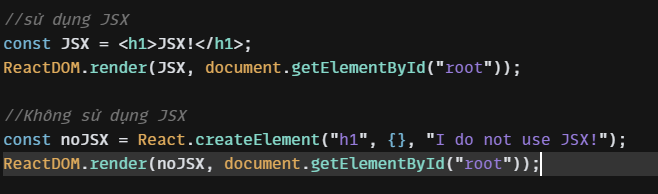
**Tìm hiểu về ReactJS**

1. NPM

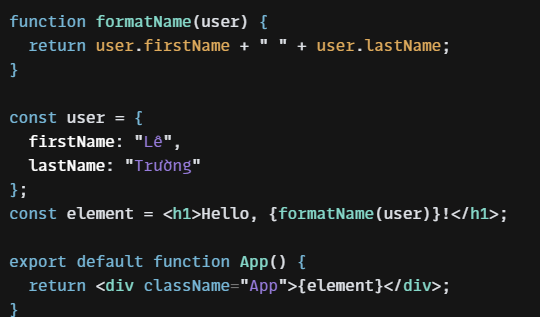
* **NMP** là viết tắt của Node package manager là một công cụ tạo và quản lý các thư viện lập trình Javascript cho [Node.js](https://nodejs.org/). Trong cộng đồng Javascript, các lập trình viên chia sẻ hàng trăm nghìn các thư viện với các đoạn code đã thực hiện sẵn một chức năng nào đó. Nó giúp cho các dự án mới tránh phải viết lại các thành phần cơ bản, các thư viện lập trình hay thậm chí cả các [framework](https://topdev.vn/blog/framework-la-gi/).
* **NPM** có sẵn khi bạn tải NodeJS về. Để kiểm tra xem trên hệ thống của bạn đã được cài npm chưa chúng ta sử dụng lệnh npm -v, nếu một phiên bản hiện ra thì hệ thống của bạn đã được cài đặt npm.
* **NPM** cung cấp 2 chức năng chính bao gồm:
* Là kho lưu trữ trực tuyến cho các package/module. Chúng ta có thể tìm kiếm các package trên [search.nodejs.org](http://search.nodejs.org/).
* Quản lý các module javascript và phiên bản của chúng trong các dự án của chúng ta đơn giản hơn, dễ dàng hơn, tiết kiệm thời gian hơn.
* Có 2 cách để cài đặt một gói bằng npm:
* **Local:** sẽ tạo ra thư mục node\_modules nếu chưa có trong project hoặc nếu có rồi nó sẽ lấy code của gói cần cài đặt đưa vào đây, tức chỉ hiện diện trong thư mục của project hiện tại. Khi cần sử dụng bạn có thể sử dụng lệnh require().
* **Global:** sẽ lưu trữ code của gói trong các file hệ thống cố định trong máy, chỉ có thể dùng các package này thông qua các hàm CLI (Command Line Interface) ví dụ như gulp. Không thể dùng package thông qua require().
* Để quản lý các gói cài đặt cục bộ bằng **npm** thì cách tốt nhất là thông qua file **package.json**, chính là file nằm trong thư mục gốc của project. File [JSON](https://topdev.vn/blog/json-la-gi/) này chứa các nội dung:
* Các gói thư viện lập trình mà project sử dụng.
* Cho phép xác định phiên bản chính xác của các gói thư viện lập trình được sử dụng.
* Các gói bạn xây dựng có thể chia sẻ dễ dàng với các lập trình viên khác trên toàn cầu thông qua npm.

1. JSX

* **JSX** là viết tắt của JavaScript XML, một cú pháp mở rộng cho phép lập trình viên viết HTML trong React một cách dễ dàng. React sử dụng JSX để tạo khuôn mẫu thay vì JavaScript thông thường. Bạn không bắt buộc phải sử dụng JSX nhưng phần mở rộng này sẽ giúp bạn viết ứng dụng dễ dàng hơn.
* Một số ưu điểm của JSX trong ReactJs
* Mang lại tốc độ nhanh hơn vì nó thực hiện tối ưu hóa khi biên dịch mã sang JavaScript.
* Hầu hết các lỗi đều có thể được phát hiện ngay trong quá trình biên dịch mã.
* Nếu bạn quen thuộc với HTML, JSX sẽ giúp bạn viết các mẫu (templates) nhanh và dễ dàng hơn.
* JSX cho phép lập trình viên viết các phần tử HTML bằng JavaScript và đặt chúng trong DOM mà không cần bất kỳ phương thức nào createElement()  và/hoặc appendChild().



* Với JSX, bạn có thể viết các biểu thức bên trong dấu ngoặc nhọn {}. Biểu thức có thể là một biến hoặc thuộc tính hoặc bất kỳ biểu thức JavaScript hợp lệ nào khác. JSX sẽ thực thi biểu thức và trả về kết quả.

