Chapter 02: Xử lý Sự kiện và Hiển thị Điều kiện trong ReactJS

A. Mục tiêu

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

- Thành thạo cách xử lý các sự kiện người dùng như click chuột và thay đổi dữ liệu nhập.
- Hiểu cách truyền dữ liệu từ component con lên component cha.
- Áp dụng kỹ thuật hiển thị điều kiện để kiểm soát giao diện người dùng.
- Hiển thị danh sách dữ liệu hiệu quả với việc quản lý `key` đúng cách.

B. Nội dung lý thuyết

1. Xử lý Sự kiện

Cú pháp sự kiện trong React

React sử dụng tên sự kiện theo chuẩn camelCase (ví dụ: `onClick`, `onChange`, `onSubmit`) để gắn hàm xử lý sự kiện vào các phần tử JSX. Hàm xử lý sự kiện là các hàm được gọi khi sự kiện xảy ra.

Ví dụ minh họa:

```
function Button() {
  function handleClick() {
    console.log('Nút đã được nhấn!');
  }
  return <button onClick={handleClick}>Nhấn tôi</button>;
}
```

Giải thích:

- `handleClick` là một hàm xử lý sự kiện, được định nghĩa trong component.
- Thuộc tính `onClick` nhận một tham chiếu đến hàm `handleClick` (không phải gọi hàm trực tiếp `handleClick()`).
- Khi người dùng nhấn nút, hàm `handleClick` sẽ được gọi và in thông báo ra console.

Hàm xử lý sự kiện

- Hàm xử lý thường được định nghĩa trong component.
- Để truyền tham số vào hàm xử lý, sử dụng arrow function hoặc phương thức `bind`:

```
function handleClick(id) {
  console.log(`Đã nhấn vào mục ${id}`);
}
<button onClick={() => handleClick(1)}>Nhấn</button>;
```

- **Lưu ý:** Gọi trực tiếp `handleClick(1)` trong `onClick` sẽ khiến hàm chạy ngay lập tức khi component được render, thay vì chờ sự kiện.

Giải thích thêm: Sử dụng arrow function (`() => handleClick(id)`) đảm bảo hàm chỉ được gọi khi sự kiện xảy ra. Điều này đặc biệt quan trọng khi bạn cần truyền tham số vào hàm xử lý.

Đối tượng sự kiện (Event Object)

React cung cấp đối tượng `event` trong hàm xử lý, chứa thông tin về sự kiện. Một số thuộc tính phổ biến:

- `event.target.value`: Lấy giá trị của trường input.
- `event.preventDefault()`: Ngăn hành vi mặc định của trình duyệt (ví dụ: ngăn form gửi dữ liệu).

Ví dụ:

```
function Input() {
  function handleChange(event) {
    console.log(event.target.value);
  }
  return <input type="text" onChange={handleChange} placeholder="Nhập vào đâ
}
```

Giải thích:

- Khi người dùng nhập vào ô input, sự kiện `onChange` kích hoạt và hàm `handleChange` được gọi.
- `event.target.value` trả về giá trị hiện tại của ô input, được in ra console.

Lưu ý về Class Components

Trong class components, bạn cần `bind` hàm xử lý vào `this` trong constructor để đảm bảo ngữ cảnh đúng:

```
class Button extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Nút đã được nhấn!');
  }
  render() {
    return <button onClick={this.handleClick}>Nhấn tôi</button>;
  }
}
```

Giải thích thêm: Functional components với hooks không cần `bind`, giúp mã đơn giản hơn. Đây là lý do functional components được khuyến khích sử dụng.

---

2. Nâng cấp trạng thái (Lifting State Up)

Khi nhiều component cần chia sẻ và cập nhật cùng một trạng thái, trạng thái nên được quản lý bởi component cha chung gần nhất. Kỹ thuật này gọi là "nâng cấp trạng thái" (lifting state up).

Quy trình:

- Định nghĩa trạng thái và hàm xử lý trong component cha.
- Truyền hàm xử lý xuống component con qua props.
- Component con gọi hàm xử lý để cập nhật trạng thái của cha.

