1. React Là Gì?

React là một thư viện JavaScript mã nguồn mở được phát triển và duy trì bởi Facebook. Nó được thiết kế để xây dựng các giao diện người dùng (UI) trực quan và tương tác, với mục tiêu tạo ra các ứng dụng web hiệu suất cao và dễ bảo trì.

Một trong những đặc điểm nổi bật của React là khái niệm "component-based architecture" (kiến trúc dựa trên thành phần). Thay vì xây dựng một trang web toàn diện, React cho phép chúng ta chia nhỏ giao diện người dùng thành các thành phần độc lập, tái sử dụng được. Mỗi thành phần có thể quản lý trạng thái và logic riêng của mình, giúp cho việc phát triển và bảo trì trở nên dễ dàng hơn.

Ngoài ra, React sử dụng một kỹ thuật gọi là "virtual DOM" (Document Object Model ảo) để tối ưu hóa hiệu suất. Thay vì cập nhật trực tiếp vào DOM thật, React sẽ tạo ra một bản sao ảo của DOM, thực hiện các thay đổi cần thiết ở đó, và sau đó chỉ cập nhật những phần thay đổi trong DOM thật. Điều này giúp giảm thiểu số lượng thao tác trên DOM, từ đó cải thiện tốc độ và hiệu suất của ứng dụng.

Ngoài những tính năng cơ bản trên, React còn có nhiều tính năng và công cụ hữu ích khác, như:

- **JSX**: Một cú pháp mở rộng của JavaScript, cho phép chúng ta viết mã giao diện trực quan và dễ đọc hơn.
- **State Management**: Cung cấp các cơ chế quản lý trạng thái của ứng dụng, giúp duy trì tính nhất quán và đồng bộ trong giao diện.
- **Routing**: Cho phép xây dựng các ứng dụng web đơn trang (SPA) với các trang và URL riêng biệt.
- **Server**-side Rendering (SSR): Hỗ trợ kỹ thuật rendering ở phía máy chủ, giúp cải thiện tốc độ tải trang và SEO.
- **Hooks**: Một tính năng mới giúp quản lý trạng thái và vòng đời của các thành phần một cách hiệu quả hơn.

2. Những Ứng Dụng Thực Tế Được Xây Dựng Bằng React

React không chỉ là một công cụ hữu ích cho các nhà phát triển web, mà nó đã được ứng dụng rộng rãi trong nhiều loại ứng dụng web thực tế. Dưới đây là một số ví dụ về những ứng dụng web lớn và nổi tiếng được xây dựng bằng React:

1. **Facebook**: Trang web và ứng dụng di động của Facebook được xây dựng chủ yếu bằng React. Đây là một ví dụ điển hình về cách React có thể được sử dụng để xây dựng các ứng dụng web quy mô lớn và phức tạp.
2. **Instagram**: Cũng như Facebook, ứng dụng Instagram được phát triển bằng React, đặc biệt là phần giao diện người dùng.
3. **Netflix**: Trang web và ứng dụng di động của Netflix sử dụng React để xây dựng các tính năng như danh sách phim, trang chi tiết phim, và trang cá nhân của người dùng.
4. **Airbnb**: Airbnb, một trong những nền tảng đặt phòng khách sạn lớn nhất thế giới, đã chuyển sang sử dụng React để xây dựng lại giao diện người dùng của trang web và ứng dụng di động.
5. **The New York Times**: Trang web của The New York Times, một trong những tờ báo uy tín nhất thế giới, cũng đã áp dụng React để cải thiện hiệu suất và trải nghiệm người dùng.
6. **Dropbox**: Dropbox, một trong những dịch vụ lưu trữ đám mây phổ biến nhất, sử dụng React để xây dựng các tính năng trên trang web và ứng dụng di động.
7. **Airbnb**: Airbnb, một trong những nền tảng đặt phòng khách sạn lớn nhất thế giới, đã chuyển sang sử dụng React để xây dựng lại giao diện người dùng của trang web và ứng dụng di động.

Như có thể thấy, React đã được áp dụng rộng rãi trong các ứng dụng web quy mô lớn và phức tạp, từ mạng xã hội đến dịch vụ trực tuyến, từ báo chí đến du lịch. Điều này chứng minh sức mạnh và tính linh hoạt của React trong việc xây dựng các ứng dụng web hiện đại và tương tác.

3. Tại Sao Chúng Ta Nên Chọn React?

Với những ưu điểm nổi bật, React đã trở thành một lựa chọn phổ biến cho các nhà phát triển web khi xây dựng các ứng dụng web hiện đại. Dưới đây là một số lý do chính tại sao chúng ta nên chọn React:

1. **Hiệu Suất Cao**: Nhờ sử dụng kỹ thuật Virtual DOM, React có thể tối ưu hóa quá trình cập nhật giao diện, giúp các ứng dụng web chạy nhanh và mượt mà hơn.
2. **Tái Sử Dụng Thành Phần**: Với kiến trúc dựa trên thành phần, React cho phép chúng ta xây dựng các thành phần UI độc lập và tái sử dụng chúng trong các dự án khác. Điều này giúp tiết kiệm thời gian và nâng cao hiệu quả phát triển.
3. **Dễ Học và Sử Dụng**: React có cú pháp đơn giản và dễ hiểu, đặc biệt với những lập trình viên đã quen với JavaScript. Điều này giúp các nhà phát triển nhachóng tiếp cận và làm chủ được công nghệ này.
4. **Cộng Đồng Lớn và Hỗ Trợ Tích Cực**: React có một cộng đồng người dùng rất lớn và hoạt động tích cực. Điều này đảm bảo sự hỗ trợ, tài liệu hướng dẫn, và các công cụ phát triển phong phú.
5. **Linh Hoạt và Mở Rộng**: React là một thư viện mở rộng, cho phép chúng ta kết hợp nó với các thư viện và công nghệ khác để xây dựng các ứng dụng web phức tạp hơn.
6. **Hỗ Trợ Tốt Cho Các Ứng Dụng Di Động**: Với sự ra đời của React Native, lập trình viên có thể sử dụng React để xây dựng các ứng dụng di động đa nền tảng (iOS và Android) với codebase chung.
7. **Dễ Bảo Trì và Mở Rộng**: Nhờ kiến trúc dựa trên thành phần, các ứng dụng React trở nên dễ bảo trì và mở rộng hơn so với các ứng dụng web truyền thống.

Tóm lại, React là một lựa chọn tuyệt vời cho các nhà phát triển web khi xây dựng các ứng dụng web hiện đại, với những ưu điểm về hiệu suất, tính tái sử dụng, và sự linh hoạt. Nó đã được chứng minh là một công nghệ mạnh mẽ và đáng tin cậy trong thực tế.

4. Tìm hiểu SPA & MPA

Khi nói đến ứng dụng web được xây dựng bằng React, chúng ta không thể không nhắc đến khái niệm "ứng dụng web đơn trang" (Single Page Application - SPA). SPA là một loại ứng dụng web mà toàn bộ trang web chỉ gồm một trang duy nhất, và các trang/view khác được tải động bằng JavaScript thay vì tải lại toàn bộ trang web.

Ngược lại, ứng dụng web truyền thống (Multi-Page Application - MPA) là những ứng dụng web có nhiều trang riêng biệt, mỗi trang được tải lại hoàn toàn khi người dùng chuyển đến trang khác.

Dưới đây là một số điểm khác biệt chính giữa SPA và MPA:

1. **Tốc Độ Tải Trang**: Các ứng dụng SPA thường tải nhanh hơn MPA, vì chỉ cần tải lại một phần nhỏ của trang web thay vì tải lại toàn bộ.
2. **Trải Nghiệm Người Dùng**: SPA cung cấp trải nghiệm người dùng liền mạch, với các chuyển đổi giữa các trang/view diễn ra mượt mà. MPA có thể cảm thấy "giật lag" khi chuyển đổi giữa các trang.
3. **Cấu Trúc Ứng Dụng**: SPA có cấu trúc ứng dụng dựa trên JavaScript và API, trong khi MPA có cấu trúc ứng dụng dựa trên các trang web truyền thống.
4. **Khả Năng Tìm Kiếm**: MPA thường dễ được tìm kiếm và lập chỉ mục bởi công cụ tìm kiếm hơn SPA, vì các trang web truyền thống dễ được "đọc" hơn.
5. **Khả Năng Phát Triển**: SPA yêu cầu nhiều kỹ năng lập trình JavaScript và quản lý trạng thái, trong khi MPA dễ phát triển hơn với kiến thức HTML, CSS và server-side.

Mặc dù SPA và MPA có những ưu và nhược điểm khác nhau, nhưng React là một công cụ rất phù hợp để xây dựng các ứng dụng web đơn trang (SPA). Nhờ các tính năng như Virtual DOM, routing và quản lý trạng thái, React giúp các nhà phát triển xây dựng các SPA hiệu suất cao, linh hoạt và dễ bảo trì.

5. Create React App

Create React App (CRA) là một công cụ tiện ích giúp bạn nhanh chóng khởi tạo một ứng dụng React mà không cần cấu hình phức tạp. Được phát triển bởi Facebook, CRA cung cấp một bộ công cụ sẵn sàng để sử dụng, cho phép bạn tập trung vào việc phát triển ứng dụng mà không phải lo lắng về việc thiết lập môi trường phát triển.

Trang chủ của Create React App có thể tìm thấy tại https://create-react-app.dev . Tại đây, bạn có thể tìm thấy tài liệu hướng dẫn chi tiết, thông tin về các tính năng và cách sử dụng CRA.