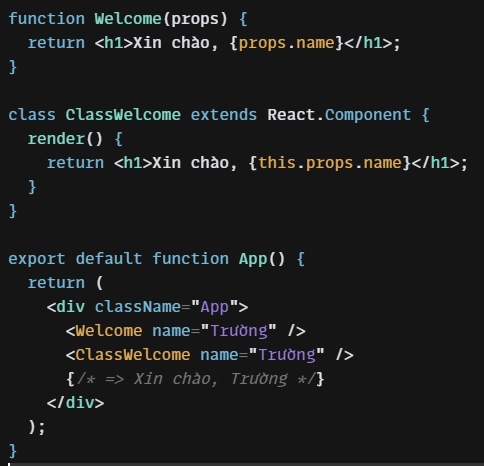


* Thuộc tính chỉ định với JSX



1. **Components and Props**

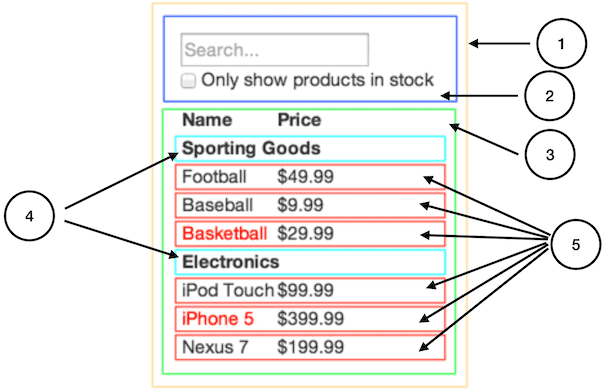
* **Components**
* Components cho phép bạn chia UI thành các phần độc lập, có thể tái sử dụng và bạn có thể sử dụng chúng 1 cách độc lập
* Components cũng giống như function JavaScript. Nó cho phép nhập vào các đầu vào tùy ý (props) và trả về những phần tử React mô tả những gì sẽ xuất hiện trên màn hình.
* Có 2 loại gồm Function và Class Components

****

* Các component có thể tham chiếu đến các component khác tại output của nó. Điều này cho phép chúng ta tái sử dụng lại component với các props khác nhau.

****

* Để cho việc dễ quản lý, bảo trì và kiểm thử, chúng ta cần chia các component cho một phần của mô hình dữ liệu.

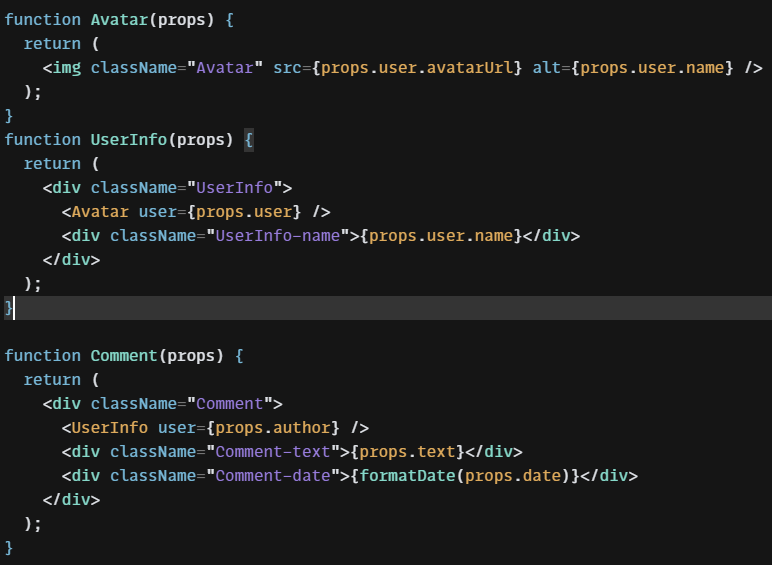


Như ứng dụng trên chúng ta sẽ chia nhỏ ra 5 components như trên:

* **FilterableProductTable (orange):** chứa toàn bộ cả ứng dụng
* **SearchBar (blue):** nơi *người dùng nhập từ khoá tìm kiếm*
* **ProductTable (green):** lọc và hiển thị *kết quả* dựa trên *từ khoá tìm kiếm*
* **ProductCategoryRow (turquoise):** hiển thị trương mục theo *thể loại*
* **ProductRow (red):** hiển thị *sản phẩm* theo từng dòng

****

Từ 1 components lớn ban đầu chúng ta có thể chia nhỏ nó ra



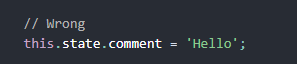
* **Props**
* **Props** là một object được truyền vào trong một components, mỗi components sẽ nhận vào props và trả về react element.  
  **Props** cho phép chúng ta giao tiếp giữa các components với nhau bằng cách truyền tham số qua lại giữa các components.
* Khi một components cha truyền cho component con một props thì components con chỉ có thể đọc và không có quyền chỉnh sửa nó bên phía components cha.
* Cho dù là khi khai báo 1 component bằng function hay class, nó không bao giờ được sửa đổi prop của chính nó.

