**Bài 1: Tổng quan về lập trình front end với ReactJS**

* 1. Giới thiệu

React.js là một thư viện Javascript đang nổi lên trong những năm gần đây với xu hướng Single Page Application. Trong khi những framework khác cố gắng hướng đến một mô hình MVC hoàn thiện thì React nổi bật với sự đơn giản và dễ dàng phối hợp với những thư viện Javascript khác. Nếu như AngularJS là một Framework cho phép nhúng code javasscript trong code html thông qua các attribute như ng-model, ng-repeat...thì với react là một library cho phép nhúng code html trong code javascript nhờ vào JSX, bạn có thể dễ dàng lồng các đoạn HTML vào trong JS.Tích hợp giữa javascript và HTML vào trong JSX làm cho các component dễ hiểu hơn

* 1. Giới thiệu về ReactJS

ReactJS được phát triển bởi Facebook và được giới thiệu lần đầu tiên vào năm 2011. Ban đầu, ReactJS được phát triển để xây dựng giao diện người dùng trên trang web Facebook, nhằm cải thiện tốc độ và hiệu suất của ứng dụng web.

Tuy nhiên, ReactJS không được công bố cho cộng đồng phát triển cho đến năm 2013, khi Facebook công bố mã nguồn mở của nó và giới thiệu cho cộng đồng lập trình viên. Từ đó, ReactJS nhanh chóng trở thành một trong những thư viện phổ biến nhất để phát triển các ứng dụng web động.

Sau đó, vào năm 2015, Facebook giới thiệu phiên bản React Native, một framework phát triển ứng dụng di động sử dụng ReactJS. React Native cho phép các nhà phát triển xây dựng các ứng dụng di động cho cả iOS và Android sử dụng cùng một mã nguồn, tương tự như ReactJS trên web.

Hiện nay, ReactJS đã trở thành một trong những thư viện phát triển web phổ biến nhất, được sử dụng rộng rãi bởi các công ty lớn và nhỏ trên toàn thế giới. Facebook cũng tiếp tục đầu tư phát triển và nâng cấp ReactJS để đáp ứng nhu cầu của cộng đồng phát triển.

1.3. Những lợi ích tuyệt vời mà ReactJS mang lại cho lập trình viên

ReactJS mang đến nhiều lợi ích cho việc phát triển ứng dụng web, bao gồm:

* Hiệu suất cao: ReactJS sử dụng Virtual DOM để tối ưu hóa hiệu suất của ứng dụng. Virtual DOM cho phép ReactJS cập nhật các thay đổi trên trang web một cách nhanh chóng và hiệu quả hơn so với cách truyền thống, giúp tăng tốc độ và hiệu suất của ứng dụng.
* Tái sử dụng: ReactJS cho phép tái sử dụng các thành phần UI, giúp giảm thiểu thời gian và chi phí phát triển. Các thành phần UI có thể được sử dụng lại trong nhiều phần khác nhau của ứng dụng, giúp tăng tính linh hoạt và khả năng mở rộng của ứng dụng.
* Dễ dàng quản lý trạng thái: ReactJS giúp quản lý trạng thái của ứng dụng một cách dễ dàng. Sử dụng State và Props, ReactJS cho phép các nhà phát triển quản lý trạng thái của các thành phần UI một cách chính xác và dễ dàng.
* Hỗ trợ tốt cho SEO: ReactJS cho phép các nhà phát triển xây dựng ứng dụng web với khả năng tương thích tốt với SEO. Với sự hỗ trợ của các thư viện như React Helmet, ReactJS cho phép các nhà phát triển tùy chỉnh và quản lý các phần tử meta và title cho từng trang web.
* Hỗ trợ đa nền tảng: ReactJS không chỉ được sử dụng để phát triển các ứng dụng web, mà còn được sử dụng để phát triển các ứng dụng di động với React Native. Sử dụng React Native, các nhà phát triển có thể xây dựng ứng dụng di động cho cả iOS và Android sử dụng cùng một mã nguồn.