Cài đặt Create React App

Để cài đặt Create React App, bạn chỉ cần mở terminal và chạy lệnh sau:

```
npx create-react-app my-app
```

Lệnh này sẽ tạo một thư mục mới có tên "my-app" và cài đặt tất cả các tệp cần thiết để bắt đầu một ứng dụng React. Sau khi cài đặt hoàn tất, bạn có thể chuyển vào thư mục dự án và khởi động ứng dụng bằng lệnh:

```
cd my-app
npm start
```

Cấu trúc thư mục và ý nghĩa của từng tệp/folder

Khi Create React App hoàn tất việc cài đặt, bạn sẽ thấy cấu trúc thư mục như sau:

```
my-app
├── node_modules
├── public
│   ├── favicon.ico
│   ├── index.html
│   ├── logo192.png
│   ├── logo512.png
│   ├── manifest.json
│   └── robots.txt
├── src
│   ├── App.css
│   ├── App.js
│   ├── App.test.js
│   ├── index.css
│   ├── index.js
│   └── logo.svg
├── .gitignore
├── package.json
├── README.md
└── yarn.lock (hoặc package-lock.json)
```

node_modules

Thư mục này chứa tất cả các gói (packages) mà ứng dụng của bạn phụ thuộc vào. Đây là nơi mà **npm** hay **yarn** tải các thư viện mà bạn sử dụng trong dự án.

public

Thư mục này chứa các tệp tĩnh mà bạn cần trong ứng dụng. Các tệp trong thư mục này sẽ được phục vụ trực tiếp. Một số tệp quan trọng bao gồm:

- **favicon.ico**: Biểu tượng của trang web.
- **index.html**: Tệp HTML chính cho ứng dụng. Đây là nơi mà React sẽ chèn các thành phần của mình.
- **logo192.png**, **logo512.png**: Hình ảnh logo cho ứng dụng.
- **manifest.json**: Tệp cấu hình cho Progressive Web App (PWA).
- **robots.txt**: Tệp này hướng dẫn các công cụ tìm kiếm cách quét trang web của bạn.

src

Thư mục này là nơi chứa mã nguồn chính của ứng dụng. Bạn sẽ thực hiện hầu hết các công việc phát triển ở đây. Một số tệp quan trọng trong thư mục này bao gồm:

- **App.css**: Tệp CSS cho thành phần chính App.
- **App.js**: Thành phần chính của ứng dụng, nơi định nghĩa giao diện người dùng.
- **App.test.js**: Tệp kiểm tra cho thành phần App, giúp bạn viết các bài kiểm tra đơn vị.
- **index.css**: Tệp CSS cho toàn bộ ứng dụng.
- **index.js**: Tệp nhập khẩu chính, nơi React được khởi động và kết nối với DOM. Đây là tệp đầu tiên được chạy khi bạn khởi động ứng dụng.
- **logo.svg**: Hình ảnh logo SVG sử dụng trong ứng dụng.

.gitignore

Tệp này chỉ định những tệp và thư mục nào nên bị bỏ qua khi sử dụng Git. Thường thì các thư mục như node_modules và các tệp build sẽ được liệt kê tại đây.

package.json

Tệp này chứa thông tin về dự án của bạn, bao gồm tên, phiên bản, mô tả, các phụ thuộc (dependencies), và các lệnh scripts để chạy và xây dựng ứng dụng. Đây là nơi quản lý tất cả các gói mà bạn đã cài đặt.

README.md

Tệp này chứa hướng dẫn và thông tin về dự án của bạn. Bạn có thể sử dụng nó để ghi chú lại cách thiết lập, chạy, và phát triển ứng dụng.

yarn.lock hoặc package-lock.json

Tệp này được tạo ra khi bạn sử dụng Yarn hoặc npm để quản lý phụ thuộc. Nó ghi lại các phiên bản cụ thể của các gói mà dự án đang sử dụng, giúp đảm bảo rằng mọi người đều có cùng một môi trường phát triển.

6. Component-based Architecture

Một trong những nguyên tắc cốt lõi của React là kiến trúc dựa trên thành phần (component-based architecture). Trong kiến trúc này, ứng dụng được chia thành các thành phần (component) độc lập và có thể tái sử dụng. Mỗi thành phần có trách nhiệm riêng biệt và có thể được quản lý và phát triển một cách độc lập.

Lợi ích chính của kiến trúc dựa trên thành phần bao gồm:

- **Tái sử dụng code**: Các thành phần có thể được tái sử dụng trong nhiều nơi khác nhau trong ứng dụng, giúp tiết kiệm thời gian và nâng cao hiệu quả phát triển.
- **Dễ dàng bảo trì và mở rộng**: Khi ứng dụng phát triển, việc thay đổi hoặc bổ sung tính năng trở nên dễ dàng hơn vì mỗi thành phần có trách nhiệm riêng biệt.
- **Tăng tính mô-đun hóa**: Chia ứng dụng thành các thành phần độc lập giúp mã nguồn trở nên rõ ràng, dễ hiểu và dễ quản lý hơn.
- **Cải thiện hiệu suất**: Các thành phần có thể được tải và cập nhật độc lập, giúp tăng tốc độ tải trang và cải thiện trải nghiệm người dùng.

Ví dụ về một ứng dụng React dựa trên kiến trúc thành phần có thể bao gồm các thành phần như Header, Sidebar, MainContent, Footer, Button, Modal, v.v. Mỗi thành phần có trách nhiệm riêng biệt và có thể được phát triển, kiểm tra và triển khai một cách độc lập.

7. Function Component

Trong React, có hai cách chính để định nghĩa các thành phần: **Function Component** và **Class Component**. Trong những năm gần đây, Function Component đã trở thành cách tiếp cận phổ biến hơn vì nó mang lại nhiều lợi ích:

- **Cú pháp đơn giản và dễ đọc**: Function Component sử dụng cú pháp JavaScript thông thường, không yêu cầu định nghĩa lớp (class) hoặc sử dụng từ khóa this. Điều này giúp mã nguồn trở nên dễ đọc và dễ hiểu hơn.
- **Hooks**: React Hooks là một tính năng mới được giới thiệu trong phiên bản 16.8, cho phép bạn sử dụng các tính năng của React như state, lifecycle, và các tính năng khác mà không cần định nghĩa lớp. Hooks giúp mã nguồn trở nên gọn gàng và dễ bảo trì hơn.
- **Hiệu suất tốt hơn**: Function Component thường có hiệu suất tốt hơn so với Class Component, đặc biệt là khi sử dụng Hooks.
- **Dễ dàng học hỏi**: Với cú pháp đơn giản và không yêu cầu kiến thức về lập trình hướng đối tượng, Function Component dễ dàng tiếp cận hơn cho những lập trình viên mới bắt đầu.

Dưới đây là một ví dụ về một Function Component đơn giản trong React:

```
import React from "react";

function HelloWorld() {
  return (
    <div>
      <h1>Hello, World!</h1>
      <p>This is a simple React component.</p>
    </div>
  );
}

export default HelloWorld;
```

Trong ví dụ này, HelloWorld là một Function Component. Nó định nghĩa một hàm JavaScript thông thường, trả về một đoạn JSX (một cú pháp mở rộng của JavaScript để mô tả giao diện người dùng). Sau đó, thành phần này có thể được sử dụng trong các thành phần khác của ứng dụng.

8. JSX

JSX là một cú pháp mở rộng của JavaScript, cho phép bạn viết mã giao diện người dùng theo cách trực quan và dễ đọc hơn. Thay vì sử dụng các hàm createElement() và appendChild() như trong JavaScript thuần túy, JSX cho phép bạn sử dụng cú pháp tương tự HTML để tạo ra các thành phần giao diện.

Ví dụ, thay vì viết:

```
return React.createElement("div", { className: "container" }, [
  React.createElement("h1", null, "Hello, World!"),
  React.createElement("p", null, "This is a simple React component."),
]);
```

Bạn có thể viết:

```
return (
  <div className="container">
    <h1>Hello, World!</h1>
    <p>This is a simple React component.</p>
  </div>
);
```

Cú pháp JSX rõ ràng và dễ đọc hơn nhiều, giúp bạn tập trung vào việc xây dựng giao diện người dùng thay vì phải viết mã JavaScript phức tạp.

Ngoài ra, JSX cũng cho phép bạn sử dụng biểu thức JavaScript trong nội dung của các thành phần. Ví dụ:

```
function HelloWorld(props) {
  const name = props.name || "World";
  return (
    <div>
      <h1>Hello, {name}!</h1>
      <p>Today is {new Date().toLocaleDateString()}.</p>
    </div>
  );
}
```

Trong ví dụ này, biểu thức {name} sẽ được thay thế bằng giá trị của props.name, và biểu thức {new Date().toLocaleDateString()} sẽ hiển thị ngày hiện tại.

JSX không chỉ giúp mã nguồn trở nên dễ đọc và dễ hiểu hơn, mà còn tạo ra một cách tiếp cận trực quan và gần gũi hơn với thiết kế giao diện người dùng. Điều này giúp lập trình viên và nhà thiết kế giao diện có thể hợp tác chặt chẽ hơn trong quá trình phát triển ứng dụng.

9. Props vs State

Trong React, "Props" (viết tắt của "Properties") và "State" là hai cách chính để quản lý và cập nhật dữ liệu trong các thành phần của ứng dụng.

- Props
  - Props là các thuộc tính được truyền vào một thành phần React từ bên ngoài.
  - Chúng đại diện cho dữ liệu đầu vào mà thành phần sẽ sử dụng.
  - Props là read-only, nghĩa là thành phần không thể thay đổi giá trị của chúng.
  - Các thành phần con nhận props từ thành phần cha và sử dụng chúng để hiển thị nội dung.
  - Props giúp tái sử dụng các thành phần bằng cách cho phép chúng nhận các dữ liệu khác nhau.
- State
  - State là dữ liệu nội bộ của một thành phần.
  - Nó biểu diễn trạng thái hiện tại của thành phần và có thể thay đổi theo thời gian.
  - State được quản lý bên trong thành phần và có thể được cập nhật thông qua các phương thức cụ thể.
  - Khi state thay đổi, React sẽ tự động re-render thành phần để phản ánh các thay đổi đó.
  - State cho phép các thành phần có hành vi động và tương tác.

