



Lesson 6: More about State & Props

🕒 Created March 29, 2022 6:02 PM

🏷 Tags Empty

🔗 Related to Untitle... Empty

💡 Sử dụng kết hợp state và props để xây dựng các ứng dụng khác nhau.

1. Xử lý Form với React

Form là một trong những thành phần luôn xuất hiện trong tất cả các ứng dụng web. Xét ví dụ bên dưới:

```
const App = () => { const handleSubmit = (event) => { event.preventDefault() }  
return ( <form onSubmit={handleSubmit}> <input type="text" /> <button  
type="submit">Submit</button> </form> ) }
```

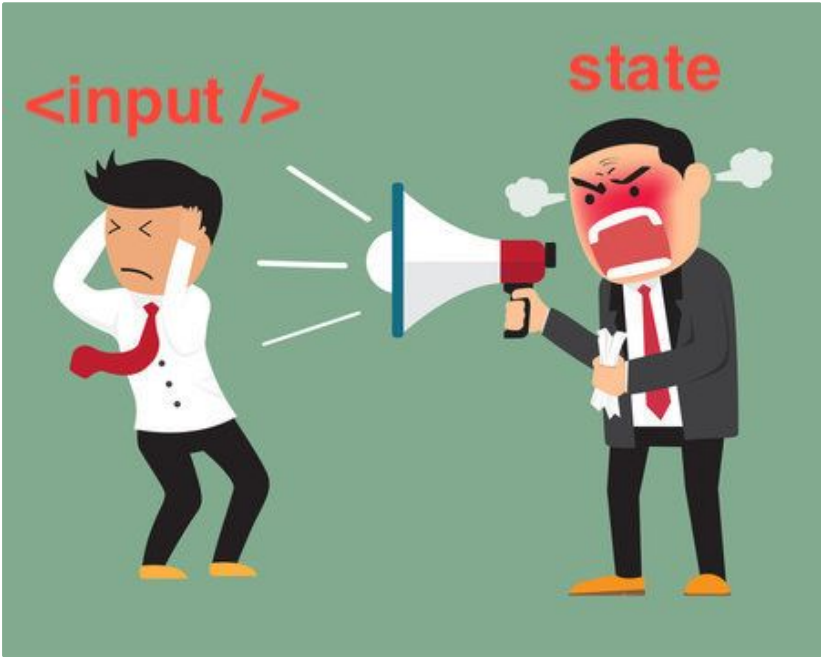
Với các function xử lý các event như `handleSubmit`, chúng ta luôn nhận được tham số là event đó. Do đó với form, chúng ta có thể gọi `event.preventDefault()` như trong ví dụ.

Có nhiều cách để có thể lấy được giá trị từ thẻ `input` bên trong form. Tuy nhiên chúng ta sẽ sử dụng state để có thể lấy được giá trị đó ra. Lúc này, ứng dụng sẽ trở thành như sau:

```
import {useState} from 'react' const App = () => { const [input, setInput] =  
useState("") const handleSubmit = (event) => { event.preventDefault()  
console.log(input) } return ( <form onSubmit={handleSubmit}> <input  
type="text" value={input} /> <button type="submit">Submit</button> </form> ) }
```

Lúc này, giá trị của input luôn bằng với giá trị của state `input` được khai báo ở trên.

Tuy nhiên, nếu người dùng cố gắng thay đổi giá trị của thẻ input, thẻ input hoàn toàn không phản ứng lại với sự thay đổi đó.




Props trong React không thể bị thay đổi từ bên trong component đó. Thẻ input nhận vào một “props” là `value`. Và nó không thể tự cập nhật value cho chính nó. Tuy nhiên, giá trị value này lại đến từ `input`, là một state trong component `App`. Vì state trong React hoàn toàn có thể thay đổi được giá trị thông qua `setState`, nên chúng ta có thể cập nhật giá trị `input`, qua đó có thể cập nhật được value của input.