1.4. Các tính năng nổi bật của ReactJS

ReactJS có nhiều tính năng hữu ích cho việc phát triển ứng dụng web, bao gồm:

* Components: ReactJS cho phép phát triển ứng dụng web theo mô hình component. Các component là các phần tử UI độc lập có thể được tái sử dụng trong nhiều phần khác nhau của ứng dụng.
* Virtual DOM: ReactJS sử dụng Virtual DOM để tối ưu hóa hiệu suất của ứng dụng. Virtual DOM là một bản sao của DOM được lưu trữ trong bộ nhớ và được cập nhật một cách nhanh chóng khi có thay đổi, giúp tăng tốc độ và hiệu suất của ứng dụng.
* JSX: JSX là một ngôn ngữ lập trình phân biệt được sử dụng trong ReactJS để mô tả các thành phần UI. JSX kết hợp HTML và JavaScript, giúp cho việc viết mã dễ hiểu và dễ bảo trì hơn.
* State và Props: ReactJS cho phép quản lý trạng thái của các thành phần UI thông qua State và Props. State là trạng thái của một thành phần được quản lý bởi nó chính, trong khi Props là các giá trị được truyền vào từ bên ngoài để tùy chỉnh hoặc điều khiển hành vi của một thành phần.
* Hỗ trợ tốt cho SEO: ReactJS hỗ trợ tốt cho việc tối ưu hóa SEO. Với các thư viện như React Helmet, các nhà phát triển có thể quản lý các phần tử meta và title cho từng trang web, giúp tăng khả năng tìm kiếm và tăng cường trải nghiệm người dùng.
* Hỗ trợ đa nền tảng: ReactJS không chỉ được sử dụng để phát triển ứng dụng web, mà còn được sử dụng để phát triển ứng dụng di động với React Native. Sử dụng React Native, các nhà phát triển có thể xây dựng ứng dụng di động cho cả iOS và Android sử dụng cùng một mã nguồn.
* Redux: Redux là một thư viện quản lý trạng thái cho các ứng dụng ReactJS. Nó giúp quản lý trạng thái của ứng dụng một cách chính xác và dễ dàng, đồng thời giúp tăng tính linh hoạt và khả năng mở rộng của ứng dụng.

Đọc thêm: https://200lab.io/blog/reactjs-la-gi/

1.5. Cách sử dụng ReactJS trong phát triển web

Để sử dụng ReactJS trong phát triển web, bạn có thể làm theo các bước sau:

Bước 1 - Cài đặt Node.js và npm: ReactJS được xây dựng trên nền tảng Node.js, do đó bạn cần cài đặt Node.js và npm để phát triển ứng dụng ReactJS.

Bước 2 - Tạo một ứng dụng React: Bạn có thể tạo một ứng dụng React bằng cách sử dụng lệnh "create-react-app" trong Command Prompt hoặc Terminal.

Bước 3 - Tạo các component: Tạo các component để xây dựng giao diện người dùng cho ứng dụng của bạn. Bạn có thể tạo component bằng cách sử dụng class hoặc hàm.

Bước 4 - Xây dựng giao diện người dùng: Sử dụng JSX để xây dựng giao diện người dùng cho ứng dụng của bạn. JSX là một ngôn ngữ phân biệt được sử dụng trong ReactJS để mô tả các thành phần UI.

Bước 5 - Quản lý trạng thái: Sử dụng State và Props để quản lý trạng thái của các thành phần UI. State là trạng thái của một thành phần được quản lý bởi nó chính, trong khi Props là các giá trị được truyền vào từ bên ngoài để tùy chỉnh hoặc điều khiển hành vi của một thành phần.

Bước 6 - Kết nối với API: Sử dụng thư viện như Axios để kết nối với API và lấy dữ liệu từ server.

Bước 7 - Build và triển khai ứng dụng: Sử dụng lệnh "npm run build" để build ứng dụng của bạn và triển khai nó trên môi trường sản phẩm.

Tóm lại, để sử dụng ReactJS trong phát triển web, bạn cần cài đặt Node.js và npm, tạo các component, xây dựng giao diện người dùng, quản lý trạng thái, kết nối với API, build và triển khai ứng dụng.