Tóm lại, props là dữ liệu được truyền vào một thành phần từ bên ngoài, trong khi state là dữ liệu nội bộ của thành phần và có thể thay đổi theo thời gian. Props là read-only, trong khi state có thể được cập nhật bởi thành phần.

10. Làm việc với Props

Để sử dụng props trong React, bạn cần phải truyền chúng vào thành phần và sau đó truy cập chúng bên trong thành phần.

Ví dụ, giả sử chúng ta có một thành phần Greeting nhận một prop name:

```
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}

// Sử dụng thành phần Greeting
<Greeting name="John" />;
```

Trong ví dụ này, chúng ta truyền prop name với giá trị "John" vào thành phần Greeting. Bên trong thành phần, chúng ta có thể truy cập prop name thông qua props.name.

Bạn cũng có thể sử dụng cú pháp giải cấu trúc (destructuring) để truy cập props:

```
function Greeting({ name }) {
  return <h1>Hello, {name}!</h1>;
}

// Sử dụng thành phần Greeting
<Greeting name="John" />;
```

Trong ví dụ này, chúng ta đã giải cấu trúc prop name từ props và sử dụng trực tiếp name trong nội dung trả về.

Ngoài ra, bạn có thể truyền nhiều props vào một thành phần:

```
function PersonCard(props) {
  return (
    <div>
      <h2>{props.name}</h2>
      <p>Age: {props.age}</p>
      <p>Email: {props.email}</p>
    </div>
  );
}

// Sử dụng thành phần PersonCard
<PersonCard name="John Doe" age={30} email="john@example.com" />;
```

Trong ví dụ này, chúng ta truyền ba props (name, age, email) vào thành phần PersonCard và sử dụng chúng để hiển thị thông tin của một người.

Lưu ý rằng props là read-only, do đó thành phần không thể trực tiếp thay đổi giá trị của chúng. Nếu bạn cần thay đổi dữ liệu, bạn cần sử dụng state.

11. Lists and Keys

Khi làm việc với React, bạn thường sẽ cần hiển thị danh sách các thành phần. Để làm điều này, bạn có thể sử dụng mảng và map() để tạo ra các thành phần tương ứng.

Ví dụ, giả sử chúng ta có một mảng các người dùng và muốn hiển thị chúng trong một danh sách:

```
const users = [
  { id: 1, name: "John Doe", email: "john@example.com" },
  { id: 2, name: "Jane Smith", email: "jane@example.com" },
  { id: 3, name: "Bob Johnson", email: "bob@example.com" },
];

function UserList() {
  return (
    <ul>
      {users.map((user) => (
        <li key={user.id}>
          <h3>{user.name}</h3>
          <p>Email: {user.email}</p>
        </li>
      ))}
    </ul>
  );
}
```

Trong ví dụ này, chúng ta sử dụng map() để lặp qua mảng users và tạo ra các thành phần li tương ứng. Lưu ý rằng chúng ta cung cấp một key duy nhất cho mỗi thành phần li, được lấy từ user.id.

Khóa (key) là một thuộc tính đặc biệt trong React, được sử dụng để giúp React theo dõi và quản lý các thành phần trong danh sách. Khi một danh sách thay đổi (thêm, xóa hoặc sắp xếp lại các phần tử), React sẽ sử dụng khóa để xác định các thành phần nào cần được cập nhật, thêm hoặc xóa.

Việc cung cấp khóa duy nhất và ổn định cho mỗi thành phần trong danh sách là rất quan trọng để giúp React hiệu quả hơn trong việc quản lý và cập nhật các thành phần. Nếu không có khóa, React sẽ không thể xác định được các thành phần nào đã thay đổi và sẽ phải thực hiện nhiều công việc không cần thiết, ảnh hưởng đến hiệu suất của ứng dụng.

12. Events and Events Handlers

Trong React, bạn có thể thêm các sự kiện vào các thành phần của mình và xử lý chúng bằng cách sử dụng bộ xử lý sự kiện.

Ví dụ, giả sử chúng ta có một nút "Click me" và muốn hiển thị một thông báo khi nút được nhấp:

```
function ClickButton() {
  function handleClick() {
    alert("You clicked the button!");
  }

  return <button onClick={handleClick}>Click me</button>;
}
```

Trong ví dụ này, chúng ta định nghĩa một hàm handleClick() để xử lý sự kiện onClick của nút. Khi người dùng nhấp vào nút, hàm handleClick() sẽ được gọi và hiển thị một thông báo.

Bạn cũng có thể truyền các tham số vào bộ xử lý sự kiện:

```
function Counter() {
  let count = 0;

  function incrementCount(amount) {
    count += amount;
    console.log(`Count is now ${count}`);
  }

  return (
    <div>
      <button onClick={() => incrementCount(1)}>Increment</button>
      <button onClick={() => incrementCount(5)}>Increment by 5</button>
    </div>
  );
}
```

Trong ví dụ này, chúng ta định nghĩa một hàm incrementCount() nhận một tham số amount. Khi người dùng nhấp vào nút "Increment", hàm incrementCount(1) sẽ được gọi, tăng biến count lên 1. Tương tự, khi người dùng nhấp vào nút "Increment by 5", hàm incrementCount(5) sẽ được gọi, tăng biến count lên 5.

13. Component life-cycle

Vòng đời của một component trong React được chia thành các giai đoạn sau:

Mounting

Giai đoạn này diễn ra khi một instance của component được tạo ra và được chèn vào DOM. Các phương thức sau được gọi theo thứ tự:

- constructor()
- render()
- componentDidMount()

Updating

Giai đoạn này diễn ra khi component được cập nhật do một số thay đổi, chẳng hạn như props thay đổi hoặc setState() được gọi. Các phương thức sau được gọi:

- shouldComponentUpdate()
- render()
- getSnapshotBeforeUpdate()
- componentDidUpdate()

Unmounting

Giai đoạn này diễn ra khi component bị xóa khỏi DOM. Phương thức sau sẽ được gọi

- componentWillUnmount()

Thực tiễn

Hiểu rõ các giai đoạn trong vòng đời của component sẽ giúp bạn quản lý trạng thái và thực hiện các tác vụ side effect một cách hiệu quả hơn. Ví dụ, bạn có thể sử dụng componentDidMount() để thực hiện các tác vụ như gọi API, đăng ký các event listener, hoặc khởi tạo các thư viện bên ngoài. Và componentWillUnmount() được sử dụng để dọn dẹp các tài nguyên khi component bị xóa.

14. Hook useState

Trước khi React Hooks được giới thiệu, việc quản lý trạng thái của component trong React thường được thực hiện thông qua class component và phương thức setState(). Tuy nhiên, với sự ra đời của Hooks, việc quản lý trạng thái trở nên dễ dàng hơn nhiều.

Hook useState là một trong những Hook cơ bản và phổ biến nhất trong React. Nó cho phép bạn thêm trạng thái vào các functional component. Trước đây, chỉ có class component mới có thể quản lý trạng thái, nhưng với useState, functional component cũng có thể làm được điều này.

Cách sử dụng useState như sau:

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

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

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

Trong ví dụ trên, useState(0) khởi tạo một state variable count với giá trị ban đầu là 0. setCount là một function được trả về bởi useState, cho phép bạn cập nhật giá trị của count.

Khi người dùng nhấn nút "Click me", setCount(count + 1) được gọi, điều này sẽ làm cho component được re-render với giá trị mới của count.

Một số lưu ý khi sử dụng useState:

- useState có thể được sử dụng nhiều lần trong cùng một functional component để quản lý nhiều trạng thái khác nhau.
- Giá trị khởi tạo được truyền vào useState() chỉ được sử dụng trong lần gọi đầu tiên. Sau đó, React sẽ lưu trữ và quản lý giá trị trạng thái.
- Khi gọi setCount(), React sẽ tự động re-render component với giá trị trạng thái mới.

Việc sử dụng useState giúp code của bạn trở nên gọn gàng, dễ đọc và dễ bảo trì hơn so với việc sử dụng class component và setState().

15. Hook useEffect

Ngoài useState, Hook useEffect là một Hook khác rất quan trọng và phổ biến trong React. useEffect cho phép bạn thực hiện các tác vụ side effect trong functional component.

Các tác vụ side effect có thể bao gồm:

- Gọi API để lấy dữ liệu
- Đăng ký và hủy đăng ký các event listener
- Thay đổi trực tiếp DOM
- Và bất kỳ tác vụ nào có thể gây ra các tác dụng phụ ngoài quá trình render component

Cách sử dụng useEffect như sau:

```
import { useState, useEffect } from "react";

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

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

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

Trong ví dụ trên, useEffect được gọi sau mỗi lần render component. Nó sẽ thực hiện tác vụ cập nhật tiêu đề trang dựa trên giá trị của count.

Lưu ý rằng useEffect có thể trả về một function cleanup, được gọi trước khi component bị unmount hoặc trước khi effect chạy lại trong các lần render tiếp theo. Đây là nơi bạn có thể dọn dẹp các tài nguyên, như hủy đăng ký các event listener.

```
useEffect(() => {
  const handleResize = () => {
    setWindowWidth(window.innerWidth);
  };

  window.addEventListener("resize", handleResize);

  return () => {
    window.removeEventListener("resize", handleResize);
  };
}, []);
```

Trong ví dụ trên, effect đăng ký một event listener resize trên window. Hàm cleanup được trả về sẽ hủy đăng ký event listener này trước khi component bị unmount.

Ngoài ra, useEffect cũng cho phép bạn truyền vào một mảng dependencies (danh sách các biến) làm tham số thứ hai. Điều này sẽ cho phép React biết khi nào effect cần được thực hiện lại. Nếu mảng dependencies là rỗng [], effect sẽ chỉ được thực hiện một lần duy nhất, khi component được mount.