Để thay đổi được giá trị của `input` state thông qua việc thay đổi giá trị của thẻ input, ta có thể làm như sau:

```
import {useState} from 'react' const App = () => { const [input, setInput] =
useState('') const handleSubmit = (event) => { event.preventDefault()
console.log(input) setInput('') } const handleChange = (event) => {
setInput(event.target.value) } return ( <form onSubmit={handleSubmit}> <input
type="text" value={input} onChange={handleChange} /> <button
type="submit">Submit</button> </form> ) }
```

Với việc thay đổi giá trị `input` thông qua event `onChange` của input, ta đã có thể thay đổi và lấy được giá trị nằm trong input. Thêm vào đó, khi muốn thay đổi giá trị bên trong input, ta tìm cách cập nhật `input` state, và React sẽ tiến hành cập nhật giao diện giúp chúng ta.

 Hãy tạo form với nhiều loại input như select, checkbox, radio button, text area, ... và xử lý chúng với React state.

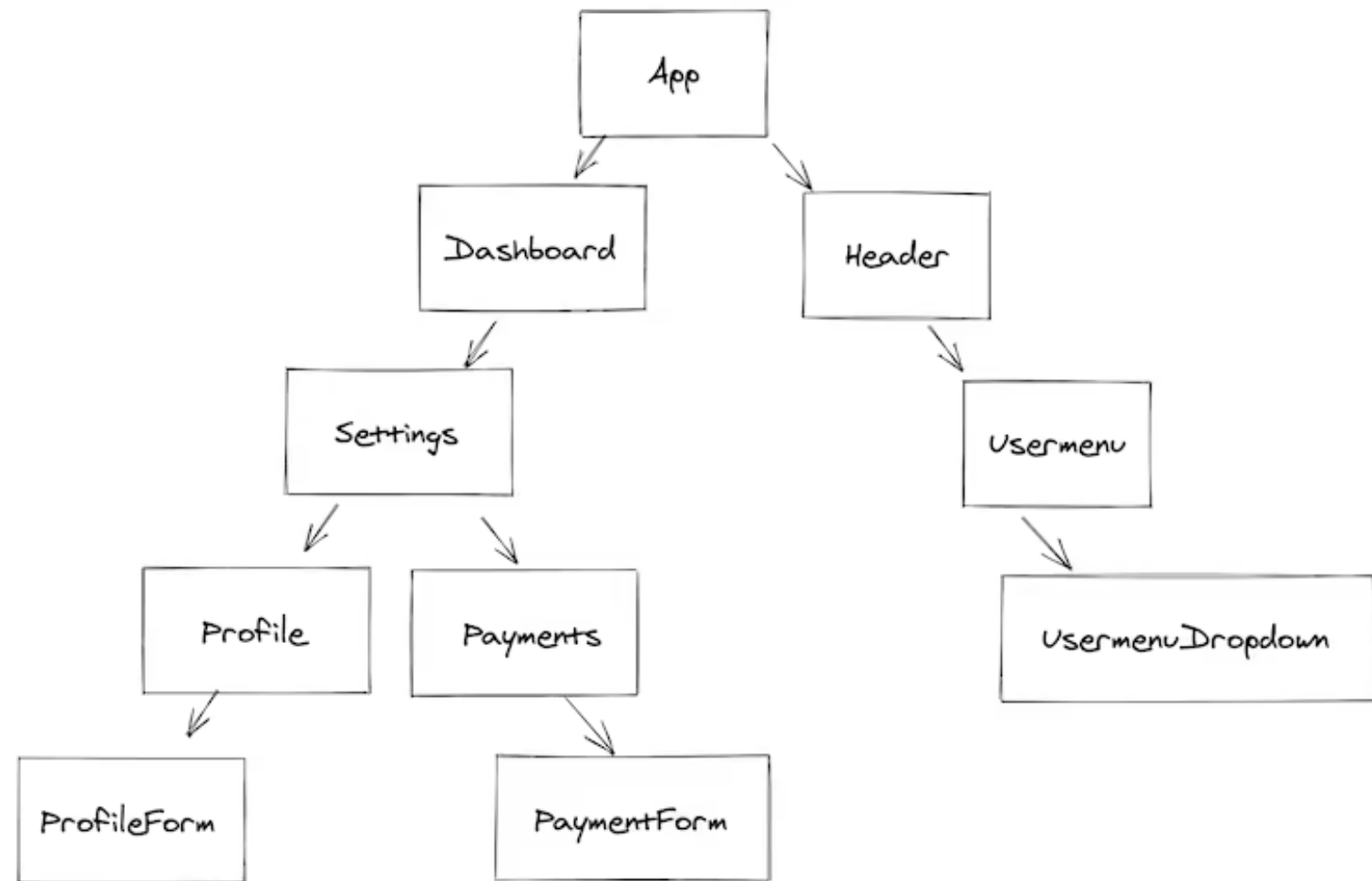
Nâng cao: thay vì sử dụng nhiều state cho mỗi input, hãy thử sử dụng 1 state để xử lý nhiều input.

