Chapter 01: Giới thiệu về ReactJS và Tư duy Component

A. Mục tiêu

Sau khi hoàn thành bài học này, học viên sẽ:

- Hiểu React là gì, lý do ra đời và các vấn đề mà nó giải quyết (Virtual DOM).
- Thành thạo cú pháp JSX và sự khác biệt so với HTML.
- Nắm vững khái niệm Component và áp dụng hiệu quả.
- Học cách truyền dữ liệu từ component cha sang con bằng Props.
- Làm quen với khái niệm State và hook `useState` để quản lý trạng thái component.

Giải thích bổ sung:
Mục tiêu bài học được thiết kế để giúp học viên hiểu rõ nền tảng của ReactJS, từ lý thuyết đến thực hành. Mỗi mục tiêu đều hướng đến việc xây dựng kỹ năng thực tế, chẳng hạn như sử dụng JSX để viết giao diện, chia nhỏ UI thành các component độc lập, và quản lý trạng thái động bằng `useState`. Các khái niệm này là nền tảng để phát triển ứng dụng web hiện đại.

B. Nội dung lý thuyết

1. Tổng quan về ReactJS

Lịch sử và sự phát triển

ReactJS là một thư viện JavaScript mã nguồn mở, được phát triển bởi **Facebook** vào năm **2013**, nhằm xây dựng giao diện người dùng (UI) hiệu quả, đặc biệt cho các ứng dụng **single-page application (SPA)**. React trở nên phổ biến nhờ tính đơn giản, hiệu suất cao và cách tiếp cận dựa trên **component**. Một số cột mốc quan trọng:

- **2013**: React ra đời và được sử dụng trong các sản phẩm của Facebook như Instagram.
- **2018**: Giới thiệu **React Hooks**, cho phép sử dụng trạng thái và các tính năng khác trong functional component, giảm sự phụ thuộc vào class component.
- **2022**: React 18 ra mắt với **Concurrent Rendering**, cải thiện hiệu suất và trải nghiệm người dùng khi xử lý các tác vụ phức tạp.

Giải thích bổ sung:
React được tạo ra để giải quyết vấn đề cập nhật giao diện phức tạp trong các ứng dụng lớn. Trước React, các thư viện như jQuery yêu cầu lập trình viên phải tự quản lý cập nhật DOM, dễ dẫn đến lỗi và hiệu suất kém. React cung cấp cách tiếp cận mới, giúp lập trình viên tập trung vào việc mô tả giao diện thay vì cách thức cập nhật nó.

Virtual DOM

Virtual DOM là một bản sao nhẹ của DOM thực tế, được lưu trong bộ nhớ. React sử dụng Virtual DOM để tối ưu hóa việc cập nhật giao diện:

1. Khi trạng thái (state) hoặc props của component thay đổi, React tạo một cây Virtual DOM mới.
2. React so sánh (diff) cây Virtual DOM mới với cây cũ để xác định phần thay đổi.
3. Chỉ những phần thay đổi được cập nhật vào DOM thực tế, giảm thiểu thao tác tốn kém.

Ví dụ minh họa:
Nếu bạn có một danh sách 100 mục, với vanilla JavaScript hoặc jQuery, bạn phải lặp qua từng mục để cập nhật thủ công. Với React, chỉ những mục thay đổi được cập nhật, giúp cải thiện hiệu suất đáng kể. Ví dụ:

```
// Vanilla JS (cách tiếp cận truyền thống)
document.getElementById('list').innerHTML = newListHTML;

// React (Virtual DOM tự động cập nhật)
return <ul>{items.map(item => <li>{item}</li>)}</ul>;
```

Giải thích bổ sung:
Virtual DOM giống như một bản phác thảo của giao diện. Thay vì vẽ lại toàn bộ bức tranh (DOM thực), React chỉ chỉnh sửa những phần cần thay đổi, giống như chỉnh sửa một vài nét vẽ trên bản phác thảo. Điều này giúp ứng dụng nhanh hơn, đặc biệt với giao diện phức tạp.