1.6. Các dự án sử dụng ReactJS thành công

ReactJS là một trong những công nghệ phát triển web phổ biến nhất hiện nay, được sử dụng trong nhiều dự án thành công của các công ty lớn trên thế giới. Dưới đây là một số dự án thành công sử dụng ReactJS:

* Facebook: ReactJS là một trong những công nghệ chính của Facebook, được sử dụng để xây dựng nhiều sản phẩm của họ như Facebook, Instagram, Messenger...
* Netflix: Netflix sử dụng ReactJS để xây dựng giao diện người dùng cho trang web của họ và cho các ứng dụng di động.
* Airbnb: Airbnb sử dụng ReactJS để xây dựng giao diện người dùng cho trang web của họ và cho các ứng dụng di động.
* Uber: Uber sử dụng ReactJS để xây dựng giao diện người dùng cho ứng dụng di động của họ.
* Dropbox: Dropbox sử dụng ReactJS để xây dựng giao diện người dùng cho trang web và cho các ứng dụng di động của họ.
* Atlassian: Atlassian sử dụng ReactJS để xây dựng giao diện người dùng cho nhiều sản phẩm của họ như Jira, Confluence...
* Tất cả những dự án trên đều cho thấy rằng ReactJS là một công nghệ rất phổ biến và thành công trong việc xây dựng các ứng dụng web và di động.

1.7 Setup môi trường phát triển ứng dụng

Để có thể bắt đầu lập trình React thì bạn cần cài đặt môi trường NodeJS, và cài Yarn là trình quản lý các thư viện trong ứng dụng của bạn, bạn có thể sử dụng NPM hoặc YARN đều được, nhưng mình khuyến khích sử dụng yarn hơn.

a) Cài đặt NodeJS

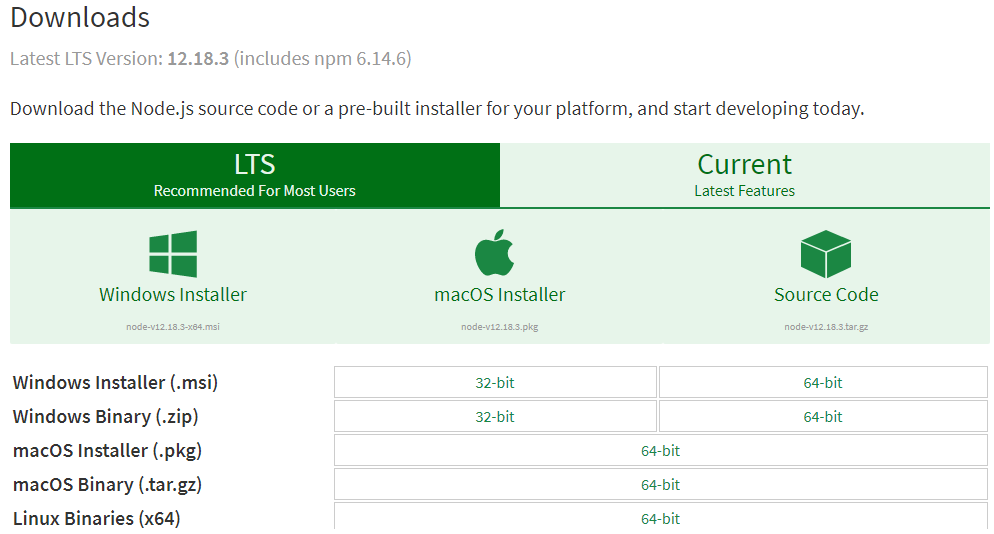
NodeJS cung cấp lệnh cho phép biên dịch TypeScript sang Javascript.

Đầu tiên, bạn cần kiểm tra xem nodejs đã cài đặt trong máy tính chưa. Bạn mở terminal (đối với windows: biểu tượng windows + r ), rồi gõ lệnh:

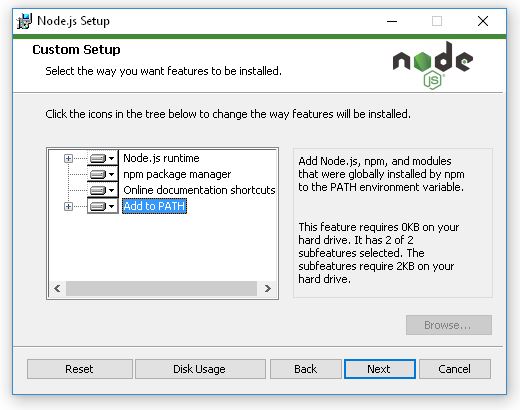


Nếu NodeJS đã tồn tại sẽ hiện thị thông tin version thì chúng ta chuyển sang bước tiếp theo