2. Component tree

Với một ứng dụng thông thường, ta có thể có một cấu trúc như sau:

```
ReactDOM.render( <App> <Header> <UserMenu> <UserMenuDropdown /> </UserMenu>
</Header> <Dashboard> <Settings> <Profile> <ProfileForm /> </Profile>
<Payment> <PaymentForm /> </Payment> </Settings> </Dashboard> </App> )
```

Có nhiều thành phần tạo nên một ứng dụng web. Các component lớn render các component nhỏ hơn, tạo thành một mô hình cây như hình dưới đây:



Trong React, mô hình trên được gọi là “component tree”. Ý tưởng của component tree giống với HTML DOM tree, được sử dụng để mô hình hoá giao diện ứng dụng.

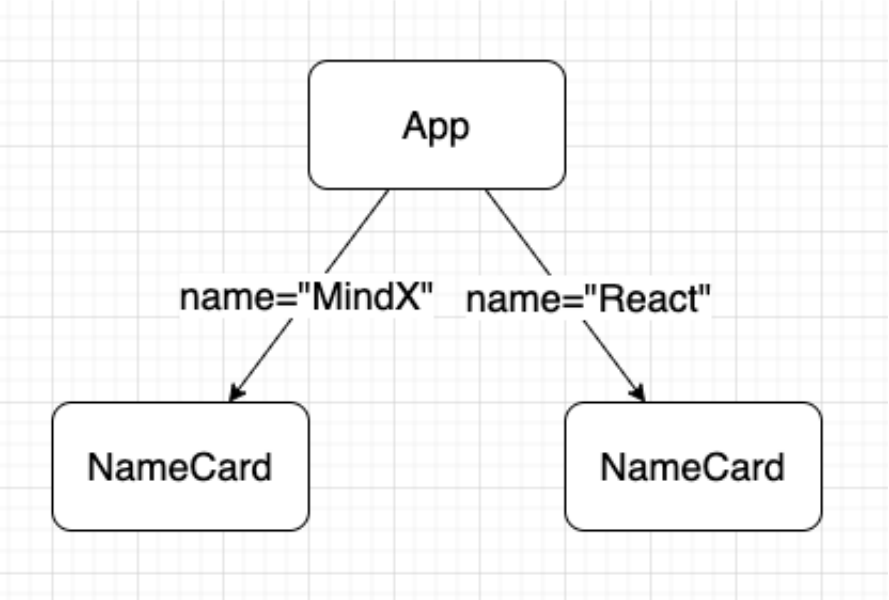
- 💡 Các khái niệm liên qua tới cây gồm:
- Node: các điểm nằm trên cây. Tất cả các component trong hình đều là các node
 - Parent Node: Node cha. Là node nằm ngay trên node hiện tại, chịu trách nhiệm render ra nốt hiện tại và các “sibling” của nó. VD: `Settings` là parent node của `Profile`
 - Child Node: Node con. Trái ngược với Parent node. VD: `UserMenu` là node con của `Header`
 - Sibling Nodes: Node anh em. Là các node ngang hàng nhau, có cùng cha: VD: `Profile` và `Payments`
 - Ancestors: tổ tiên. Là các nodes ở trên node cha, chịu trách nhiệm render ra node cha. VD: `DashBoard` `App` là các node ancestors của `Settings`. `Header` **không phải** là ancestor của `Settings`
 - Descendants: con cháu. Là các node được render từ node hiện tại. VD: `Profile` `Payment` là các node descendant của `Settings`. `UserMenuDropdown` **không phải** là descendants của `Settings`.
 - Sub tree: Cây con. Là tập hợp các node bắt đầu từ node hiện tại. VD: sub tree từ `Settings` sẽ bao gồm `Profile` `Payments` `ProfileForm` và `PaymentForm`