```
useEffect(() => {
  // Thực hiện tác vụ side effect
}, [count, someOtherState]);
```

Trong ví dụ trên, effect sẽ được thực hiện lại mỗi khi count hoặc someOtherState thay đổi.

Sử dụng useEffect hợp lý sẽ giúp bạn quản lý các tác vụ side effect một cách hiệu quả, tránh các vấn đề như memory leak hoặc các tác dụng phụ không mong muốn.

Kết hợp useState và useEffect

Thông thường, useState và useEffect được sử dụng kết hợp với nhau để quản lý trạng thái và thực hiện các tác vụ side effect.

Ví dụ, hãy xem một ứng dụng đơn giản hiển thị danh sách các bài viết từ một API:

```
import { useState, useEffect } from "react";

function BlogPosts() {
  const [posts, setPosts] = useState([]);

  useEffect(() => {
    async function fetchPosts() {
      const response = await fetch("/api/posts");
      const data = await response.json();
      setPosts(data);
    }
    fetchPosts();
  }, []);

  return (
    <div>
      <h1>Blog Posts</h1>
      <ul>
        {posts.map((post) => (
          <li key={post.id}>{post.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

Trong ví dụ này:

- useState được sử dụng để quản lý trạng thái posts, khởi tạo với một mảng rỗng.
- useEffect được sử dụng để gọi API và lấy danh sách bài viết. Lưu ý rằng mảng dependencies là rỗng [], vì tác vụ này chỉ cần được thực hiện một lần, khi component được mount.
- Khi dữ liệu được lấy về, setPosts được gọi để cập nhật trạng thái posts, điều này sẽ khiến component được re-render với danh sách bài viết mới.

Ví dụ này minh họa cách sử dụng useState và useEffect một cách hiệu quả để quản lý trạng thái và thực hiện các tác vụ side effect trong một functional component.

20. Routing and SPAs

Trong các ứng dụng web truyền thống, mỗi trang được tải lại khi người dùng chuyển đến một URL khác. Điều này dẫn đến sự gián đoạn trong trải nghiệm người dùng và ảnh hưởng đến tốc độ tải trang.

Ứng dụng SPA (Single Page Application) đã giải quyết vấn đề này bằng cách sử dụng JavaScript để quản lý việc chuyển đổi giữa các trang mà không cần tải lại toàn bộ trang web. Thay vào đó, chỉ những phần cần thiết được cập nhật, giúp tăng tốc độ tải trang và cải thiện trải nghiệm người dùng.

Định tuyến là một phần quan trọng trong việc xây dựng các ứng dụng SPA. Nó cho phép người dùng chuyển đổi giữa các trang mà không cần tải lại toàn bộ trang web. React Router DOM v6 là một thư viện định tuyến phổ biến trong React, cung cấp các công cụ và tính năng cần thiết để quản lý định tuyến trong các ứng dụng SPA.

21. Setting Up the Router

Để bắt đầu sử dụng React Router DOM v6, trước tiên bạn cần cài đặt thư viện này vào dự án của mình:

```
npm install react-router-dom
```

Tiếp theo, hãy nhập các thành phần cần thiết từ thư viện:

```
import { BrowserRouter as Router, Routes, Route, Link } from "react-router-dom";
```

Trong đó:

- BrowserRouter as Router: Cung cấp ngữ cảnh định tuyến cho ứng dụng của bạn.
- Routes: Quản lý các định tuyến trong ứng dụng.
- Route: Định nghĩa một định tuyến cụ thể.
- Link: Cung cấp các liên kết để chuyển đổi giữa các trang.

Tiếp theo, hãy tạo cấu trúc cơ bản của bộ định tuyến:

```
function App() {
  return (
    <Router>
      <nav>
        <ul>
          <li>
            <Link to="/">Home</Link>
          </li>
          <li>
            <Link to="/about">About</Link>
          </li>
          <li>
            <Link to="/contact">Contact</Link>
          </li>
        </ul>
      </nav>

      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </Router>
  );
}
```

Trong đoạn code trên:

- Chúng ta bao bọc toàn bộ ứng dụng trong thành phần Router, cung cấp ngữ cảnh định tuyến.
- Trong phần nav, chúng ta sử dụng thành phần Link để tạo các liên kết giữa các trang.
- Trong phần Routes, chúng ta định nghĩa các định tuyến cụ thể, liên kết các URL với các thành phần tương ứng (Home, About, Contact).

Khi người dùng nhấp vào các liên kết trong phần nav, React Router DOM v6 sẽ tự động chuyển đổi giữa các trang mà không cần tải lại toàn bộ trang web.

22. Switching Between Pages

Với cấu trúc bộ định tuyến đã thiết lập, bây giờ chúng ta có thể tạo các thành phần tương ứng với mỗi trang:

```
function Home() {
  return (
    <div>
      <h1>Welcome to the Home page</h1>
      <p>This is the content of the Home page.</p>
    </div>
  );
}

function About() {
  return (
    <div>
      <h1>About Us</h1>
      <p>This is the content of the About page.</p>
    </div>
  );
}

function Contact() {
  return (
    <div>
      <h1>Contact Us</h1>
      <p>This is the content of the Contact page.</p>
    </div>
  );
}
```

Khi người dùng nhấp vào các liên kết trong phần nav, React Router DOM v6 sẽ tự động hiển thị các thành phần tương ứng với mỗi trang. Điều này giúp tạo ra trải nghiệm người dùng liền mạch, mà không cần tải lại toàn bộ trang web.

23. Using Links to Switch Pages

Trong React Router DOM v6, chúng ta sử dụng thành phần Link để tạo các liên kết giữa các trang. Thành phần này sẽ thay thế các liên kết HTML truyền thống (link) và cung cấp các tính năng bổ sung.

Ví dụ, khi người dùng nhấp vào một liên kết Link, React Router DOM v6 sẽ thực hiện chuyển đổi trang mà không làm tải lại toàn bộ trang web. Điều này giúp tăng tốc độ và cải thiện trải nghiệm người dùng.

Ngoài ra, Link còn hỗ trợ các tính năng như:

- to: Xác định đường dẫn đến trang mà người dùng sẽ được chuyển đến.
- replace: Thay thế lịch sử điều hướng thay vì thêm vào.
- state: Chuyển dữ liệu trạng thái đến trang mới.

Ví dụ sử dụng Link:

```
function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/contact">Contact</Link>
            </li>
          </ul>
        </nav>

        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />} />
        </Routes>
      </div>
    </Router>
  );
}
```

Trong đoạn code trên, chúng ta sử dụng thành phần Link để tạo các liên kết giữa các trang. Khi người dùng nhấp vào một liên kết, React Router DOM v6 sẽ tự động chuyển đổi đến trang tương ứng mà không cần tải lại toàn bộ trang web.

24. URL Params

URL Params là một cách để truyền dữ liệu qua URL của ứng dụng React. Chúng ta có thể sử dụng URL Params để truyền các thông tin động như ID của một sản phẩm, tên người dùng, v.v. Điều này giúp ứng dụng của bạn trở nên linh hoạt và dễ dàng thích ứng với các yêu cầu khác nhau.

Để sử dụng URL Params trong React Router DOM v6, bạn cần làm như sau:

Định nghĩa route với URL Params:

```
<Route path="/products/:productId" element={<ProductDetails />} />
```

Ở đây, :productId là tham số động trong URL. Khi người dùng truy cập vào đường dẫn /products/123, giá trị 123 sẽ được lưu vào productId.

Truy cập và sử dụng URL Params trong component:

```
import { useParams } from "react-router-dom";

function ProductDetails() {
  const { productId } = useParams();

  return (
    <div>
      <h1>Product Details</h1>
      <p>Product ID: {productId}</p>
    </div>
  );
}
```

Bằng cách sử dụng hook useParams(), chúng ta có thể truy cập vào giá trị của URL Params và sử dụng nó trong component.

URL Params rất hữu ích khi bạn cần hiển thị thông tin chi tiết của một đối tượng cụ thể, như trang chi tiết sản phẩm, trang hồ sơ người dùng, v.v.

25. Query Params

Query Params là một cách khác để truyền dữ liệu qua URL của ứng dụng React. Chúng thường được sử dụng để truyền các tham số tùy chỉnh, chẳng hạn như các bộ lọc, sắp xếp, phân trang, v.v.

Để sử dụng Query Params trong React Router DOM v6, bạn có thể làm như sau:

Tạo URL với Query Params:

```
<Link to="/products?category=electronics&sort=price-asc">
  Electronics Products
</Link>
```

Ở đây, category và sort là các Query Params.

Truy cập và sử dụng Query Params trong component:

```
import { useSearchParams } from "react-router-dom";

function ProductList() {
  const [searchParams] = useSearchParams();
  const category = searchParams.get("category");
  const sort = searchParams.get("sort");

  return (
    <div>
      <h1>Product List</h1>
      <p>Category: {category}</p>
      <p>Sort: {sort}</p>
      {/* Hiển thị danh sách sản phẩm dựa trên category và sort */}
    </div>
  );
}
```

Bằng cách sử dụng hook useSearchParams(), chúng ta có thể truy cập vào các Query Params và sử dụng chúng trong component.

Query Params rất hữu ích khi bạn cần truyền các tham số tùy chỉnh để lọc, sắp xếp hoặc phân trang dữ liệu trong ứng dụng của mình.

26. Navigating Programmatically

Trong một số trường hợp, bạn cần điều hướng người dùng đến một URL cụ thể dựa trên một sự kiện hoặc hành động của người dùng, chẳng hạn như nhấp vào một nút, gửi một biểu mẫu, v.v. Đây là lúc bạn cần sử dụng Navigating Programmatically.

Để điều hướng người dùng programmatically trong React Router DOM v6, bạn có thể sử dụng hook useNavigate():

```
import { useNavigate } from "react-router-dom";

function ProductForm() {
  const navigate = useNavigate();

  const handleSubmit = (event) => {
    event.preventDefault();
    // Xử lý logic form ở đây
    navigate("/products");
  };

  return (
    <form onSubmit={handleSubmit}>
      {/* Các trường form */}
      <button type="submit">Save</button>
    </form>
  );
}
```

Trong ví dụ trên, khi người dùng nhấp vào nút "Save", hàm handleSubmit() sẽ được gọi. Sau khi xử lý logic của form, chúng ta sử dụng navigate('/products') để điều hướng người dùng đến trang /products.

