hiệu năng tốt. Thực hành với các ví dụ nhỏ, đo đạc bằng React DevTools, và áp dụng best practices sẽ rút ngắn đáng kể đường cong học tập.

# 8. Tài liệu tham khảo

* React Docs – react.dev: Hướng dẫn chính thức, cập nhật.
* React DevTools – tiện ích đo đạc render và tối ưu hoá.
* Create React App / Vite – khởi tạo dự án nhanh.
* Babel – biên dịch JSX sang JavaScript.