“Data down”

Props chính là công cụ để giao tiếp từ node cha xuống node con. Xét ví dụ:

```
const App = () => { const [people, setPeople] = useState([ {name: "MindX"}, {name: "React"} ]) return ( <div> <NameCard name={people[0].name} /> <NameCard name={people[1].name} /> </div> ) } const NameCard = (props) => { return <div>Hello, my name is {props.name}</div> }
```

Ta có component tree như sau:



Hai component `NameCard` đều nhận các giá trị `name` từ component cha của chúng là `App`. Và đây cũng chính là luồng dữ liệu của React: **Data luôn di chuyển từ component cha xuống component con.**

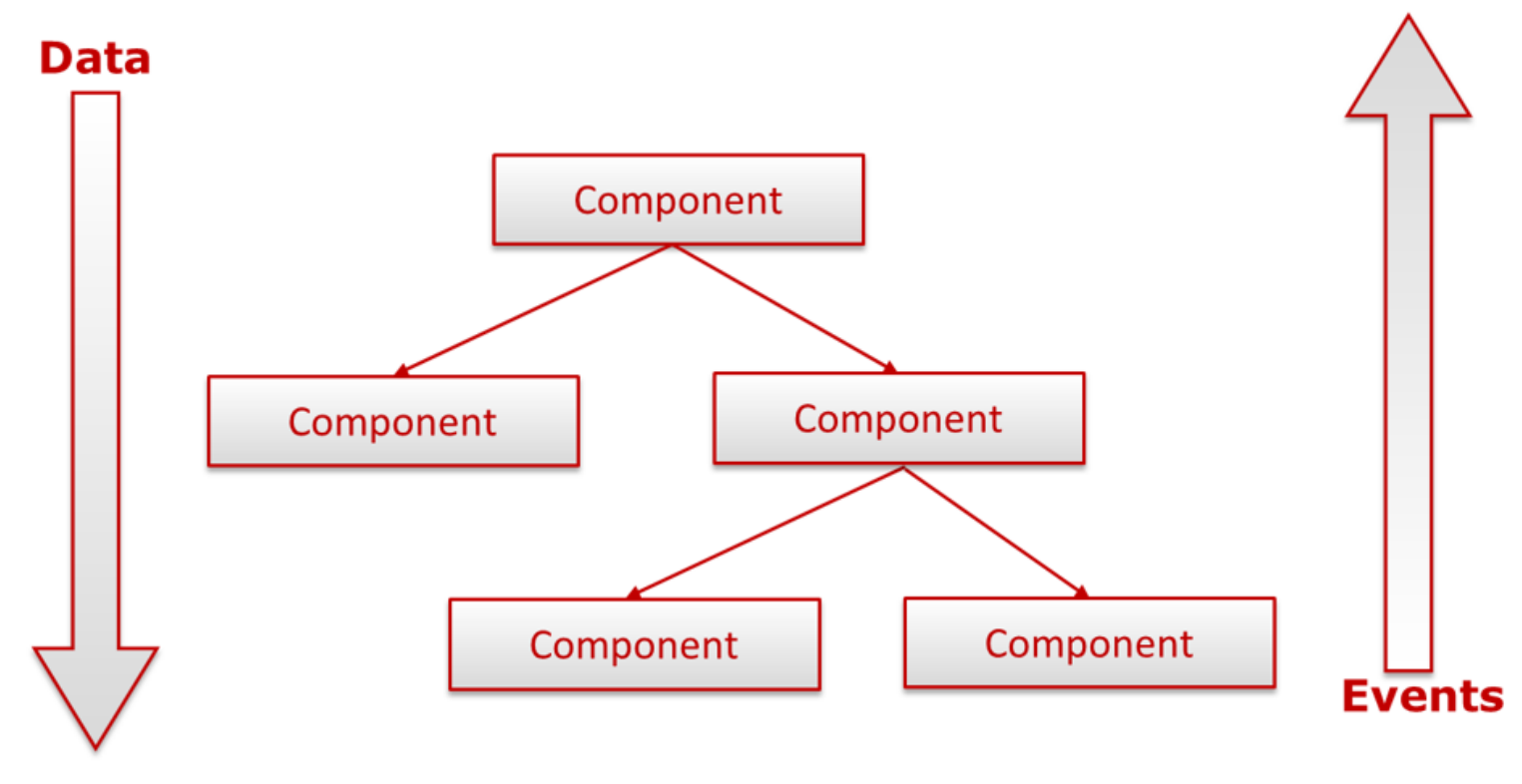
“Event up”