Bạn cũng có thể truyền các tham số vào URL khi điều hướng:

```
navigate(`/products/${productId}`);
navigate("/products", { state: { category: "electronics" } });
```

Việc điều hướng programmatically rất hữu ích khi bạn muốn kiểm soát luồng điều hướng của ứng dụng dựa trên các sự kiện hoặc hành động của người dùng.

27. Nested Routes

Nested Routes là một tính năng quan trọng của React Router DOM v6 cho phép bạn tổ chức cấu trúc URL của ứng dụng một cách logic và linh hoạt. Nested Routes cho phép bạn đặt các route con bên trong các route cha, tạo ra một cấu trúc phân cấp.

Ví dụ, hãy xem cách bạn có thể sử dụng Nested Routes để tạo một cấu trúc như sau:

```
/products
  /products/123
    /products/123/details
    /products/123/reviews
  /products/456
    /products/456/details
    /products/456/reviews
```

Để thiết lập Nested Routes, bạn cần sử dụng thẻ trong component cha để chỉ định nơi các component con sẽ được hiển thị:

```
// App.js
import { Route, Routes } from "react-router-dom";
import ProductList from "./ProductList";
import ProductDetails from "./ProductDetails";
import ProductDetailsTab from "./ProductDetailsTab";
import ProductReviews from "./ProductReviews";

function App() {
  return (
    <Routes>
      <Route path="/products" element={<ProductList />}>
        <Route path=":productId" element={<ProductDetails />}>
          <Route path="details" element={<ProductDetailsTab />} />
          <Route path="reviews" element={<ProductReviews />} />
        </Route>
      </Route>
    </Routes>
  );
}
```

Trong ví dụ trên, chúng ta có route cha /products và các route con bên trong nó. Khi người dùng truy cập /products/123, component ProductDetails sẽ được hiển thị. Bên trong ProductDetails, chúng ta có thể hiển thị các tab "Details" và "Reviews" bằng cách sử dụng thẻ .

Nested Routes giúp bạn tổ chức cấu trúc URL của ứng dụng một cách logic và dễ quản lý. Chúng cũng cho phép bạn chia nhỏ các component thành các phần nhỏ hơn, tăng tính tái sử dụng và dễ bảo trì.

28. Working with Guards

Trong React Router DOM v6, chúng ta có thể sử dụng các "guards" để bảo vệ các tuyến đường khỏi truy cập trái phép. Guards là các hàm hoặc thành phần được sử dụng để kiểm tra điều kiện trước khi cho phép người dùng truy cập vào một tuyến đường cụ thể.

Một ví dụ điển hình là bảo vệ các tuyến đường yêu cầu người dùng đăng nhập. Chúng ta có thể sử dụng một guard để kiểm tra xem người dùng đã đăng nhập hay chưa trước khi cho phép họ truy cập vào tuyến đường đó.

Để triển khai một guard trong React Router DOM v6, chúng ta có thể sử dụng thành phần và useNavigate() hook. Đây là một ví dụ về cách thiết lập một guard để bảo vệ một tuyến đường yêu cầu đăng nhập:

```
import { Navigate, Outlet, useNavigate } from "react-router-dom";

const PrivateRoute = () => {
  const navigate = useNavigate();
  const isLoggedIn = localStorage.getItem("isLoggedIn");

  // Kiểm tra xem người dùng đã đăng nhập hay chưa
  if (!isLoggedIn) {
    // Nếu chưa đăng nhập, chuyển hướng đến trang đăng nhập
    return <Navigate to="/login" />;
  }

  // Nếu đã đăng nhập, hiển thị nội dung được bảo vệ
  return <Outlet />;
};

// Sử dụng guard trong cấu hình định tuyến
const routes = [
  {
    path: "/dashboard",
    element: (
      <PrivateRoute>
        <Dashboard />
      </PrivateRoute>
    ),
  },
  {
    path: "/settings",
    element: (
      <PrivateRoute>
        <Settings />
      </PrivateRoute>
    ),
  },
];
```

Trong ví dụ trên, chúng ta định nghĩa một thành phần PrivateRoute như một guard. Thành phần này kiểm tra xem người dùng đã đăng nhập hay chưa bằng cách kiểm tra giá trị isLoggedIn trong localStorage. Nếu người dùng chưa đăng nhập, họ sẽ được chuyển hướng đến trang đăng nhập bằng cách sử dụng component.

Sau đó, chúng ta sử dụng PrivateRoute để bọc các tuyến đường cần được bảo vệ, chẳng hạn như /dashboard và /settings. Khi người dùng truy cập vào các tuyến đường này, PrivateRoute sẽ kiểm tra trạng thái đăng nhập và chỉ hiển thị nội dung nếu người dùng đã đăng nhập.

Bằng cách này, chúng ta có thể dễ dàng bảo vệ các tuyến đường quan trọng trong ứng dụng của mình.

29. Handling the 404 Case (Unknown Routes)

Khi người dùng truy cập vào một tuyến đường không tồn tại trong ứng dụng, chúng ta cần cung cấp một cách xử lý thích hợp. Điều này thường được gọi là "trường hợp 404" và là một phần quan trọng trong việc cung cấp trải nghiệm người dùng tốt.

Trong React Router DOM v6, chúng ta có thể sử dụng thành phần Routes và Route để xử lý trường hợp 404. Đây là một ví dụ về cách triển khai:

```
import { Routes, Route, Navigate } from "react-router-dom";
import HomePage from "./components/HomePage";
import AboutPage from "./components/AboutPage";
import NotFoundPage from "./components/NotFoundPage";

const App = () => {
  return (
    <Routes>
      <Route path="/" element={<HomePage />} />
      <Route path="/about" element={<AboutPage />} />
      {/* Tuyến đường 404 */}
      <Route path="*" element={<NotFoundPage />} />
    </Routes>
  );
};
```

Trong ví dụ trên, chúng ta định nghĩa ba tuyến đường: /, /about và _. Tuyến đường _ là một tuyến đường "catch-all", được sử dụng để xử lý tất cả các tuyến đường không xác định khác.

Khi người dùng truy cập vào một tuyến đường không tồn tại, React Router DOM sẽ khớp với tuyến đường * và hiển thị thành phần . Đây là nơi chúng ta có thể tạo ra một trang 404 tùy chỉnh, cung cấp một thông báo lỗi hoặc chuyển hướng người dùng trở lại trang chủ.

Ngoài ra, chúng ta cũng có thể sử dụng component để chuyển hướng người dùng đến một tuyến đường khác khi họ truy cập vào một tuyến đường không tồn tại:

```
<Route path="*" element={<Navigate to="/" replace />} />
```

Trong ví dụ này, thay vì hiển thị một trang 404 tùy chỉnh, chúng ta sẽ chuyển hướng người dùng trở lại trang chủ khi họ truy cập vào một tuyến đường không tồn tại.

Xử lý trường hợp 404 một cách thích hợp là rất quan trọng để cung cấp một trải nghiệm người dùng tốt và chuyên nghiệp cho ứng dụng của bạn.

30. Loading Routes Lazily

Khi ứng dụng của bạn có nhiều tuyến đường, việc tải tất cả các tuyến đường cùng một lúc có thể ảnh hưởng đến hiệu suất. Để cải thiện hiệu suất, chúng ta có thể sử dụng tính năng "Tải Routes Lười Biếng" trong React Router DOM v6.

Tải Routes Lười Biếng cho phép chúng ta tải các tuyến đường chỉ khi người dùng thực sự cần đến chúng. Điều này giúp giảm tải ban đầu của ứng dụng và cải thiện thời gian tải.

Để triển khai Tải Routes Lười Biếng, chúng ta sử dụng React.lazy() và component. Đây là một ví dụ:

```
import { Routes, Route, Suspense } from "react-router-dom";

const HomePage = React.lazy(() => import("./components/HomePage"));
const AboutPage = React.lazy(() => import("./components/AboutPage"));
const SettingsPage = React.lazy(() => import("./components/SettingsPage"));

const App = () => {
  return (
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<HomePage />} />
        <Route path="/about" element={<AboutPage />} />
        <Route path="/settings" element={<SettingsPage />} />
      </Routes>
    </Suspense>
  );
};
```

Trong ví dụ này, chúng ta sử dụng React.lazy() để tạo các thành phần tuyến đường động. Thay vì tải tất cả các thành phần cùng một lúc, chúng ta chỉ tải chúng khi người dùng thực sự cần đến.

Chúng ta cũng sử dụng component để xử lý trường hợp khi các thành phần chưa được tải xong. Trong trường hợp này, chúng ta hiển thị một thông báo "Loading..." trong khi chờ đợi các thành phần được tải.

Bằng cách này, ứng dụng của bạn sẽ tải nhanh hơn và hiệu suất sẽ được cải thiện, đặc biệt là khi ứng dụng có nhiều tuyến đường.

31. Understanding Fetch

Fetch API là một giao diện JavaScript hiện đại dùng để thực hiện các yêu cầu HTTP/HTTPS và xử lý các phản hồi tương ứng. Nó được giới thiệu lần đầu tiên vào năm 2015 và dần trở thành một phần tiêu chuẩn của các trình duyệt hiện đại. Fetch API cung cấp một giao diện rất linh hoạt và mạnh mẽ để thực hiện các yêu cầu mạng, với nhiều tùy chọn để tùy chỉnh các yêu cầu và xử lý các phản hồi.

Một trong những ưu điểm chính của Fetch API so với các phương pháp truyền thống như XMLHttpRequest là tính đơn giản và dễ sử dụng. Fetch API sử dụng Promises, cho phép lập trình viên xử lý các kết quả bất đồng bộ một cách dễ dàng hơn. Nó cũng cung cấp nhiều tính năng mới như hỗ trợ streaming, khả năng theo dõi tiến trình, và xử lý lỗi tốt hơn.