1. **State and Lifecycle**

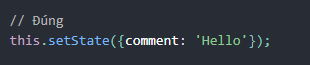
* State
* **State** là một object có thể được sử dụng để chứa dữ liệu hoặc thông tin về components. State có thể được thay đổi bất cứ khi nào mong muốn. Khác với props bạn có thể truyền props sang các components khác nhau thì state chỉ tồn tại trong phạm vi của components chứa nó, mỗi khi state thay đổi thì components đó sẽ được render lại.
* Trong các dự án React, state được dùng để phản hồi các yêu cầu từ người dùng, hay lưu trữ một dữ liệu nào đó trong components.
* Không nên sử dụng **State** cho ứng dụng tĩnh.

**Sử dụng State đúng cách:**

* Không nên sửa State trức tiếp



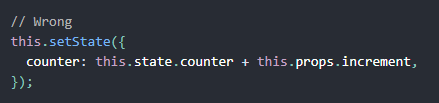
Thay vào đó chúng ta sử dụng **setState():**



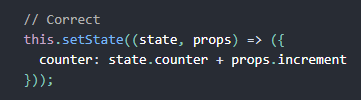
* Cập nhật State là bất đồng bộ:

Bởi vì this.props và this.state có thể được cập nhập bất đồng bộ, bạn không nên dựa vào giá trị của chúng để tính toán state tiếp theo

Ví dụ, đoạn mã này có thể thất bại để cập nhập counter:



Để sửa nó, sử dụng một dạng thứ hai của setState() nó nhận một funtion chứ không phải object. Funtion này sẽ nhận state trước đó làm tham số đầu tiên, và props tại thời điểm cập nhập được nhận làm tham số thứ hai:

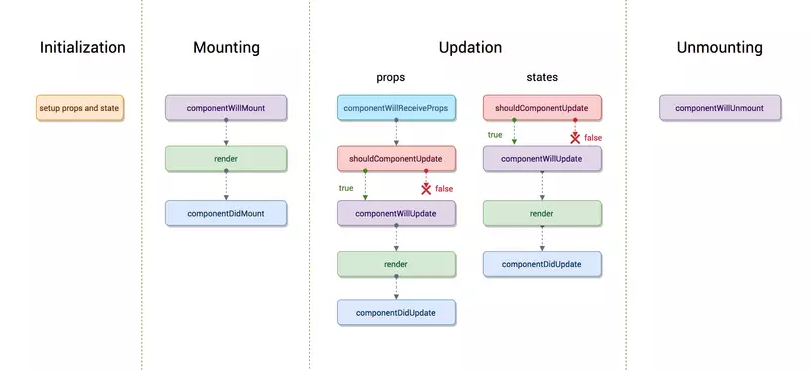


* Dữ liệu được truyền từ trên xuống dươi

Một component có thể chọn để truyền state của nó như một prop cho những child component của nó.

* **Lifecycle**

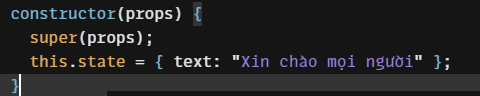
Hình ảnh mình họa cho vòng đời(**Lifecycle**) của compponent trong React.



1. Giaiđoạn Initialization

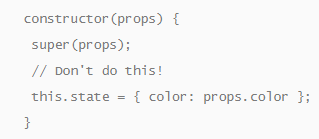
Đây là giai đoạn đầu tiên của một component, bắt đầu bằng cách khởi tạo state, props, các biến cần thiết hoặc bind các function thường. Điều này thực hiện bên trong phương thức constructor().

Method constructor() được khai báo phía trên tất cả các phương thức khác, và được gọi trước khi component được mount (render), nó nhận một argument là props.



Những điều cần tránh khi sử dụng hàm constructor:

* Hàm constructor() là nơi duy nhất bạn có thể sử dụng this.state để gán giá trị trực tiếp. Trong tất cả các hàm khác, bạn phải sử dụng this.setState().
* Tránh xử lý tất cả các side-effect hoặc subscriptions trong đây.
* Tránh sử dụng props làm giá trị của state. Đây là một lỗi hay gặp phải với những bạn mới bắt đầu.



1. Giai đoạn Mounting

Giai đoạn này được thực hiện sau khi khởi tạo xong. Đây là quá trình gắn React element (Virtual DOM) của một component vào Real DOM để thể hiện kết quả lên trình duyệt.

Mouting có 3 phương thức là:

* **ComponentWillMount()**

Phương thức này được khởi chạy trước khi một component được mount (hay còn gọi là render) lần đầu tiên duy nhất. Nghĩa là nó khởi chạy sau hàm constructor() và trước hàm render().

**Chú ý**: Không nên cập nhật giá trị state hoặc props trong phương thức này, vì thời gian chuẩn bị render đến lúc render rất ngắn, đồng thời lúc này chưa có DOM nào để tương tác. Nên việc cập nhật giá trị ở đây sẽ không hiển thị kết quả như ta mong muốn.