Props cho phép người dùng có thể truyền các dạng giá trị JS khác nhau từ component cha xuống component con, bao gồm cả các “functions”. Chúng ta sẽ truyền xuống các function từ phía component cha xuống component con. Và các component con sẽ thực hiện chạy các function đó khi cần thiết.

```
const App = () => { const [alice, setAlice] = useState({name: "Alice", age: 10}) const [bob, setBob] = useState({name: "Bob", age: 10}) const increaseAliceAge = () => { setAlice({ ...alice, age: alice.age + 1, }) } const increaseBobAge = () => { setBob({ ...bob, age: bob.age + 1, }) } return ( <div> <NameCard name={alice.name} age={alice.age} onIncreaseAge={increaseAliceAge} /> <NameCard name={bob.name} age={alice.age} onIncreaseAge={increaseBobAge} /> </div> ) } const NameCard = (props) => { return ( <div> <div>Hello, my name is {props.name}. I'm {props.age} years old</div> <button onClick={props.onIncreaseAge}>Increase age</button> </div> ) }
```

Việc tăng số tuổi của mỗi state bên trong ứng dụng trên được thực hiện khi người dùng click vào button “Increase age” bên trong component `NameCard`. Tuy nhiên, `age` là một props của `NameCard`, do đó, nó không thể được chỉnh sửa trực tiếp từ trong chính component đó. Giải pháp ở đây là tạo ra function `increase...Age` ở bên trong component `App`. Vì `alice` và `bob` đều là các state của `App` nên các function đó đều có thể thay đổi được giá trị tuổi của hai biến state trên. Sử dụng props để đưa các function đó xuống các component bên dưới. Và khi chúng được thực thi, state của `App` sẽ được cập nhật, dẫn tới cập nhật toàn bộ ứng dụng.

Như vậy, luồng dữ liệu của React được luôn chuyển theo công thức: **Data down - Events up**



3. Lifting state Up

Các ứng dụng web hiện đại đều muốn có sự chia sẻ dữ liệu giữa các components. Để thực hiện được điều này trong ứng dụng React cần phải sử dụng một kĩ thuật đặc biệt: “Lifting state up”.