Nếu chưa tồn tại chúng ta chúng ta vào trang chủ của Node và tải trình Installer về cài đặt tại địa chỉ <https://nodejs.org/en/download/>



Sau đó chúng ta tiến hành cài đặt như bình thường. Lưu ý trong quá trình cài đặt chúng ta nên chọn để trình Installer gán địa chỉ thư mục vào biến môi trường PATH. Nếu không sau này khi code sẽ rất mệt.



Sau khi cài đặt xong chúng ta kiểm tra lại bằng lệnh node -v, nếu thành công máy sẽ hiển thị thông tin version NodeJS được cài đặt

b) Cài đặt Npm

Cài được NodeJS thành công, máy bạn sẽ tự có npm (tức npm đã được cài đặt)

Để kiểm tra phiên bản npm, bạn gõ npm –v

c) Cài Yarn

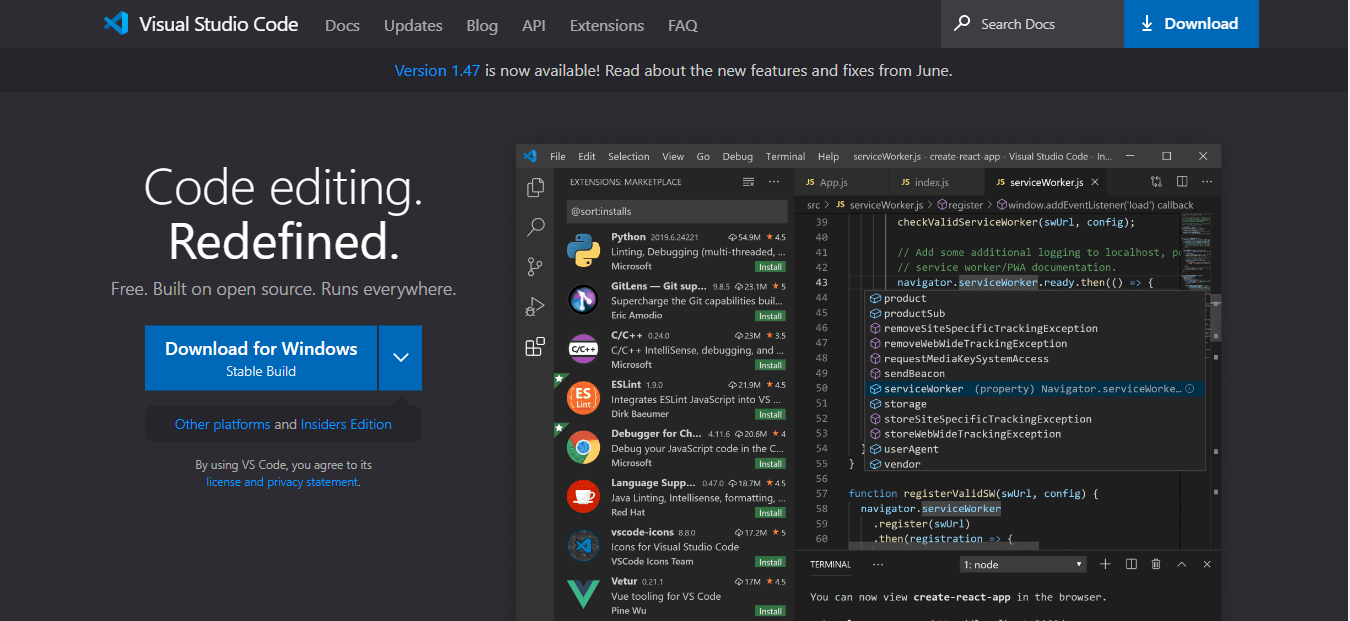
Sau đó bạn cài Yarn bằng lệnh npm install -g yarn sau khi cài xong thì bạn có thể sử dụng yarn thay cho npm rồi.

d) Lựa chọn IDE

IDE thì bạn có thể sử dụng IDE tùy thích. Có thể là WebStorm, Atom, VS Code, Netbean, ... hoặc thậm chí là notepad vẫn được. Tuy nhiên thì tớ khuyến khích bạn nên sử dụng VS Code vì nó nhẹ, dễ sử dụng, hiển thị cũng dễ nhìn.