32. Understand HTTP Protocols

Trước khi tìm hiểu sâu hơn về cách sử dụng Fetch API, chúng ta cần có một sự hiểu biết cơ bản về các giao thức HTTP (Hypertext Transfer Protocol). HTTP là giao thức chính được sử dụng để truyền tải dữ liệu trên web, bao gồm các yêu cầu từ trình duyệt đến máy chủ web và các phản hồi từ máy chủ web trở lại trình duyệt.

Các phương thức HTTP phổ biến bao gồm:

- **GET**: Dùng để yêu cầu dữ liệu từ một nguồn cụ thể.
- **POST**: Dùng để gửi dữ liệu mới lên máy chủ.
- **PUT**: Dùng để cập nhật dữ liệu hiện có trên máy chủ.
- **DELETE**: Dùng để xóa dữ liệu khỏi máy chủ.

Ngoài ra, các mã trạng thái HTTP cũng rất quan trọng để hiểu và xử lý các phản hồi từ máy chủ, chẳng hạn như:

- **200 OK**: Yêu cầu thành công.
- **404 Not Found**: Tài nguyên không tồn tại.
- **500 Internal Server Error**: Lỗi từ phía máy chủ.

Hiểu rõ các giao thức HTTP cơ bản sẽ giúp bạn sử dụng Fetch API một cách hiệu quả hơn.

33. RESTful API

RESTful API (Representational State Transfer API) là một kiến trúc phổ biến để xây dựng các API, dựa trên các nguyên tắc của giao thức HTTP. Trong một RESTful API, các tài nguyên (resources) được định danh bằng các URL duy nhất, và các phương thức HTTP được sử dụng để thực hiện các hành động CRUD (Create, Read, Update, Delete) trên các tài nguyên này.

Ví dụ, để lấy danh sách người dùng từ một RESTful API, bạn có thể sử dụng phương thức GET với URL **/users**. Để tạo một người dùng mới, bạn có thể sử dụng phương thức POST với URL **/users**. Các phương thức PUT và DELETE tương ứng được sử dụng để cập nhật và xóa người dùng.

Việc tuân thủ các nguyên tắc RESTful giúp API trở nên dễ hiểu, dễ sử dụng và tái sử dụng hơn.

34. Using fetch to make a Request API

Bây giờ chúng ta đã hiểu được cơ bản về Fetch API và các giao thức HTTP, hãy cùng xem cách sử dụng Fetch để thực hiện các yêu cầu API.

Cú pháp cơ bản của Fetch API như sau:

```
fetch(url, options)
  .then((response) => response.json())
  .then((data) => console.log(data))
  .catch((error) => console.error(error));
```

Trong đó:

- **url**: Là đường dẫn URL của API mà bạn muốn gọi.
- **options**: Là một đối tượng tùy chọn cho phép bạn cấu hình các chi tiết của yêu cầu, như phương thức HTTP, tiêu đề, dữ liệu gửi đi, v.v.

Ví dụ, để lấy danh sách người dùng từ một RESTful API, bạn có thể sử dụng mã sau:

```
fetch("/users")
  .then((response) => response.json())
  .then((data) => {
    console.log(data);
    // Xử lý dữ liệu người dùng ở đây
  })
  .catch((error) => console.error(error));
```

Trong ví dụ này, chúng ta sử dụng phương thức GET mặc định để yêu cầu dữ liệu từ /users. Fetch API trả về một Promise, cho phép chúng ta sử dụng các phương thức .then() và .catch() để xử lý kết quả.

Nếu bạn muốn gửi dữ liệu lên máy chủ, ví dụ như tạo một người dùng mới, bạn có thể sử dụng phương thức POST và thêm dữ liệu vào options:

```
const newUser = { name: "John Doe", email: "john@example.com" };

fetch("/users", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
  },
  body: JSON.stringify(newUser),
})
  .then((response) => response.json())
  .then((data) => {
    console.log(data);
    // Xử lý dữ liệu người dùng mới được tạo ở đây
  })
  .catch((error) => console.error(error));
```

Trong ví dụ này, chúng ta đã thêm method: 'POST' vào options để chỉ định sử dụng phương thức POST. Chúng ta cũng đã thêm tiêu đề Content-Type để xác định định dạng dữ liệu gửi đi (JSON), và chuyển đổi đối tượng newUser thành chuỗi JSON trước khi gửi vào body.

Fetch API cung cấp nhiều tùy chọn khác nhau để tùy chỉnh yêu cầu, như thêm tiêu đề, xác thực, theo dõi tiến trình, v.v. Bạn có thể tham khảo tài liệu chính thức của Fetch API để tìm hiểu thêm về các tính năng nâng cao.

35. Load data from API in React

Khi làm việc với React, Fetch API trở nên đặc biệt hữu ích để tải dữ liệu từ các API và cập nhật giao diện người dùng tương ứng. Dưới đây là một ví dụ về cách sử dụng Fetch API trong một React component:

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

function UserList() {
  const [users, setUsers] = useState([]);

  useEffect(() => {
    fetch("/users")
      .then((response) => response.json())
      .then((data) => setUsers(data))
      .catch((error) => console.error(error));
  }, []);

  return (
    <div>
      <h1>User List</h1>
      <ul>
        {users.map((user) => (
          <li key={user.id}>{user.name}</li>
        ))}
      </ul>
    </div>
  );
}

export default UserList;
```

Trong ví dụ này, chúng ta sử dụng useState để lưu trữ danh sách người dùng, và useEffect để gọi API /users khi component được render lần đầu tiên. Khi dữ liệu được nhận về, chúng ta cập nhật state users và render danh sách người dùng trong giao diện.

Lưu ý rằng, trong ví dụ này chúng ta chỉ gọi API một lần khi component được mount. Trong thực tế, bạn có thể muốn thêm các logic để tải lại dữ liệu khi cần thiết, ví dụ khi người dùng tương tác với component.

36. Optimize Request API

Khi làm việc với các API, việc tối ưu hóa các yêu cầu là rất quan trọng để cải thiện hiệu suất và trải nghiệm người dùng. Dưới đây là một số kỹ thuật để tối ưu hóa yêu cầu API:

- **Caching**: Lưu trữ các phản hồi API vào bộ nhớ cache để tránh phải gọi API liên tục khi dữ liệu không thay đổi. Bạn có thể sử dụng các thư viện như axios-cache-adapter hoặc tự xây dựng cơ chế cache riêng.
- **Lazy Loading**: Chỉ tải dữ liệu khi thực sự cần thiết, thay vì tải tất cả dữ liệu ngay từ đầu. Ví dụ, bạn có thể tải dữ liệu phân trang hoặc tải dữ liệu khi người dùng cuộn đến cuối trang.
- **Batch Requests**: Thay vì gửi nhiều yêu cầu API riêng lẻ, hãy nhóm chúng lại thành một yêu cầu duy nhất. Điều này giúp giảm số lượng kết nối và tải trang.
- **Sử Dụng Abortable Fetch**: Fetch API cung cấp một cách để hủy bỏ các yêu cầu không cần thiết, ví dụ khi người dùng chuyển sang trang khác. Điều này giúp giảm tải cho máy chủ và cải thiện trải nghiệm người dùng.
- **Sử Dụng HTTP/2**: HTTP/2 cung cấp nhiều cải tiến so với HTTP/1.1, như hỗ trợ đa luồng, nén tiêu đề, và push server. Sử dụng HTTP/2 có thể giúp cải thiện hiệu suất của các yêu cầu API.
- **Theo Dõi và Xử Lý Lỗi**: Luôn theo dõi các lỗi xảy ra trong quá trình gọi API và xử lý chúng một cách thích hợp. Điều này giúp cải thiện trải nghiệm người dùng và hỗ trợ việc gỡ lỗi.

Bằng cách áp dụng các kỹ thuật tối ưu hóa này, bạn có thể cải thiện hiệu suất và trải nghiệm người dùng khi làm việc với các API.

37. Global State management

Quản lý trạng thái toàn cục (global state management) là một vấn đề phổ biến trong phát triển ứng dụng React. Khi ứng dụng của bạn phát triển, việc quản lý trạng thái trở nên ngày càng phức tạp, đặc biệt khi các thành phần được chia sẻ giữa nhiều thành phần khác nhau. Việc chia sẻ và đồng bộ hóa trạng thái giữa các thành phần có thể trở nên khó khăn và dễ dẫn đến lỗi.

38. Introduce React Signify

React Signify là một thư viện đơn giản cung cấp các tính năng quản lý và cập nhật global state một cách hiệu quả. Nó đặc biệt hữu ích trong các ứng dụng React để quản lý state và tự đồng bộ khi giá trị của chúng thay đổi.

Ưu điểm của thư viện:

- Thư viện nhỏ gọn
- Cú pháp đơn giản
- Hỗ trợ kiểm soát re-render hiệu quả

Thông tin thư viện:

- **Trang chủ** : https://reactsignify.dev
- **Git** : https://github.com/VietCPQ94/react-signify
- **NPM** : https://www.npmjs.com/package/react-signify

Cài đặt:

```
<div style="page-break-after: always;"></div>
# NPM
npm install react-signify

<div style="page-break-after: always;"></div>
# Yarn
yarn add react-signify
```

39. Manage Global State

Khởi tạo

Bạn có thể quản lý global state bằng cách khởi tạo Signify ở bất kỳ file nào, hãy tham khảo ví dụ sau

```
import { signify } from "react-signify";

const sCount = signify(0);
```

Tại đây ta tạo 1 biến sCount với giá trị khởi tạo là 0.

Sử dụng tại nhiều nơi

Việc sử dụng đơn giản bằng công cụ export/import của module.

Component A (export Signify)

```
import { signify } from "react-signify";

export const sCount = signify(0);

export default function ComponentA() {
  return (
    <div>
      <h1>{sCount.html}</h1>
      <button onClick={() => sCount.set((pre) => (pre.value += 1))}>UP</button>
    </div>
  );
}
```

Component B (import Signify)

```
import { sCount } from "./ComponentA";

export default function ComponentB() {
  return (
    <div>
      <h1>{sCount.html}</h1>
      <button onClick={() => sCount.set((pre) => (pre.value += 1))}>UP</button>
    </div>
  );
}
```

Từ đây ta thấy được tính linh động của Signify, khai báo đơn giản, sử dụng ở mọi nơi.

Các tính năng cơ bản

Hiển thị trên giao diện

Ta sẽ sử dụng thuộc tính **html** để hiện thị giá trị lên giao diện.