Ví dụ:

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

function App() {
  const [value, setValue] = useState('');
  function handleChange(newValue) {
    setValue(newValue);
  }
  return <Child onChange={handleChange} value={value} />;
}

function Child({ onChange, value }) {
  return <input value={value} onChange={(e) => onChange(e.target.value)} />;
}
```

Giải thích:

- `App` quản lý trạng thái `value` và hàm `handleChange`.
- `Child` nhận `value` và `onChange` qua props, sử dụng chúng để hiển thị và cập nhật giá trị input.
- Khi người dùng nhập, `Child` gọi `onChange` để cập nhật trạng thái trong `App`.

Giải thích thêm: Kỹ thuật này đảm bảo trạng thái được đồng bộ giữa các component, đặc biệt hữu ích trong các ứng dụng có nhiều component tương tác với cùng một dữ liệu.

---

3. Hiển thị Điều kiện (Conditional Rendering)

Hiển thị điều kiện cho phép kiểm soát giao diện dựa trên trạng thái hoặc props. Có ba cách phổ biến:

Sử dụng câu lệnh `if`

Dùng `if` ngoài JSX để trả về các component khác nhau:

```
function Greeting({ isLoggedIn }) {
  if (isLoggedIn) {
    return <h1>Chào mừng bạn trở lại!</h1>;
  }
  return <h1>Vui lòng đăng nhập.</h1>;
}
```

Giải thích: Dựa trên giá trị của `isLoggedIn`, component trả về JSX tương ứng. Cách này phù hợp với logic phức tạp.

Toán tử ba ngôi (Ternary Operator)

Sử dụng toán tử `condition ? expr1 : expr2` trong JSX để hiển thị điều kiện ngắn gọn:

```
function Greeting({ isLoggedIn }) {
  return (
    <h1>{isLoggedIn ? 'Chào mừng bạn trở lại!' : 'Vui lòng đăng nhập.'}</h1>
  );
}
```

Giải thích: Toán tử ba ngôi phù hợp khi cần hiển thị điều kiện trực tiếp trong JSX, giúp mã gọn gàng hơn.

Toán tử `&&`

Sử dụng `&&` để chỉ hiển thị một phần tử khi điều kiện đúng:

```
function Notification({ hasUnread }) {
  return (
    <div>
      {hasUnread && <p>Bạn có tin nhắn chưa đọc!</p>}
    </div>
  );
}
```

Giải thích: Nếu `hasUnread` là `true`, đoạn `<p>` sẽ được hiển thị; nếu `false`, không có gì được hiển thị.

So sánh các phương pháp

- `if`: Tốt cho logic phức tạp hoặc trả về nhiều JSX khác nhau.
- **Toán tử ba ngôi**: Phù hợp cho các điều kiện đơn giản trong JSX.
- `&&`: Lý tưởng khi chỉ cần hiển thị hoặc bỏ qua một phần tử.

Lưu ý: Chọn phương pháp dựa trên tính dễ đọc và độ phức tạp của logic.

---

4. Hiển thị Danh sách (Rendering Lists)

React sử dụng `Array.prototype.map()` để chuyển đổi mảng dữ liệu thành mảng các phần tử JSX. Mỗi phần tử cần có một prop `key` duy nhất để React quản lý hiệu quả.

Sử dụng `map`

```jsx
function ItemList({ items }) {
  return (
    <ul>
      {items.map(item => (
        <li key={item.id}>{item.name}</li>
      ))}
    </ul>
  );
}
```

Giải thích:

- `items.map()` chuyển mỗi phần tử trong mảng `items` thành một phần tử `<li>`.
- Prop `key` giúp React nhận diện từng phần tử, tối ưu hóa việc cập nhật DOM.

Tầm quan trọng của `key`

- `key` là một định danh duy nhất (chuỗi hoặc số) cho mỗi phần tử trong danh sách.
- React sử dụng `key` để theo dõi các phần tử đã thay đổi, được thêm hoặc xóa.
- Nếu thiếu `key`, React có thể render lại toàn bộ danh sách, gây giảm hiệu suất.

Sai lầm phổ biến: Sử dụng chỉ số (index) làm `key`

Sử dụng chỉ số của mảng (`index`) làm `key` là không nên, vì nó có thể gây lỗi khi danh sách thay đổi (thêm, xóa, sắp xếp):

```jsx
// Sai
items.map((item, index) => <li key={index}>{item.name}</li>)
// Đúng
items.map(item => <li key={item.id}>{item.name}</li>)
```

Giải thích thêm: Sử dụng `id` duy nhất từ dữ liệu đảm bảo React có thể theo dõi chính xác các phần tử, đặc biệt khi danh sách thay đổi.

C. Bài tập thực hành

Bài 1: Tạo Component SearchBar

Nhiệm vụ: Tạo component `SearchBar.js` với một ô input, in giá trị nhập vào console khi có thay đổi.

Bước thực hiện:

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