Để cài đặt VS Code

Bạn truy cập trang web <https://code.visualstudio.com/> để tải về bộ cài đặt của VSCode. Chỉ cần nhấn nút Download thì bạn sẽ tải bộ cài phù hợp với hệ thống của mình. Nếu muốn tải những phiên bản khác hoặc cho hệ điều hành khác, bạn chỉ việc nhấn vào nút bên cạnh ô Download và lựa chọn bộ cài mình cần.



Nhấn vào nút download để tải file cài đặt và thực hiện các bước cài đặt theo hướng dẫn

1.8 Tạo project đầu tiên

Tiếp theo, bạn có thể cài đặt ReactJS bằng cách vào thư mục chứa dự án và mở temrinal và gõ dòng lệnh:

npx create-react-app my-app

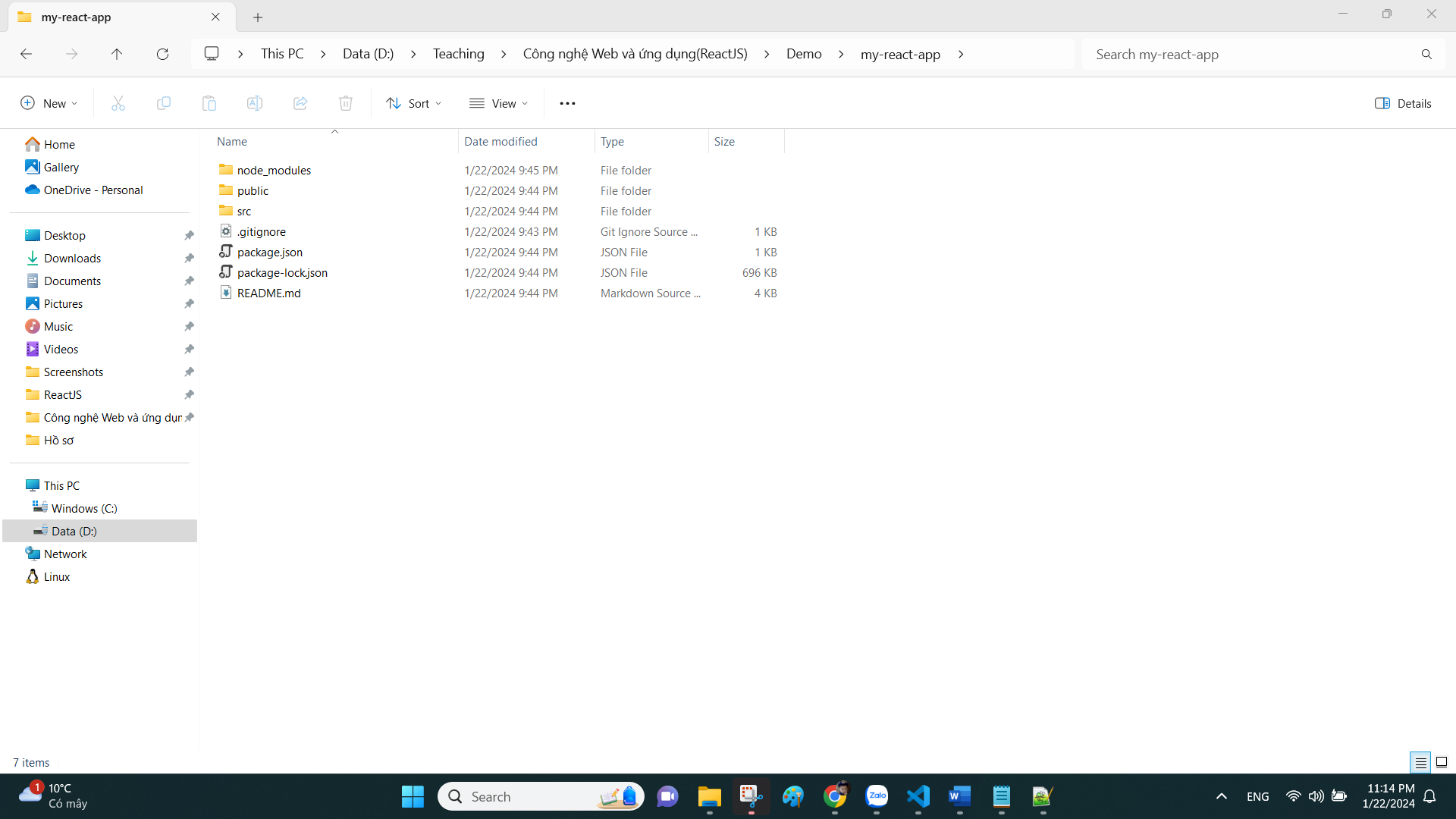
hoặc yarn create-react-app my-app

npx create-react-app ten-du-an --template typescript

Trong đó my-app là tên thư mục chứa dự án của bạn, giả sử mình muốn khởi tạo dự án có tên banhang-app, chúng ta sẽ có:

npx create-react-app banhang-app

Bạn đợi một khoảng thời gian cho quá trình cài đặt hoàn tất, sau khi cài đặt thành công bạn sẽ thấy kết quả như sau:

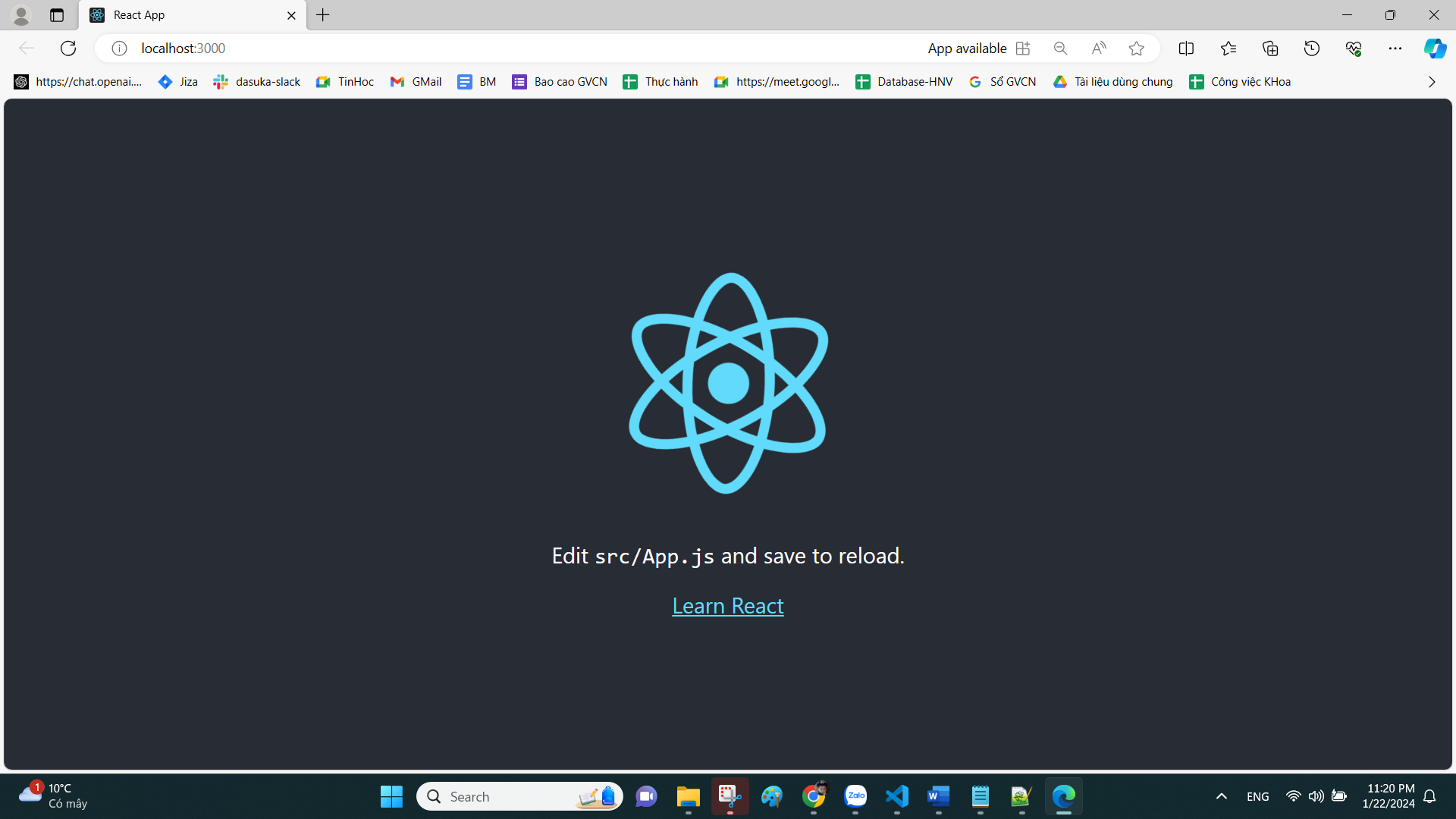


Để chạy chương trình dùng lệnh

npm start

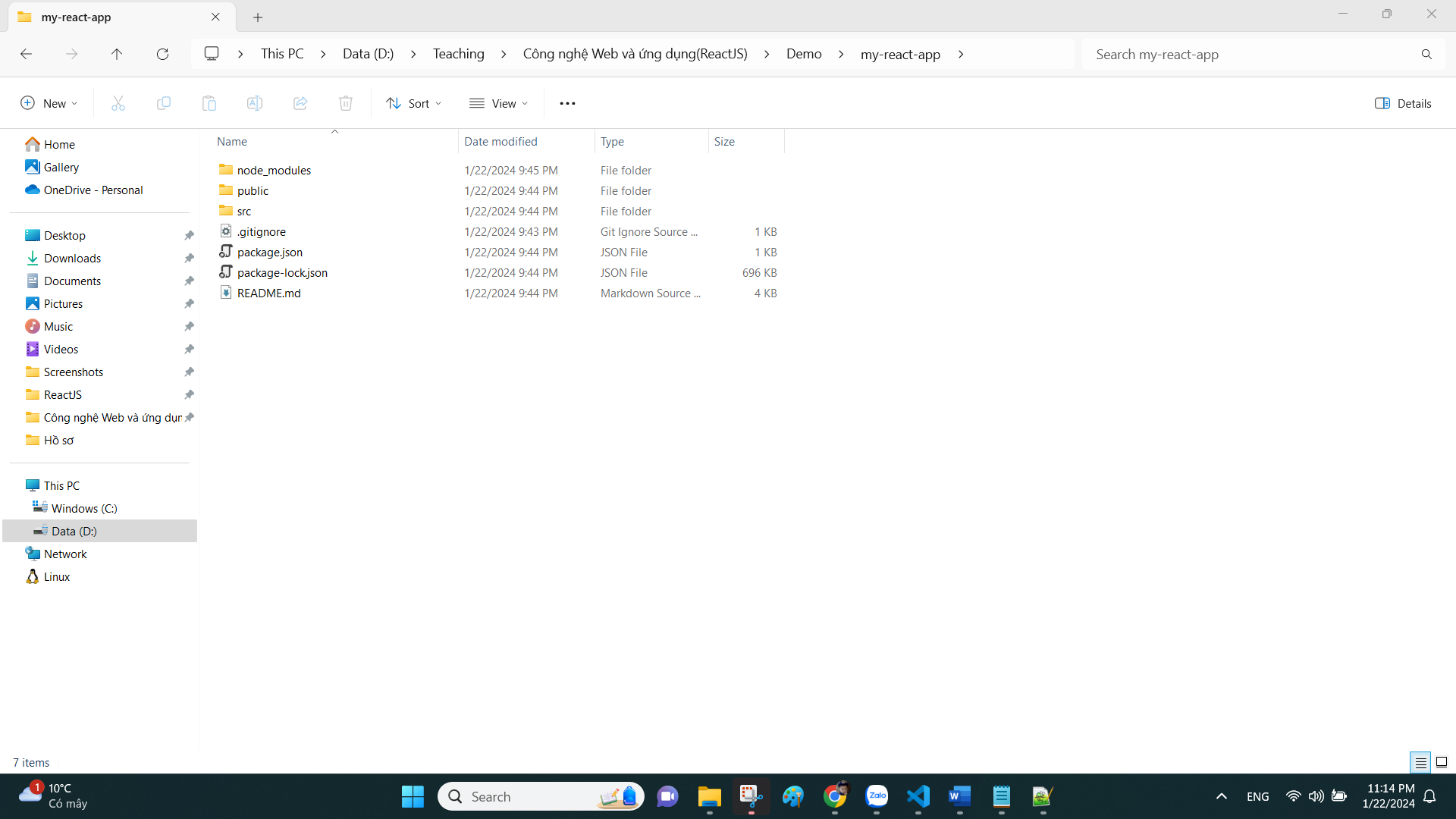
hoặc yarn start

Kết quả hiển thị



1.9 Cấu trúc một ứng dụng ReactJS

Sau khi tạo xong dự án Angular, cấu trúc dự án ReactJS có dạng như sau



* node\_modules: Thư mục này chứa tất cả các module và dependencies của dự án, được quản lý bởi npm (Node Package Manager).
* public: Thư mục này chứa các tệp tin không thay đổi khi dự án được xây dựng (ví dụ: index.html, các hình ảnh, favicon, ...).
* index.html: Là trang HTML chính của ứng dụng React. Nó sẽ chứa một thẻ <div> có id là "root", nơi React sẽ render các component vào.
* src: Thư mục chính chứa mã nguồn của ứng dụng React.
* index.js: Là tập tin đầu tiên được chạy khi ứng dụng bắt đầu. Nó thường sẽ chứa đoạn mã để render một component vào thẻ có id là "root" trong file index.html.