```
import { signify } from "react-signify";

const sCount = signify(0);

export default function App() {
  return (
    <div>
      <h1>{sCount.html}</h1>
    </div>
  );
}
```

Cập nhật giá trị

React Signify cung cấp tính năng set, cho phép chúng ta thay đổi giá trị một cách hiệu quả

```
import { signify } from "react-signify";

const sCount = signify(0);

export default function App() {
  return (
    <div>
      <h1>{sCount.html}</h1>
      <button onClick={() => sCount.set(1)}>Set 1</button>
      <button onClick={() => sCount.set((pre) => (pre.value += 1))}>
        UP 1
      </button>
    </div>
  );
}
```

Việc nhấn nút sẽ thay đổi giá trị của Signify và được cập nhật tự động lên giao diện.

Hook use

Tính năng cho phép lấy giá trị của Signify và sử dụng nó như một state của component.

```
import { useEffect } from "react";
import { signify } from "react-signify";

const sCount = signify(0);

export default function App() {
  const countValue = sCount.use();

  useEffect(() => {
    console.log(countValue);
  }, [countValue]);

  return (
    <div>
      <h1>{countValue}</h1>
      <button onClick={() => sCount.set((pre) => (pre.value += 1))}>UP</button>
    </div>
  );
}
```

40. Optimize re-render

watch

Tính năng cho phép theo dõi sự thay đổi giá trị của Signify một cách an toàn.

```
import { signify } from "react-signify";

const sCount = signify(0);

export default function App() {
  sCount.watch((newValue) => {
    console.log(newValue);
  }, []);
  return (
    <div>
      <button onClick={() => sCount.set((pre) => (pre.value += 1))}>UP</button>
    </div>
  );
}
```

Wrap

Tính năng áp dụng giá trị của Signify trong một vùng giao diện cụ thể.

```
import { signify } from "react-signify";

const sCount = signify(0);

export default function App() {
  return (
    <div>
      <sCount.Wrap>
        {(value) => (
          <div>
            <h1>{value}</h1>
          </div>
        )}
      </sCount.Wrap>
      <button onClick={() => sCount.set((pre) => (pre.value += 1))}>UP</button>
    </div>
  );
}
```

Hardwrap

Tính năng áp dụng giá trị của Signify trong một vùng giao diện và hạn chế được các re-render không cần thiết khi component cha re-render.

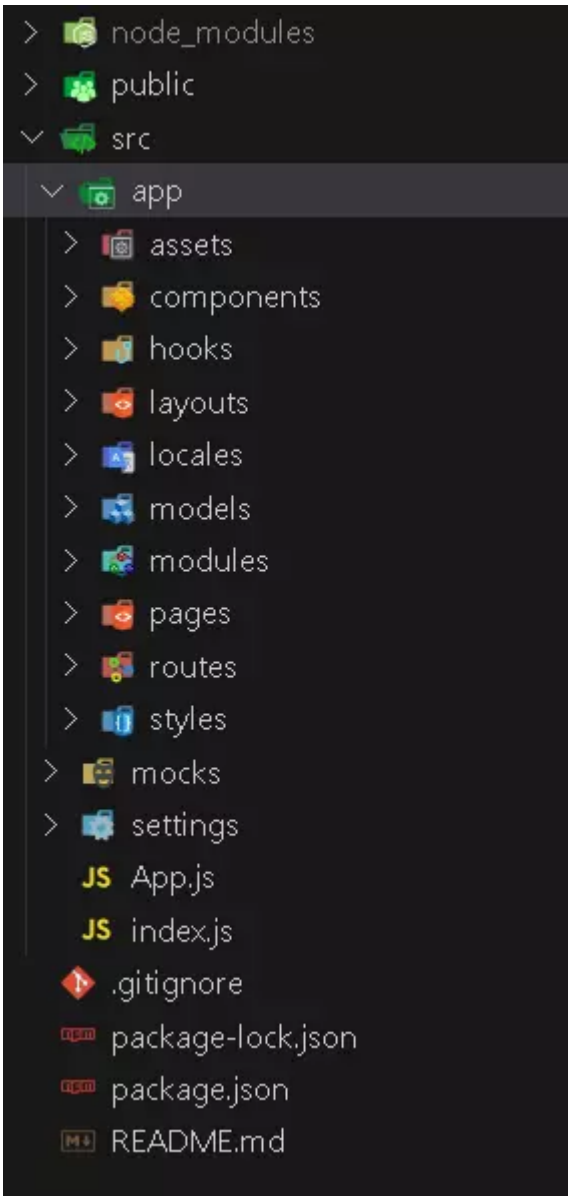
```
import { signify } from "react-signify";

const sCount = signify(0);

export default function App() {
  return (
    <div>
      <sCount.HardWrap>
        {(value) => (
          <div>
            <h1>{value}</h1>
          </div>
        )}
      </sCount.HardWrap>
      <button onClick={() => sCount.set((pre) => (pre += 1))}>UP</button>
    </div>
  );
}
```

39. Folder structure

Tiếp theo, hãy xây dựng cấu trúc folder theo mẫu sau



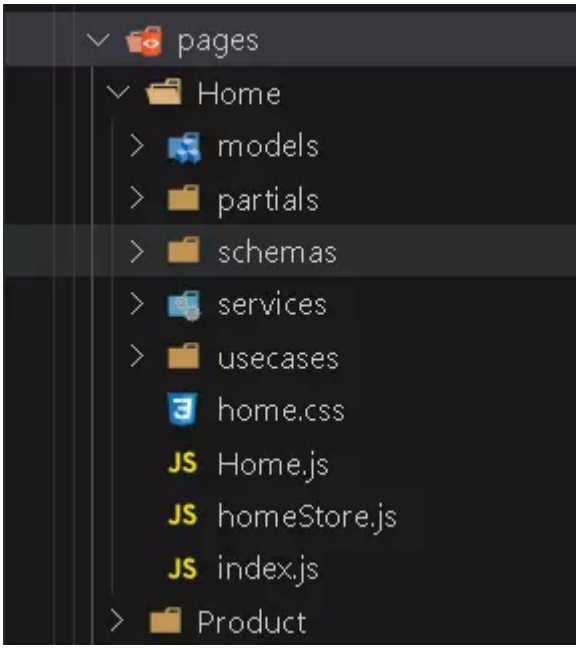
Hãy đi sâu và giải thích tính năng của từng folder nhé:

- **node_modules** : folder chứa thư viện dự án
- **public** : folder mô tả thông tin static của dự án như : index.html, favicon, v.v...
- **src** : folder chưa cấu trúc chính của dự án
    - **app** : folder chứa source phát triển
        - **assets** : folder chưa các tài nguyên dự án như image, txt, v.v...
        - **components** : folder chứa các component common xài chung cho dự án như : button, input, v.v...
        - **hooks** : folder chứa các custom hook xài chung cho dự án
        - **layouts** : folder chứa các component layout tĩnh như header, footer, empty page, 404 page, v.v...
        - **locales** : folder chưa các file ngôn ngữ xài cho dự án như : en.json, vn.json, v.v...
        - **modules** : folder chứa thư viện tự xây dựng để xài chung cho dự án
        - **pages** : folder chứa mã nguồn các màn hình
        - **routes** : folder chứa thông tin router, guard của dự án
        - **styles** : folder chứa source css common của dự án
    - **mocks** : folder thiết lập data mock cho dự án
    - **settings** : folder chứa setting dự án, thông tin môi trường, cấu hình hosting
    - **App.js** : file component đại diện cho dự án
    - **index.js** : file root khởi chạy của dự án
- **.gitignore** : file cấu hình loại bỏ các file/folder không cần thiết trên GIT
- **package-lock.json** : file liệt kê nơi tải thư viện
- **package.json** : file mô tả thông tin dự án như tên, version, thư viện, v.v...

Tuyệt vời, vậy là bạn đã bước đầu thành công trong việc phân tích cấu trúc chính của một dự án rồi đấy, hãy tiếp tục session tiếp theo nhé.

40. Project workflow

Sau khi đã xây dựng xong các file và folder hệ thống, ta sẽ đi đến bước tiếp theo để xây dựng cấu trúc file và folder của từng màn hình. Hãy tham khảo cấu trúc sau



Đây là hệ thống file và folder được triển khai trong folder **pages**, sau đây là giải thích chi tiết :

- **pages** : folder chứa mã nguồn các màn hình
    - **Home** : folder màn hình Home
        - **models** : folder chứa các đối tượng trên màn hình
        - **partials** : folder chứa các mảnh giao diện trên màn hình (spliting source)
        - **schemas** : folder chứa source validate form hoặc các điều kiện xảy ra trên màn hình
        - **services** : folder chứa source thực hiện tính năng Side Effect như : request API, tracking time, service worker, v.v...
        - **usecases** : folder chứa các file xử lý dữ liệu hoặc nghiệp phức tạp trên màn hình
        - **home.css** : file style của màn hình
        - **Home.js** : file Component đại diện cho màn hình
        - **homeStore.js** : file store chứa các biến global state đại diện hoặc được sử dụng cho màn hình
        - **index.js** : file export của màn hình, nhằm giảm lược import path
    - **[Folder màn hình khác]** : folder có tên được đại diện bằng mã hoặc tên màn hình

Trên mô hình màn hình trên, ta sẽ vận hành dự án theo các nguyên tắc như sau:

- Màn hình sẽ có hệ thống tổ chức folder riêng biệt.
- Hạn chế sử dụng source code bên ngoài folder màn hình khi không thật sự cần thiết
- Nếu nhận thấy source của màn hình có thể được ứng dụng tại màn hình khác, hãy cân nhắc dời source ra ngoài phạm vi chung của dự án để tái sử dụng hiệu quả.

41. Build React Product

Khi bạn đã sẵn sàng để xây dựng ứng dụng React của mình, hãy chạy lệnh sau:

```
npm run build
```

Lệnh này sẽ tạo ra một thư mục "build" trong dự án của bạn, chứa các tệp đã được tối ưu hóa và sẵn sàng để triển khai.

Trong thư mục "build", bạn sẽ tìm thấy các tệp sau:

- **index.html**: Đây là tệp HTML chính, nơi ứng dụng React của bạn sẽ được nhúng.
- **static/js/main.[hash].js**: Đây là tệp JavaScript chính của ứng dụng, chứa mã nguồn đã được gom và tối ưu hóa.
- **static/css/main.[hash].js**: Đây là tệp CSS chính của ứng dụng, chứa các kiểu đã được gom và tối ưu hóa.
- **asset-manifest.json**: Tệp này chứa thông tin về các tệp tài nguyên được sử dụng trong ứng dụng.
- **manifest.json**: Tệp này chứa thông tin về ứng dụng web, chẳng hạn như tên, biểu tượng và màu chủ đề.
- **robots.txt**: Tệp này chỉ định các quy tắc cho các robot tìm kiếm, chẳng hạn như Google hoặc Bing.

Các tệp này đã được tối ưu hóa để tăng tốc độ tải và cải thiện hiệu suất của ứng dụng. Ví dụ, mã nguồn JavaScript và CSS đã được gom lại và nén để giảm kích thước tệp. Ngoài ra, các tệp cũng được đánh dấu với các mã băm để ngăn chặn bộ đệm trình duyệt.

43. useRef

useRef là một Hook trong React cho phép bạn tạo ra các biến mutable (có thể thay đổi giá trị) mà không gây ảnh hưởng đến quá trình render của component. Điều này rất hữu ích khi bạn cần lưu trữ một giá trị mà không muốn nó gây ra một lần render mới.

Cách sử dụng useRef khá đơn giản:

```
const refVariable = useRef(initialValue);
```

Ở đây, refVariable là một đối tượng có một thuộc tính current mà bạn có thể truy cập và thay đổi giá trị của nó.

Một số trường hợp sử dụng phổ biến của useRef bao gồm:

- **Lưu trữ các giá trị mutable**: Bạn có thể sử dụng useRef để lưu trữ các giá trị mà không cần trigger một lần render mới, chẳng hạn như một bộ đếm, một timer hoặc một reference đến một DOM element.
- **Truy cập trực tiếp vào DOM elements**: Bằng cách gán reference của một DOM element vào một ref, bạn có thể truy cập và thao tác trực tiếp với nó mà không cần sử dụng các phương thức như querySelector hay getElementById.
- **Lưu trữ các giá trị giữa các lần render**: Khi một component được re-render, React sẽ giữ nguyên giá trị của ref và không reset nó về giá trị ban đầu. Điều này rất hữu ích khi bạn cần lưu trữ các giá trị giữa các lần render.

Ví dụ, hãy xem cách sử dụng useRef để lưu trữ một bộ đếm:

```jsx
import { useState, useRef } from "react";

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

  const increment = () => {
    setCount(count + 1);
    countRef.current++;
  };

  const logCount = () => {
    console.log(`Count from state: ${count}`);
    console.log(`Count from ref: ${countRef.current}`);
  };

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={logCount}>Log Counts</button>
    </div>
  );
}
```

Trong ví dụ trên, chúng ta sử dụng useState để quản lý trạng thái count và useRef để lưu trữ một biến đếm riêng biệt. Khi người dùng nhấn nút "Increment", chúng ta cập nhật cả count (thông qua useState) và countRef.current (thông qua useRef). Khi nhấn nút "Log Counts", chúng ta sẽ in ra cả giá trị từ state và giá trị từ ref.

Lưu ý rằng, khi bạn cập nhật giá trị của ref, nó không sẽ không gây ra một lần render mới như khi bạn cập nhật state bằng useState. Đây là một ưu điểm chính của useRef - nó cho phép bạn lưu trữ các giá trị mutable mà không ảnh hưởng đến quá trình render của component.

43. useMemo

useMemo là một Hook trong React cho phép bạn memoize (lưu vào bộ nhớ) kết quả của một function, giúp tối ưu hóa hiệu suất của component bằng cách tránh thực hiện lại những tính toán không cần thiết.

Cách sử dụng useMemo như sau:

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

Ở đây, computeExpensiveValue là một function thực hiện một tính toán tốn kém. useMemo sẽ gọi function này và lưu kết quả vào memoizedValue. Nếu các giá trị a và b không thay đổi giữa các lần render, useMemo sẽ không gọi lại computeExpensiveValue mà thay vào đó sẽ trả về giá trị đã được lưu.

Một số trường hợp sử dụng phổ biến của useMemo bao gồm:

- **Tránh thực hiện lại các tính toán tốn kém**: Khi bạn có một function thực hiện một tính toán phức tạp, bạn có thể sử dụng useMemo để lưu vào bộ nhớ kết quả của function này và tránh thực hiện lại nó nếu các tham số đầu vào không thay đổi.
- **Tối ưu hóa hiệu suất của component**: Bằng cách sử dụng useMemo, bạn có thể tránh thực hiện lại những tính toán không cần thiết, giúp cải thiện hiệu suất của component, đặc biệt là trong các trường hợp có nhiều tính toán phức tạp.
- **Tránh tạo ra các đối tượng mới không cần thiết**: Khi bạn cần tạo ra một đối tượng phức tạp, bạn có thể sử dụng useMemo để lưu vào bộ nhớ đối tượng đó và tránh tạo ra các đối tượng mới không cần thiết.

Ví dụ, hãy xem cách sử dụng useMemo để tối ưu hóa hiệu suất của một component thực hiện một tính toán tốn kém:

```jsx
import { useState, useMemo } from "react";

function ExpensiveCalculation() {
  const [a, setA] = useState(0);
  const [b, setB] = useState(0);

  const result = useMemo(() => {
    console.log("Performing expensive calculation...");
    return a * b;
  }, [a, b]);

  return (
    <div>
      <input
        type="number"
        value={a}
        onChange={(e) => setA(parseInt(e.target.value))}
      />
      <input
        type="number"
        value={b}
        onChange={(e) => setB(parseInt(e.target.value))}
      />
      <p>Result: {result}</p>
    </div>
  );
}
```

Trong ví dụ trên, chúng ta sử dụng useMemo để lưu vào bộ nhớ kết quả của phép tính a * b. Khi người dùng thay đổi giá trị của a hoặc b, useMemo sẽ chỉ gọi lại function tính toán nếu một trong các giá trị đó thay đổi. Điều này giúp tránh thực hiện lại tính toán không cần thiết, cải thiện hiệu suất của component.

Lưu ý rằng, useMemo không chỉ hữu ích khi thực hiện các tính toán tốn kém. Nó cũng có thể được sử dụng để tránh tạo ra các đối tượng mới không cần thiết, chẳng hạn như các hàm callback hoặc các đối tượng phức tạp.

44. useCallback

useCallback là một Hook trong React cho phép bạn memoize (lưu vào bộ nhớ) một function, giúp tối ưu hóa hiệu suất của component bằng cách tránh tạo ra các function mới không cần thiết.

Cách sử dụng useCallback như sau:

```
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

Ở đây, doSomething là một function mà bạn muốn memoize. useCallback sẽ trả về một reference đến function này, và chỉ tạo ra một function mới khi một trong các giá trị a hoặc b thay đổi.

Một số trường hợp sử dụng phổ biến của useCallback bao gồm:

- **Tối ưu hóa hiệu suất của component**: Khi bạn truyền một function làm prop cho một component con, bạn có thể sử dụng useCallback để tránh tạo ra một function mới không cần thiết trong mỗi lần render, giúp cải thiện hiệu suất của component.
- **Tránh re-render không cần thiết trong component con**: Khi một function được truyền làm prop cho một component con, nếu function đó được tạo lại trong mỗi lần render, component con có thể sẽ re-render không cần thiết. Bằng cách sử dụng useCallback, bạn có thể tránh được vấn đề này.
- **Tối ưu hóa hiệu suất của các hàm callback**: Khi bạn cần sử dụng các hàm callback, chẳng hạn như onClick hoặc onChange, bạn có thể sử dụng useCallback để lưu vào bộ nhớ các hàm callback này và tránh tạo ra các hàm mới không cần thiết.

Ví dụ, hãy xem cách sử dụng useCallback để tối ưu hóa hiệu suất của một component con:

```
import { useState, useCallback } from "react";

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

  const handleIncrement = useCallback(() => {
    setCount((prevCount) => prevCount + 1);
  }, []);

  return (
    <div>
      <ChildComponent onIncrement={handleIncrement} />
      <p>Count: {count}</p>
    </div>
  );
}

function ChildComponent({ onIncrement }) {
  console.log("ChildComponent rendered");

  return <button onClick={onIncrement}>Increment</button>;
}
```

Trong ví dụ trên, chúng ta sử dụng useCallback để memoize hàm handleIncrement và truyền nó làm prop cho component con ChildComponent. Điều này giúp tránh tạo ra một function mới trong mỗi lần render của ParentComponent, từ đó giúp tránh re-render không cần thiết của ChildComponent.

Lưu ý rằng, useCallback và useMemo có một số điểm tương đồng, nhưng chúng có mục đích sử dụng khác nhau. useMemo được sử dụng để memoize kết quả của một function, trong khi useCallback được sử dụng để memoize chính function đó. Bạn nên lựa chọn sử dụng useCallback khi bạn muốn tránh tạo ra các function mới không cần thiết, và sử dụng useMemo khi bạn muốn tránh thực hiện lại các tính toán tốn kém.