Declarative UI

React sử dụng cách tiếp cận **declarative** (mô tả) thay vì **imperative** (chỉ thị):

- **Imperative**: Lập trình viên phải chỉ rõ từng bước để cập nhật giao diện, ví dụ: `document.getElementById('button').innerText = 'Clicked';`.
- **Declarative**: Chỉ cần mô tả giao diện sẽ trông như thế nào dựa trên trạng thái hiện tại, React tự động xử lý phần còn lại.

Ví dụ minh họa:

```
// Imperative
if (isClicked) {
  document.getElementById('button').innerText = 'Clicked';
} else {
  document.getElementById('button').innerText = 'Click me';
}

// Declarative
<button>{isClicked ? 'Clicked' : 'Click me'}</button>
```

Giải thích bổ sung:
Cách tiếp cận declarative giúp mã dễ đọc, dễ bảo trì và ít lỗi hơn. Thay vì phải viết các lệnh cập nhật DOM phức tạp, bạn chỉ cần mô tả giao diện dựa trên trạng thái, và React sẽ đảm bảo giao diện luôn đồng bộ với dữ liệu.

---

2. Thiết lập môi trường

Công cụ

Ngày nay, **Vite** là lựa chọn phổ biến để khởi tạo dự án React nhờ tốc độ nhanh và hỗ trợ **Hot Module Replacement (HMR)**. So với `create-react-app`, Vite có thời gian build nhanh hơn và phù hợp với các dự án hiện đại.

Giải thích bổ sung:
Vite sử dụng ES modules (mô-đun JavaScript hiện đại) thay vì cách đóng gói truyền thống, giúp giảm thời gian tải trong quá trình phát triển. Điều này đặc biệt hữu ích khi bạn làm việc với các dự án lớn hoặc cần cập nhật giao diện ngay lập tức.

Các bước tạo dự án React với Vite

1. Cài đặt **Node.js** (phiên bản 18 trở lên).
2. Mở terminal và chạy:

```
npm create vite@latest my-react-app -- --template react
cd my-react-app
npm install
npm run dev
```

3. Truy cập `http://localhost:5173` để xem ứng dụng mặc định.

Lưu ý: Nếu bạn gặp lỗi khi chạy `npm create vite`, hãy thử cập nhật npm bằng lệnh `npm install -g npm@latest`.

Cấu trúc dự án

Cấu trúc thư mục của một dự án Vite + React:

- `public/`: Chứa các tài nguyên tĩnh như hình ảnh, favicon.
- `src/`: Thư mục chính chứa mã nguồn:
  - `App.jsx`: Component gốc của ứng dụng.
  - `main.jsx`: Điểm vào, nơi ứng dụng được gắn vào DOM.
- `index.html`: File HTML chính.

- `vite.config.js` : File cấu hình Vite.

Giải thích bổ sung:
Cấu trúc dự án được thiết kế để tách biệt mã nguồn (`src`) và tài nguyên tĩnh (`public`). File `main.jsx` đóng vai trò kết nối React với DOM, thường chứa lệnh `ReactDOM.render` để gắn ứng dụng vào `#root` trong `index.html` .

---

3. JSX - JavaScript XML

JSX là một phần mở rộng cú pháp cho JavaScript, cho phép viết mã giống HTML trong file JavaScript. JSX không phải HTML mà được chuyển đổi thành các lệnh `React.createElement()` bởi **Babel**.

Quy tắc quan trọng của JSX

- **Một thẻ cha duy nhất**: Mọi JSX phải được bọc trong một thẻ cha (ví dụ: `<div>` hoặc `<React.Fragment>` / `<>`).
- **Thuộc tính camelCase**: Sử dụng `className` thay vì `class` , `onClick` thay vì `onclick` .
- **Biểu thức JavaScript**: Dùng `{}` để nhúng biểu thức JavaScript, ví dụ `{2 + 2}` hoặc `{user.name}` .
- **Bình luận trong JSX**: Sử dụng `{/* bình luận */}` thay vì `<!-- -->` .