* Render()

Hàm render() return duy nhất một React element, nên khi cần trả về nhiều elements thì ta phải gom nhóm chúng lại thành một tag khác ví dụ như <form>, <div>… hoặc dùng thẻ rỗng <></> cho trường hợp bạn không muốn render thêm một thẻ khác.

* ComponentDidMount()

Hàm này được gọi sau hàm render(), nghĩa là sau khi các element được render thành công lần đầu tiên. Tức là lúc này các element của component đã được gắn vào cây DOM, và đây là nơi được lựa chọn tốt nhất để chúng ta có thể xử lý các side effect như gọi API, thay đổi state, props… hoặc thiết lập bất kì các subscriptions

1. Giai đoạn Updating

Đây là giai đoạn thứ 3 sau khi component đã render thành công lần đầu tiên. Trong giai đoạn này, dữ liệu của state và props sẽ được cập nhật để đáp ứng với các events theo yêu cầu của người dùng. Điều này dẫn đến việc re-render ở component.

Tương ứng với nó ta có 4 phương thức chính:

* componentWillReceiveProps()

Hàm này sẽ khởi chạy khi một component nhận một props mới từ component cha, hàm này có nhận một argument là nextProps.

* shouldComponentUpdate()

Hàm này giúp tăng hiệu năng của React, để React biết được việc render có bị ảnh hưởng bởi props và state hiện tại hay không. Nếu nó return false thì các phương thức phía sau nó như componentWillUpdate(), render(), componentDidUpdate() sẽ không được thực hiện. Default hàm này return true để chạy được các hàm phía sau nó.

* componentWillUpdate()

Hàm này được gọi trước khi render khi component nhận state hoặc props mới. Hàm này làm công tác chuẩn bị trước khi việc update lại UI xảy ra, và nó được sử dụng cho lần render thứ 2 trở đi. Cũng tương tự hàm componentWillMount(), nó ít được tương tác trong application, vì việc setState hầu hết được gọi trong hàm componentWillReceiveProps()

* componentDidUpdate()

Hàm này được gọi ngay sau khi đã re-render thành công (render lần thứ 2 trở đi, để áp dụng cho lần render đầu tiên thì sử dụng hàm componentDidMount()). Hàm này được sử dụng để xử lý DOM khi DOM đã được cập nhật. Hàm này nhận 3 argument là prevProps, prevState, snapshot (khi có getSnapshotBeforeUpdate()).

1. Giai đoạn Unmounting

Đây là bước cuối cùng cũng như kết thúc một của lifecycle trong một component. Khi tất các tác vụ hoàn thành và ta cần tiến hành unmount DOM (component bị remove khỏi cây DOM). Gia đoạn này chỉ có 1 phương thức là componentWillUnmount()

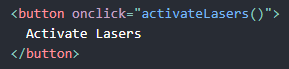
Không nên gọi setState() ở hàm này bởi vì component sẽ không bao giờ re-render. Một khi component đã bị remove khỏi DOM thì sẽ không được render lại lần nào nữa. Hàm này thường dùng để clear các thao tác không cần thiết như xử lý ngưng bộ hẹn giờ, ngắt kết nối mạng, hoặc xóa các subscriptions được tạo trước đó.

1. **Handling Events**

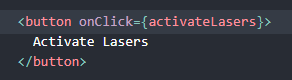
Việc bắt sự kiện của những element React rất giống với những element DOM. Có một số khác biệt về cú pháp như:

* Những sự kiện của React được đặt tên theo dạng camelCase, thay vì lowercase.
* Với JSX, bạn có thể sử dụng “hàm” (function) để bắt sự kiện thay vì phải truyền vào một chuỗi.

Ví dụ trong HTML:



trong React



Trong React, chúng ta không thể “return false” để chặn những hành vi mặc định của trình duyệt mà phải gọi preventDefault trực tiếp.

Khi sử dụng React, ta không cần gọi “addEventListener” để element DOM lắng nghe khi nó được khởi tạo. Thay vào đó, chỉ cần cung cấp listener khi element đó được render lần đầu tiên.

Khi ta định nghĩa một class component, một pattern phổ biến là sử dụng các phương thức của class để bắt sự kiện.

Khi sử dụng funtion thường để trong React, chúng ta bắt buộc phải binding nó. Nếu không thì giá trị this sẽ là “undefined” và ta không thể gọi function cần để thực thi.

1. **Router**

Thư viện ReactJs giúp chúng ta tạo ra các component và từ các component đó chúng ta có thể xây dựng thành các view giao diện, bản thân ReactJs không hỗ trợ Router và có một thư viện hỗ trợ công việc này đó là: react-router.

React-Router là một thư viện định tuyến (routing) tiêu chuẩn trong React. Nó giữ cho giao diện của ứng dụng đồng bộ với URL trên trình duyệt. React-Router cho phép bạn định tuyến "luồng dữ liệu" (data flow) trong ứng dụng của bạn một cách rõ ràng. Nó tương đương với sự khẳng định, nếu bạn có URL này, nó sẽ tương đương với Route này, và giao diện tương ứng.