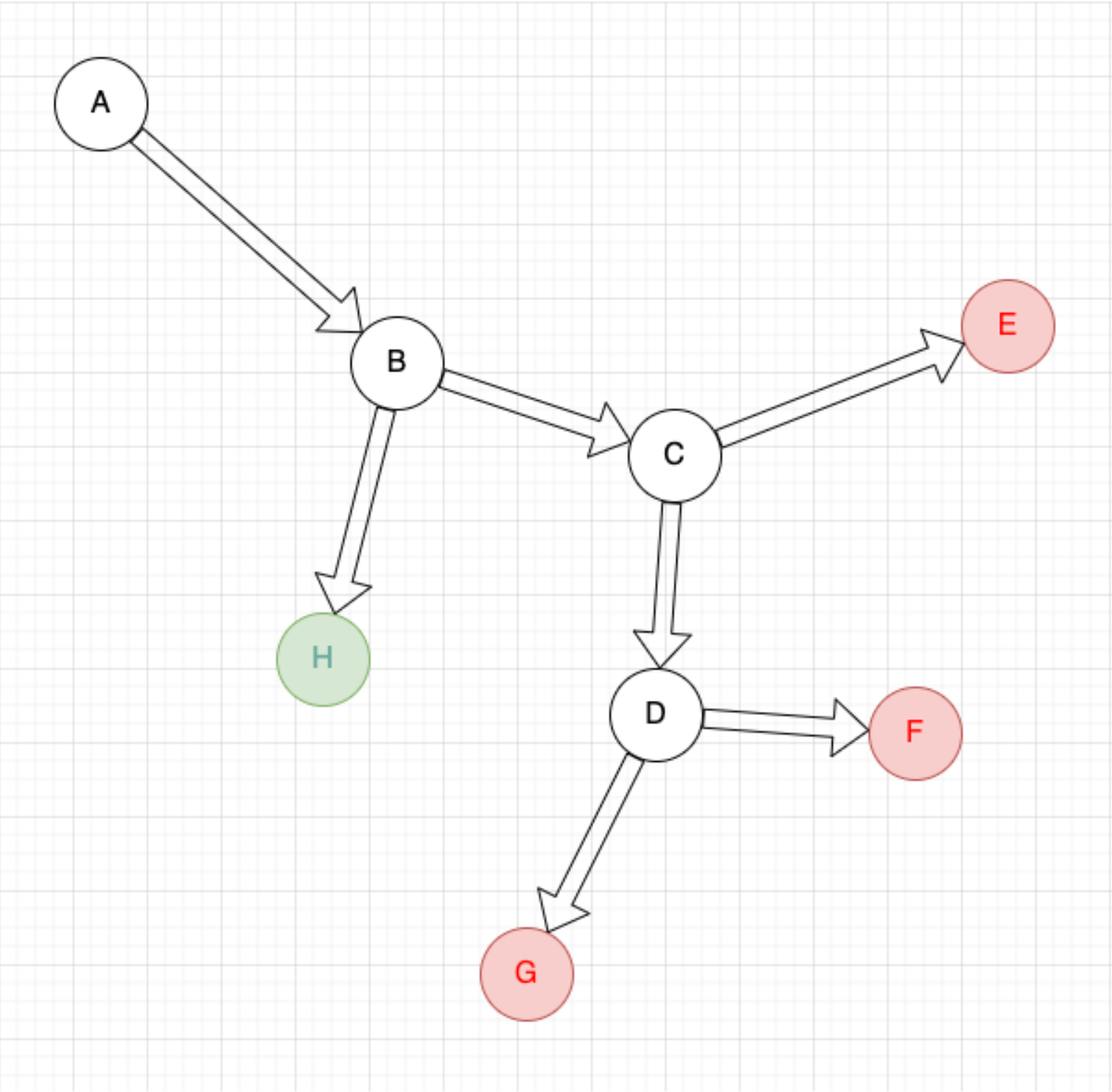
Xét ví dụ sau:

```
import {useState} from 'react' const App = () => { return ( <div> <ProductList /> <Cart/> </div> ) } const ProductList = () => { const [products, setProducts] = useState([]) return (...) // UI here } const Cart = () => { const [cartItems, setCartItems] = useState([]) return (...) // UI here }
```

Làm thế nào để component `Cart` có thể thêm được sản phẩm vào khi người dùng click “Add to cart” từ `ProductList`?

Do dữ liệu luôn được luân chuyển theo hướng từ trên xuống dưới, component `Cart` và component `ProductList` là 2 node sibling, chúng không thể trực tiếp trao đổi dữ liệu cho nhau. Với luồng dữ liệu được luân chuyển từ trên xuống dưới như React, ta cần phải sử dụng một kĩ thuật đặc biệt là “lifting state up”.