Ví dụ minh họa:

```
const name = "John";
const element = (
  <div className="greeting">
    <h1>Hello, {name}!</h1>
  </div>
);
```

Giải thích bổ sung:
JSX giúp lập trình viên viết giao diện một cách trực quan, giống như viết HTML, nhưng thực chất là JavaScript. Điều này cho phép kết hợp logic (JavaScript) và giao diện (JSX) trong cùng một file, giúp mã dễ đọc và quản lý hơn.

---

4. Component - Trái tim của React

Tư duy Component

React khuyến khích chia nhỏ giao diện thành các **component** độc lập, tái sử dụng được (ví dụ: `Header`, `Button`, `Card`). Mỗi component chịu trách nhiệm cho một phần cụ thể của giao diện, giúp mã dễ bảo trì và mở rộng.

Ví dụ minh họa:
Một trang web có thể được chia thành:

- `Header`: Thanh điều hướng.
- `MainContent`: Nội dung chính.
- `Footer`: Chân trang.

Giải thích bổ sung:
Tư duy component giống như việc lắp ráp LEGO: mỗi mảnh (component) có chức năng riêng, nhưng khi kết hợp lại sẽ tạo thành một ứng dụng hoàn chỉnh. Điều này giúp bạn dễ dàng sửa đổi hoặc thay thế từng phần mà không ảnh hưởng đến toàn bộ hệ thống.

Functional Component

Ngày nay, React chủ yếu sử dụng **functional component** (component dạng hàm), thay vì class component. Một functional component là một hàm JavaScript trả về JSX.

Ví dụ minh họa:

```jsx
function Welcome() {
  return <h1>Chào mừng đến với React!</h1>;
}
```

Props (Thuộc tính)

Props là cách truyền dữ liệu từ component cha sang component con. Props là **chỉ đọc** (read-only), đảm bảo luồng dữ liệu một chiều, tránh tác dụng phụ không mong muốn.

Ví dụ minh họa:

```jsx
function UserCard(props) {
  return (
    <div>
      <h2>{props.name}</h2>
      <p>{props.email}</p>
    </div>
  );
}

// Sử dụng trong component cha
<UserCard name="Alice" email="alice@example.com" />
```

Giải thích bổ sung:
Props giống như tham số của một hàm, cho phép bạn tùy chỉnh component. Ví dụ, cùng một component `UserCard` có thể hiển thị thông tin của nhiều người khác nhau chỉ bằng cách thay đổi giá trị props.

---

5. State và Hook `useState`

State là gì?

State là dữ liệu nội bộ của component, có thể thay đổi theo thời gian (ví dụ: giá trị ô input, trạng thái hiển thị modal). Khi state thay đổi, React tự động re-render component để cập nhật giao diện.

Giải thích bổ sung:
State giống như bộ nhớ của component, lưu trữ thông tin thay đổi theo hành động của người dùng (như nhấn nút, nhập liệu). React đảm bảo giao diện luôn khớp với state hiện tại.

Giới thiệu về Hooks

Hooks là các hàm đặc biệt của React, cho phép functional component sử dụng các tính năng như state, lifecycle, v.v. Hook `useState` là hook cơ bản nhất, dùng để quản lý trạng thái.

Hook `useState`

Cú pháp: `const [state, setState] = useState(initialValue);`

- `state` : Giá trị trạng thái hiện tại.
- `setState` : Hàm để cập nhật trạng thái, kích hoạt re-render.

Ví dụ minh họa:

```
import { useState } from 'react';

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

  return (
    <div>
      <p>Đếm: {count}</p>
      <button onClick={() => setCount(count + 1)}>Tăng</button>
    </div>
  );
}
```

Giải thích bổ sung:
`useState` giống như một hộp chứa giá trị mà component có thể thay đổi. Khi bạn gọi `setState`, React sẽ cập nhật giá trị và vẽ lại giao diện để phản ánh thay đổi đó. Điều quan trọng là `setState` không thay đổi state ngay lập tức mà lên lịch cho việc re-render.