Cách cài đặt:



Các thành phần trong React Router:

* **BrowserRouter vs HashRouter**
* BrowserRouter: Được sử dụng phổ biến hơn, nó sử dụng History API có trong HTML5 để theo dõi lịch sử bộ định tuyến của bạn.
* HashRouter: Sử dụng hash của URL (window.location.hash) để ghi nhớ mọi thứ.

****

* **Route**

Định nghĩa một ánh xạ (mapping) giữa một URL và một Component. Điều đó có nghĩa là khi người dùng truy cập theo một URL trên trình duyệt, một Component tương ứng sẽ được render trên giao diện.

****

Trong đó:

* **path:** Là đường dẫn trên URL.
* **exact:** Gúp cho route này này chỉ hoạt động nếu URL trên trình duyệt phù hợp tuyệt đối với giá trị của thuộc tính path của nó.
* **component:** Là component sẽ đươc load ra tương ứng với Route đó.
* **Link**

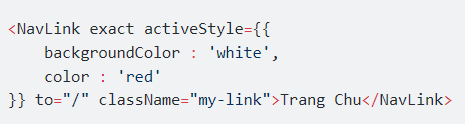
Trong HTML thì cặp thẻ để chuyển hướng đó là thẻ <a></a> thì trong react chúng ta sẽ dử dụng cặp thẻ <Link></Link> được import từ React-Router.



Trong đó:

* **to**: Giống như thuộc tính href trong thẻ a.
* **Nav Link**

NavLink thì rất giống với Link về cách sử dụng, nhưng **NavLink** tốt hơn vì nó hỗ trợ thêm một số thuộc tính như là **activeClassName** và **activeStyle** 2 thuộc tính này giúp cho khi mà nó trùng khớp thì nó sẽ được active lên và chúng ta có thể style cho nó.

****

* **Custom Link**

Ở trên ta có thẻ NavLink giúp chúng ta có thêm một thuộc tính nhưng giả sử khi bạn không muốn activeClassName hoặc activeStyle tại thẻ NavLink mà nó lại nằm ở một thẻ bao nó ví dụ như thẻ div hay thẻ li thì sao? sau đây mình sẽ custom lại để có thể sử dụng các class hoặc style ở thẻ bao ngoài của nó.

* **Đối tượng Match**

Khi bạn muốn lấy một số thông tin ở trên URL thì bạn có thể dùng đối tượng match để lấy dữ liệu về. Tại cấu hình Router ta chỉ cần truyền thêm đối tượng match vào component mà cần sử dụng đối tượng match.

* **Đối tượng prompt - Xác nhận trước khi chuyển trang**

Giả sử khi bạn đang nhập liệu ở form nào đó mà không may click nút back hay chuyển trang thì thôi xong dữ liệu bạn nhập sẽ mất hết để khác phục điều đó ta có đối tượng prompt nó sẽ giúp chúng ta trước khi back hay chuyển trang nó sẽ xác nhận xem là chúng ta có chắc chắn muốn back hay chuyển trang không!

Khi muốn sử dụng đối tượng prompt thì chúng ta chỉ cần import nó từ react-router:



* **Redirect**

Chức năng dùng để chuyển trang.

Có thể truy xuất thông tin trang trước đó thông qua đối tượng location. Để sử dụng Redirect ta chỉ cần import nó từ react-router.



Khi bạn muốn sử dụng location thì tại cấu hình Router ta chỉ cần truyền thêm đối tượng location vào component mà cần sử dụng đối tượng location.

1. **Context API**

Context cung cấp phương pháp truyền data xuyên suốt component tree mà không cần phải truyền props một cách thủ công qua từng level.

Việc truyền dữ liệu từ component cha xuống component con thông qua props là tương đối dài dòng và khó kiểm sóat so với việc sử dụng Context API. Bằng việc sử dụng Context API, chúng ta không còn cần phải truyền các dữ liệu muốn chia sẻ với nhau thông qua việc dùng props.

* Khi nào dùng Context

Những thứ thuộc về context là dữ liệu được coi là global như thông tin người dùng, hay thông tin giỏ hàng... Vậy các lý do khác nhau cho việc sử dụng context:

* Dữ liêụ là cần thiết ở nhiều nơi: dữ liệu cần được sử dụng bởi nhiều component như chủ đề, người dùng, giỏ hàng...
* Truyền dữ liệu qua nhiều component: sử dụng context trong trường hợp này là tốt hơn khi bạn muốn chuyển 1 giá trị props thông qua nhiều component.
* Khối xây dựng và API

Context API bao gồm một số khối xây dựng quan trọng:

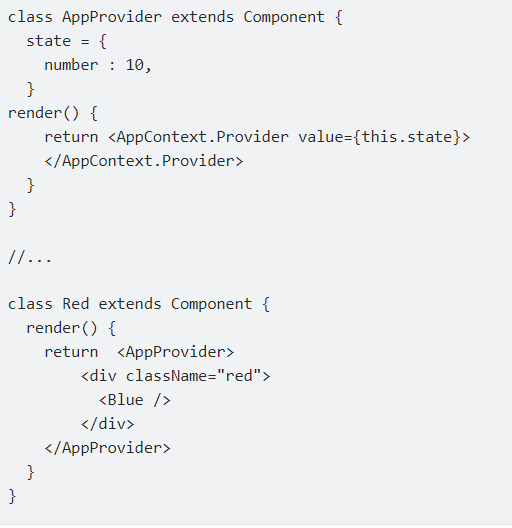
* **context**: đối tượng context là một đối tượng lưu giữ giá trị context hiện tại và có thể được đăng ký.
* **provider**: là một componet của React cung cấp giá trị, nó lấy từ đối tượng context.
* **consumer**: là một component có thể sử dụng giá trị của provider và có thể hiển thị giá trị.

Ví dụ về các bước để sử dụng Context API

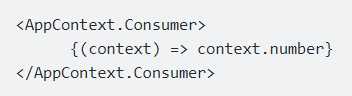
* Chúng ta có thể khởi tạo một **Context Provider** thông qua việc sử dụng hàm React.createContext.



* AppContext sẽ được sử dụng để tạo ra một context provider component. Provider này sẽ chứa dữ liệu chúng ta cần trong state của nó và nó sẽ gói toàn bộ nội dung của component Red:

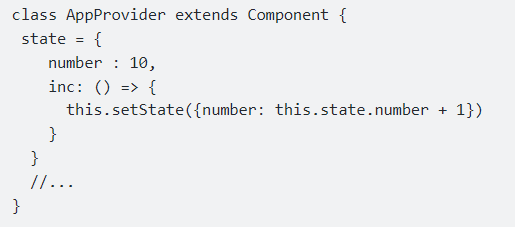


* Sau khi đã setup xong provider, chúng ta có thể truy xuất data bằng cách sử dụng **Context Consumer**.

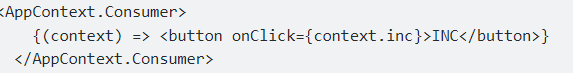


* Các để thay đổi data trong Context API

Điều chúng ta cần làm là định nghĩa một hàm trong state của AppProvider, và xử lý các thay đổi tới dữ liệu trong state.



Sau khi định nghĩa xong hàm, chúng ta có thể sử dụng nó thông qua AppContext.Consumer và gọi nó trong event onClick:



[**Toàn bộ đoạn code ở đây**](https://codesandbox.io/s/contextapi-6mg9f5?file=/src/App.js)

1. **Hooks**

Hooks chính thức được giới thiệu trong phiên bản React 16.8. Nó cho phép chúng ta sử dụng **state** và các tính năng khác của React mà không phải dùng đến **Class.**

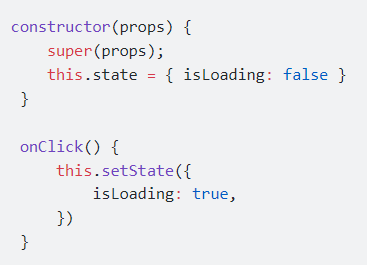
Điều này có nghĩa là từ phiên bản 16.8 trở đi, chúng ta đã có thể sử dụng **state** trong **stateless (functional) component**, việc mà từ trước tới nay ta bắt buộc phải khai báo **Class**. Có thể thấy, các nhà phát triển React họ đang muốn hướng đến 1 tương lai **Functional Programming** thay vì sử dụng những **Class**  Cộng với việc không sử dụng Class kế thừa từ React Component nữa nên giờ đây kích thước bundle sẽ được giảm đáng kể bởi code sử dụng Hooks.

**Basic Hooks**

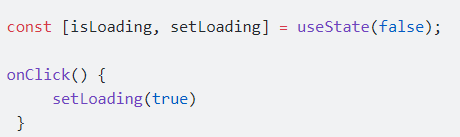
* **UseState**

Hàm này nhận đầu vào là giá trị khởi tạo của 1 state và trả ra 1 mảng gồm có 2 phần tử, phần tử đầu tiên là state hiện tại, phần tử thứ 2 là 1 function dùng để update state (giống như hàm setState cũ vậy).

Ngày trước dùng Class thì viết như này:

****

Còn bây giờ với hooks ta có thể viết ngắn gọn như sau:



Khi muốn update state cho isLoading là true thì chỉ cần gọi đến hàm setLoading(true) là Ok, rất đơn giản và gọn nhẹ phải không nào  Nếu như bạn đang làm việc với React-Redux để quản lý State thì mình khuyên bạn chỉ nên sử dụng useState để quản lý các UI State (là những state có giá trị boolean nhằm mục đích render ra UI) để tránh việc conflict với cả Redux State và maintain sau này.

[**Ví dụ về sử dụng UseState**](https://codesandbox.io/s/todolist-usestate-coh4o9?file=/src/App.js)

* **UseEffect**

Mục đích ***useEffect*** để quản lý vòng đời của của một component và nó phục vụ chúng ta sử dụng trong function component thay vì các lifecycle như trước đây trong class component.