```
function SearchBar() {
  function handleChange(event) {
    console.log(event.target.value);
  }
  return <input type="text" onChange={handleChange} placeholder="Tìm kiếm...
}
export default SearchBar;
```

2. Sử dụng trong `App.jsx`:

```
import SearchBar from './components/SearchBar';
function App() {
  return <SearchBar />;
}
export default App;
```

Giải thích: Mỗi khi người dùng nhập, giá trị được in ra console thông qua sự kiện `onChange`. Đây là ví dụ cơ bản về xử lý sự kiện trong React.

Bài 2: Nâng cấp trạng thái với SearchBar

Nhiệm vụ: Trong `App.js`, quản lý trạng thái `searchTerm`. Truyền `searchTerm` và hàm `handleSearchChange` vào `SearchBar`. Cập nhật `searchTerm` khi người dùng nhập và hiển thị giá trị trong `App`.

Bước thực hiện:

1. Cập nhật `src/App.jsx`:

```
import { useState } from 'react';
import SearchBar from './components/SearchBar';

function App() {
  const [searchTerm, setSearchTerm] = useState('');

  function handleSearchChange(newTerm) {
    setSearchTerm(newTerm);
  }

  return (
    <div>
      <SearchBar searchTerm={searchTerm} onSearchChange={handleSearchChange}
      <p>Tìm kiếm hiện tại: {searchTerm}</p>
    </div>
  );
}
export default App;
```

2. Cập nhật `src/components/SearchBar.js`:

```
function SearchBar({ searchTerm, onSearchChange }) {
  return (
    <input
      type="text"
      value={searchTerm}
      onChange={(e) => onSearchChange(e.target.value)}
```

```
      placeholder="Tìm kiếm..."
    />
  );
}
export default SearchBar;
```

Giải thích: `App` quản lý trạng thái `searchTerm`, truyền nó và hàm `handleSearchChange` vào `SearchBar`. Khi người dùng nhập, `SearchBar` gọi `onSearchChange` để cập nhật trạng thái trong `App`.

Bài 3: Hiển thị điều kiện với LoginControl

Nhiệm vụ: Tạo component `LoginControl.js` với trạng thái `isLoggedIn`. Hiển thị nút "Đăng xuất" và "Chào mừng bạn trở lại!" khi đăng nhập, hoặc nút "Đăng nhập" và "Vui lòng đăng nhập." khi chưa đăng nhập. Chuyển đổi trạng thái khi nhấn nút.

Bước thực hiện:

1. Tạo `src/components/LoginControl.js`:

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

function LoginControl() {
  const [isLoggedIn, setIsLoggedIn] = useState(false);

  function handleToggle() {
    setIsLoggedIn(!isLoggedIn);
  }

  return (
    <div>
      <button onClick={handleToggle}>{isLoggedIn ? 'Đăng xuất' : 'Đăng nhập'
      <p>{isLoggedIn ? 'Chào mừng bạn trở lại!' : 'Vui lòng đăng nhập.'}</p>
    </div>
  );
```

```
}
export default LoginControl;
```

2. Sử dụng trong `App.jsx`:

```
import LoginControl from './components/LoginControl';
function App() {
  return <LoginControl />;
}
export default App;
```

Giải thích: Component sử dụng trạng thái `isLoggedIn` để hiển thị điều kiện. Nút chuyển đổi trạng thái thông qua `handleToggle`, minh họa cách sử dụng toán tử ba ngôi và xử lý sự kiện.

Bài 4: Hiển thị danh sách sản phẩm

Nhiệm vụ: Tạo component `ProductList.js` nhận một mảng sản phẩm (`id`, `name`, `price`) và hiển thị dưới dạng danh sách `<ul>` với các phần tử `<li>`, sử dụng `id` làm `key`.