C. Bài tập thực hành

Bài 1: Thiết lập dự án React

Nhiệm vụ: Tạo dự án React mới bằng Vite, sửa `App.jsx` để hiển thị dòng chữ "Xin chào từ React!". **Các bước**:

1. Chạy lệnh:

```
npm create vite@latest my-react-app -- --template react
cd my-react-app
npm install
npm run dev
```

2. Sửa `src/App.jsx`:

```
function App() {
  return <h1>Xin chào từ React!</h1>;
}
export default App;
```

3. Kiểm tra kết quả tại `http://localhost:5173`.

Giải thích bổ sung:
Bài tập này giúp làm quen với quy trình tạo dự án và chạy ứng dụng React. Bạn sẽ thấy

cách Vite nhanh chóng khởi động server phát triển và hiển thị kết quả ngay lập tức.

Bài 2: Tạo Component UserProfileCard

Nhiệm vụ: Tạo component `UserProfileCard.js` nhận các props `name`, `avatarUrl`, `bio`, `email` và hiển thị trong một thẻ có kiểu dáng. **Các bước**:

1. Tạo file `src/components/UserProfileCard.js`:

```
function UserProfileCard({ name, avatarUrl, bio, email }) {
  return (
    <div style={{ border: '1px solid #ccc', padding: '16px', borderRadius: '
      <img src={avatarUrl} alt={name} style={{ width: '100px', borderRadius:
      <h2>{name}</h2>
      <p>{bio}</p>
      <p>Email: {email}</p>
    </div>
  );
}
export default UserProfileCard;
```

Giải thích bổ sung:
Component này sử dụng props để hiển thị thông tin người dùng một cách linh hoạt. Kiểu dáng inline (`style={{...}}`) được dùng để đơn giản hóa, nhưng trong thực tế, bạn nên sử dụng CSS hoặc thư viện như Tailwind CSS.

Bài 3: Sử dụng UserProfileCard trong App

Nhiệm vụ: Sử dụng `UserProfileCard` trong `App.jsx` để hiển thị thông tin của ba người dùng khác nhau. **Các bước**:

1. Sửa `src/App.jsx`:

```jsx
import UserProfileCard from './components/UserProfileCard';

function App() {
  return (
    <div>
      <UserProfileCard
        name="Alice"
        avatarUrl="https://via.placeholder.com/100"
        bio="Lập trình viên giao diện, đam mê thiết kế UI."
        email="alice@example.com"
      />
      <UserProfileCard
        name="Bob"
        avatarUrl="https://via.placeholder.com/100"
        bio="Kỹ sư backend, chuyên về API."
        email="bob@example.com"
      />
      <UserProfileCard
        name="Charlie"
        avatarUrl="https://via.placeholder.com/100"
        bio="Lập trình viên full-stack, yêu công nghệ."
        email="charlie@example.com"
      />
    </div>
  );
}
export default App;
```

Giải thích bổ sung:

Bài tập này giúp bạn hiểu cách tái sử dụng component với các props khác nhau. Mỗi `UserProfileCard` hiển thị dữ liệu riêng, nhưng mã nguồn component không cần thay đổi, thể hiện tính tái sử dụng của React.

Bài 4: Tạo Component Counter

Nhiệm vụ: Tạo component `Counter.js` sử dụng `useState` để quản lý trạng thái `count`, bắt đầu từ 0, với các nút "Tăng" và "Giảm". **Các bước**:

1. Tạo file `src/components/Counter.js`:

```
import { useState } from 'react';

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

  return (
    <div>
      <h2>Đếm: {count}</h2>
      <button onClick={() => setCount(count + 1)}>Tăng</button>
      <button onClick={() => setCount(count - 1)}>Giảm</button>
    </div>
  );
}
export default Counter;
```

2. Sử dụng trong `App.jsx`:

```
import Counter from './components/Counter';

function App() {
  return <Counter />;
}
export default App;
```

Giải thích bổ sung:
Bài tập này giới thiệu cách sử dụng `useState` để quản lý trạng thái động. Mỗi lần nhấn nút, `setCount` cập nhật giá trị `count`, và React tự động re-render để hiển thị giá trị mới.