Chúng ta đã nhắc đến **useEffect** là một nơi quản lý các **side-effect** bên trong một React Components.

Thường được sử dụng cho các mục đích như:

* Gọi API để lấy các thông tin bổ sung cho Components
* Thay đổi tiêu đề cho trang
* Cập nhật các state của components.
* Và vô số các hành động side-effect khác.

**useEffect** được kích hoạt sau khi quá trình **render** của React component hoàn tất. Nó sẽ được gọi và thực hiện tính toán các hành động bên trong nó trong một **callback**.

Ví dụ cơ bản về thay đổi tiêu đề với useEffect:



React.useEffect chấp nhận 2 đối số



* **Callback**: sẽ được gọi trong **useEffect** sau khi **return** thực thi nhiệm vụ kết xuất giao diện của nó.
* **Dependencies**: Là một mảng chứa các đối số mà **useEffect** sẽ phụ thuộc vào đó để thực thi. Trong ví dụ trên là một mảng trống, nó đồng nghĩa với việc chỉ thực hiện một lần duy nhất sau khi component render.

Các dependencies trong các trường hợp:

1. Không cung cấp

Trong trường hợp bạn không cung cấp bất kỳ đôi số nào. **useEffect** sẽ được gọi thực thi các tính toán bên trong nó mỗi khi thành phần render.

1. Một mảng rỗng []

Khi bạn truyền một mảng trống vào, nó sẽ chỉ thực thi một lần duy nhất sau khi thành phần đó render lần đầu tiên, cách hoạt động tương tự như componentDidMount của Class Component

1. Khi truyền các Props, State

Khi **dependencies** là các **props, state** bên trong một mảng [props1, props2,.. stateA]. **React useEffect** sẽ dựa vào giá trị props, state. Trong lần render tiếp theo, nó sẽ kiểm tra giá trị của props, state mới với giá trị props, state trước đó. Nếu khác nhau sẽ thực hiện useEffect callback sẽ được gọi. Ngược lại thì không có gì xảy ra. Cơ chế này tương tự như bạn sử dụng Life Cycle componentDidUpdate và shouldComponentUpdate

của Class Component.

[Ví dụ về UseEffect](https://codesandbox.io/s/useeffect-tvjtwr?file=/src/App.js)

**Additional Hooks**

* **useReducer**

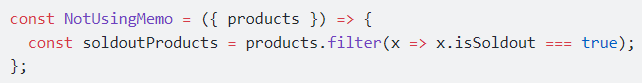
Thực tế khi sử dụng useState thì nó sẽ trả về 1 phiên bản đơn giản của useReducer, vậy nên chúng ta có thể coi useReducer như một phiên bản nâng cao hơn dùng để thay thế cho việc sử dụng useState. Nếu đã làm việc với React-Redux thì chắc hẳn bạn sẽ dễ dàng nhận ra flow quen thuộc này phải không nào. Giống như reducer trong Redux thì useReducer cũng nhận vào một reducer dạng (state, action) và trả ra một newState. Khi sử dụng chúng ta sẽ nhận được một cặp bao gồm current state và dispatch function.

[Ví dụ về useReducer](https://codesandbox.io/s/usereducer-tm1k1i?file=/src/App.js)

* **useMemo**

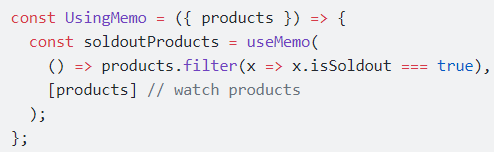
useMemo giúp ta kiểm soát việc được render dư thừa của các component con, nó khá giống với hàm **shouldComponentUpdate** trong LifeCycle. Bằng cách truyền vào 1 tham số thứ 2 thì chỉ khi tham số này thay đổi thì thằng useMemo mới được thực thi.

* Không sử dụng useMemo



soldoutProduct sẽ luôn được thực thi mỗi khi NotUsingMemo được re-render

* Có sử dụng useMemo



soldoutProducts sẽ chỉ thực thi khi props products thay đổi

* **useCallback**

**useCallback** được sử dụng để tối ưu quá trình render của **React functional components**. Nó sẽ rất hữu ích đối với trường hợp một thành phần (component) liên tục được hiển thị lại không cần thiết trong quá trình xử lý sự kiện người dùng và có hành vi chức năng phức tạp.