Bước thực hiện:

1. Tạo `src/components/ProductList.js`:

```
function ProductList({ products }) {
  return (
    <ul>
      {products.map(product => (
        <li key={product.id}>
          {product.name} - ${product.price}
        </li>
```

```
      ))}
    </ul>
  );
}
export default ProductList;
```

2. Sử dụng trong `App.jsx`:

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

const products = [
  { id: 1, name: 'Máy tính xách tay', price: 999 },
  { id: 2, name: 'Điện thoại', price: 499 },
  { id: 3, name: 'Tai nghe', price: 99 },
];

function App() {
  return <ProductList products={products} />;
}
export default App;
```

Giải thích: Component `ProductList` sử dụng `map` để hiển thị danh sách sản phẩm. Prop `key` đảm bảo mỗi `<li>` được nhận diện duy nhất, giúp tối ưu hiệu suất.

Bài 5: Xây dựng ứng dụng Todo List

Nhiệm vụ: Tạo ứng dụng Todo List với:

- `App` quản lý trạng thái `todos` (mảng các đối tượng todo).
- `TodoForm` để thêm todo mới (nâng cấp trạng thái).
- `TodoList` để hiển thị danh sách todo (hiển thị danh sách).
- Mỗi todo có nút "Xóa" và nút "Hoàn thành/Đảo ngược" (hiển thị điều kiện cho gạch ngang).

Bước thực hiện:

1. Tạo `src/App.jsx`:

```jsx
import { useState } from 'react';
import TodoForm from './components/TodoForm';
import TodoList from './components/TodoList';

function App() {
  const [todos, setTodos] = useState([]);

  function addTodo(text) {
    setTodos([...todos, { id: Date.now(), text, completed: false }]);
  }

  function deleteTodo(id) {
    setTodos(todos.filter(todo => todo.id !== id));
  }

  function toggleComplete(id) {
    setTodos(
      todos.map(todo =>
        todo.id === id ? { ...todo, completed: !todo.completed } : todo
      )
    );
  }

  return (
    <div>
      <h1>Danh sách công việc</h1>
      <TodoForm onAdd={addTodo} />
      <TodoList todos={todos} onDelete={deleteTodo} onToggle={toggleComplete
    </div>
  );
}
export default App;
```

2. Tạo `src/components/TodoForm.js`:

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

function TodoForm({ onAdd }) {
  const [text, setText] = useState('');

  function handleSubmit(e) {
    e.preventDefault();
    if (text.trim()) {
      onAdd(text);
      setText('');
    }
  }

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={text}
        onChange={(e) => setText(e.target.value)}
        placeholder="Thêm công việc"
      />
      <button type="submit">Thêm</button>
    </form>
  );
}
export default TodoForm;
```

Cập nhật: Sửa lỗi trong tài liệu gốc, sử dụng thẻ `<form>` thay vì `<div>` cho `onSubmit` để xử lý sự kiện gửi form đúng cách.

3. Tạo `src/components/TodoList.js`:

```
function TodoList({ todos, onDelete, onToggle }) {
  return (
    <ul>
      {todos.map(todo => (
        <li key={todo.id} style={{ textDecoration: todo.completed ? 'line-th
          {todo.text}
```

```
            <button onClick={() => onToggle(todo.id)}>
              {todo.completed ? 'Đảo ngược' : 'Hoàn thành'}
            </button>
            <button onClick={() => onDelete(todo.id)}>Xóa</button>
          </li>
        ))}
      </ul>
    );
}
export default TodoList;
```

Giải thích:

- `App` quản lý danh sách `todos` và cung cấp các hàm `addTodo`, `deleteTodo`, `toggleComplete`.
- `TodoForm` cho phép thêm công việc mới, sử dụng trạng thái cục bộ `text` và gửi dữ liệu lên `App` qua `onAdd`.
- `TodoList` hiển thị danh sách todo, sử dụng `key` và hiển thị điều kiện để gạch ngang công việc đã hoàn thành.

D. Kiểm tra và cập nhật

Kiểm tra nội dung gốc:

- Nội dung gốc chính xác về mặt lý thuyết và ví dụ, nhưng thiếu giải thích chi tiết cho học viên mới.
- Bài tập 5 có lỗi nhỏ trong `TodoForm.js`: sử dụng `<div>` thay vì `<form>` cho sự kiện `onSubmit`, gây lỗi khi xử lý gửi form.
- Cập nhật đã sửa lỗi này và bổ sung giải thích chi tiết bằng tiếng Việt để dễ hiểu hơn.
- Nội dung được giữ nguyên cấu trúc nhưng được viết lại để rõ ràng, phù hợp với học viên Việt Nam.

Lưu ý cho học viên:

- Thực hành các bài tập trên môi trường React (sử dụng Create React App hoặc Vite).
- Đảm bảo cài đặt React và các công cụ cần thiết trước khi bắt đầu.
- Sử dụng console để kiểm tra kết quả và debug khi cần.