Bài 5: Nâng cấp UserProfileCard với tính năng Follow/Unfollow

Nhiệm vụ: Thêm trạng thái `isFollowing` vào `UserProfileCard` bằng `useState`. Thêm nút để chuyển đổi giữa "Follow" và "Unfollow". **Các bước**:

1. Sửa `src/components/UserProfileCard.js`:

```
import { useState } from 'react';

function UserProfileCard({ name, avatarUrl, bio, email }) {
  const [isFollowing, setIsFollowing] = useState(false);

  return (
    <div style={{ border: '1px solid #ccc', padding: '16px', borderRadius: '
      <img src={avatarUrl} alt={name} style={{ width: '100px', borderRadius:
      <h2>{name}</h2>
      <p>{bio}</p>
      <p>Email: {email}</p>
      <button onClick={() => setIsFollowing(!isFollowing)}>
        {isFollowing ? 'Bỏ theo dõi' : 'Theo dõi'}
      </button>
    </div>
  );
}
export default UserProfileCard;
```

2. Kiểm tra chức năng trong `App.jsx` với một hoặc nhiều `UserProfileCard`.

Giải thích bổ sung:
Bài tập này kết hợp props và state, giúp bạn hiểu cách component có thể vừa nhận dữ liệu từ bên ngoài (props) vừa quản lý trạng thái nội bộ (state). Nút "Follow/Unfollow" thay đổi trạng thái `isFollowing`, và giao diện tự động cập nhật.

D. Kiểm tra và cập nhật nội dung

Kiểm tra nội dung gốc

Nội dung gốc đã khá đầy đủ và chính xác, nhưng có một số điểm cần cập nhật:

1. **Phiên bản Node.js**: Đề xuất sử dụng Node.js 18 trở lên, nhưng hiện tại (2025), Node.js 20 là phiên bản ổn định mới nhất, nên khuyến nghị sử dụng phiên bản này.
2. **Cú pháp JSX**: Nội dung gốc không đề cập đến việc sử dụng `<React.Fragment>` thay thế `<div>` để tránh thêm thẻ không cần thiết. Điều này quan trọng để tối ưu hóa cấu trúc DOM.
3. **Kiểu dáng**: Các bài tập sử dụng style inline, nên bổ sung khuyến nghị sử dụng CSS modules hoặc Tailwind CSS trong các dự án thực tế để quản lý kiểu dáng tốt hơn.
4. **Hiệu suất**: Phần Virtual DOM có thể bổ sung thêm ví dụ minh họa để làm rõ cách React tối ưu hóa cập nhật.

Bổ sung

- **Khuyến nghị sử dụng Tailwind CSS**: Trong các bài tập thực hành, có thể tích hợp Tailwind CSS để thay thế style inline, giúp mã dễ đọc và bảo trì hơn. Ví dụ, thay vì:

```
<div style={{ border: '1px solid #ccc', padding: '16px', borderRadius: '8px'
```

Sử dụng Tail締

```
<div className="border border-gray-300 p-4 rounded-lg max-w-xs">
```

Để sử dụng Tailwind, thêm CDN hoặc cài đặt qua npm theo tài liệu chính thức.

- **Tối ưu hóa** `<React.Fragment>` : Trong bài tập 3, thay vì bọc các `UserProfileCard` trong `<div>` , có thể sử dụng `<React.Fragment>` để giảm thẻ dư thừa:

```jsx
import UserProfileCard from './components/UserProfileCard';

function App() {
  return (
    <>
      <UserProfileCard name="Alice" avatarUrl="https://via.placeholder.com/1
      <UserProfileCard name="Bob" avatarUrl="https://via.placeholder.com/100
      <UserProfileCard name="Charlie" avatarUrl="https://via.placeholder.com
    </>
  );
}
export default App;
```

- **Khuyến nghị về tổ chức mã**: Trong các dự án thực tế, nên tạo thư mục `components/` riêng để chứa các component như `UserProfileCard` và `Counter`, giúp mã nguồn dễ quản lý hơn.