* **useRef**

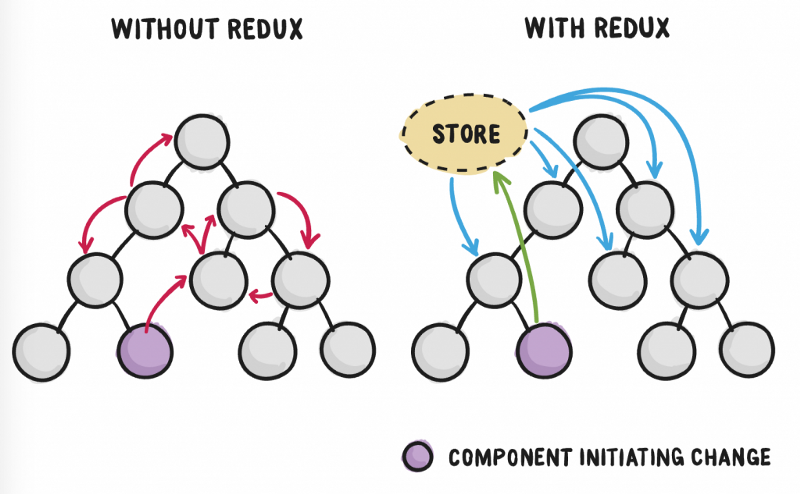
useRef trả về một đối tượng ref có thể thay đổi nơi mà thuộc tính .current được khởi tạo và thêm vào giá trị của (initialValue). Object trả về sẽ kiên định cho cả vòng đời của component.

Nhớ rằng **useRef** không thông báo cho bạn khi nội dung của nó thay đổi. Thay đổi thuộc tính .current sẽ không re-render. Nếu bạn muốn chạy code khi React gán hoặc tách một ref vào DOM node, bạn có thể muốn sử dụng một [callback ref](https://vi.reactjs.org/docs/hooks-faq.html#how-can-i-measure-a-dom-node) để thay thế.

1. **Redux**

**Redux là một thư viện để quản lý và cập nhật trạng thái ứng dụng, sử dụng các sự kiện được gọi là "actions".**Nó phục vụ như một cửa hàng tập trung cho trạng thái cần được sử dụng trên toàn bộ ứng dụng của bạn, với các quy tắc đảm bảo rằng trạng thái chỉ có thể được cập nhật theo cách có thể dự đoán được.

Nó giúp bạn viết các ứng dụng hoạt động một cách nhất quán, chạy trong các môi trường khác nhau (client, server, and native) và dễ dàng để test. **Redux** ra đời lấy cảm hứng từ tư tưởng của ngôn ngữ **Elm** và kiến trúc **Flux**của Facebook. Do vậy Redux thường dùng kết hợp với React. Tuy nhiên hoàn toàn có thể sử dụng với các framework khác như Angular, Angular2, Backbone, Falcor, Deku.



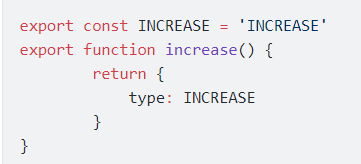
* **Khi nào nên sử dụng Redux**
* Bạn có một lượng lớn trạng thái ứng dụng cần thiết ở nhiều nơi trong ứng dụng
* Trạng thái ứng dụng được cập nhật thường xuyên theo thời gian
* Logic để cập nhật trạng thái đó có thể phức tạp
* Ứng dụng có cơ sở mã kích thước trung bình hoặc lớn và có thể được nhiều người sử dụng
* **Nguyên lý**

Redux được xây dựng dựa trên 3 nguyên lý:

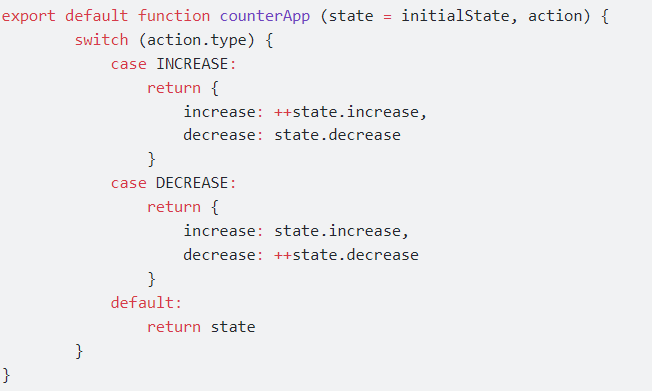
* Nguồn dữ liệu tin cậy duy nhất: State của toàn bộ ứng được chứa trong một object tree nằm trong Store duy nhất
* Trạng thái chỉ được phép đọc: Cách duy nhất để thay đổi State của ứng dụng là phát một Action (là 1 object mô tả những gì xảy ra)
* Thay đổi chỉ bằng hàm thuần túy: Để chỉ ra cách mà State được biến đổi bởi Action chúng ta dùng các pure function gọi là Reducer
* **Cấu trúc**

Về cơ bản Redux có 4 thành phần như sau:

* **Action:** Là nơi mang các thông tin dùng để gửi từ ứng dụng đến Store. Các thông tin này là 1 object mô tả những gì đã xảy ra.

****

* Reducer: Là nơi xác định State thay đổi như thế nào.



* Store: Là nơi quản lý State, cho phép truy cập State qua getState(), update State qua dispatch(action), đăng kí listener qua subscribe(listener).



* View: Hiển thị dữ liệu được cung cấp bởi Store