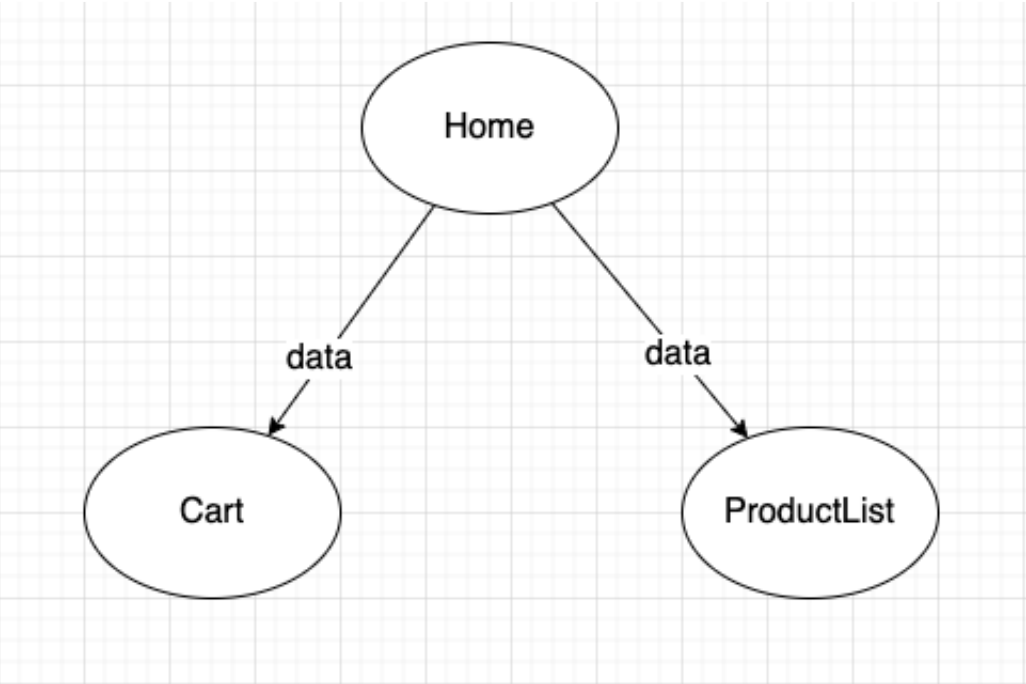
“Lifting state up” cơ bản là đặt state đúng vị trí của chúng (thường nói đến việc đưa các giá trị state lên các tầng cao hơn) để đạt được mục đích chia sẻ dữ liệu giữa các thành phần.



Giả sử đây là sơ đồ của một trận chiến ven sông. Các điểm màu đỏ là nơi địch đóng quân. Điểm màu xanh là nơi ta đóng quân. Dòng sông có hướng chảy theo chiều mũi tên. Nhiệm vụ của chúng ta là cần tiêu hao sinh lực kẻ thù bằng cách sử dụng thuốc độc đổ xuống sông, khiến địch không có nước sinh hoạt. Vậy chúng ta nên đổ thuốc độc ở điểm nào trên bản đồ?

Như đã biết ở trên, dữ liệu trong React chỉ có thể đi theo một chiều (từ component cha xuống component con). Các component “sibling” không thể truyền được dữ liệu cho nhau. Có nghĩa là khi một bên cập nhật dữ liệu, bên còn lại sẽ không được cập nhật đồng bộ theo.

Giải pháp ở đây là cần đưa dữ liệu đó lên level cao hơn trong cây component. Và truyền xuống bên dưới thông qua props.



Khi này, cả `Cart` và `ProductList` đều nhận chung một nguồn dữ liệu từ component `Home` truyền xuống, đảm bảo được một điều quan trọng: **chúng có chung một nguồn dữ liệu gốc (single source of truth)**. Việc chỉnh sửa dữ liệu bên trong component `Home` sẽ làm cả 2 thành phần bên dưới đều được thay đổi. Khi cần thay đổi dữ liệu của cả 2 component `Cart` và `ProductList`, ta luôn cần phải nhớ một điều: Hãy cập nhật dữ liệu ở nguồn của chúng.

Một khi dữ liệu gốc bị thay đổi, React, giống như dòng sông, sẽ đưa dữ liệu đó tới khắp các component cần chịu ảnh hưởng, và tiến hành tính toán lại các component đó cho chúng ta.

```
import {useState} from 'react' const App = () => { const [products, setProducts] = useState([]) const [cartItems, setCartItems] = useState([]) const addToCart = (p) => { setCartItems((prev) => { return [ ...prev, { productId: p.id, quantity: 1, price: p.price, } ] }) } return ( <div> <ProductList products={products} onAddToCart={addToCart} /> <Cart cartItems={cartItems} /> </div> ) } const ProductList = (props) => { return (...) // UI here } const Cart = (props) => { return (...) // UI here }
```



Để chọn được đúng vị trí đặt dữ liệu cần chia sẻ giữa component A và B, chúng ta cần tìm tới node tổ tiên chung gần nhất của cả 2 component đó. Ví dụ với hình minh hoạt ở trên, cha chung gần nhất của F và G là D, của E và G là C.