* App.js: Là nơi bạn sẽ xây dựng component chính của ứng dụng. Thông thường, nó sẽ được import và render trong index.js.
* App.css: Chứa các quy tắc CSS liên quan đến component App.js.
* logo.svg: Một ví dụ về hình ảnh được import và sử dụng trong mã nguồn.
* index.css: Chứa các quy tắc CSS toàn cục được áp dụng cho toàn bộ ứng dụng.
* package.json: Tệp tin mô tả thông tin về dự án và các phụ thuộc của nó. Nó chứa tên dự án, phiên bản, các lệnh npm để chạy và xây dựng dự án, và nhiều cấu hình khác.
* package-lock.json: Là một tệp tự động tạo ra bởi npm để đảm bảo rằng mọi người sử dụng dự án có thể cài đặt các phiên bản giống nhau của dependencies.
* README.md: Chứa thông tin mô tả về dự án, cách cài đặt và chạy ứng dụng, và các hướng dẫn khác.
* .gitignore: Tệp này liệt kê các tệp và thư mục mà Git sẽ bỏ qua khi bạn commit và đẩy dự án lên kho lưu trữ Git.
* public/favicon.ico: Một biểu tượng được hiển thị trong thanh địa chỉ trình duyệt và trên màn hình nền của ứng dụng.
* public/manifest.json: Một tệp đặc tả về ứng dụng web, chẳng hạn như tên, mô tả, và các cài đặt khác. Thường được sử dụng khi bạn muốn "cài đặt" ứng dụng vào màn hình chính của điện thoại di động.

1.10. Một số câu lệnh cơ bản của ReactJS Cli

NPM (Node Package Manager), NPX và Yarn đều là các công cụ quản lý gói cho Node.js và JavaScript, được sử dụng phổ biến trong phát triển ứng dụng ReactJS.

*NPM (Node Package Manager):*

* Chức năng: NPM là một trình quản lý gói (package manager) cho Node.js, giúp bạn quản lý các thư viện và công cụ cần thiết cho dự án của mình.
* Sử dụng trong ReactJS: Khi bạn tạo một dự án React bằng Create React App hoặc bất kỳ dự án Node.js nào khác, npm thường được sử dụng để cài đặt và quản lý các gói phụ thuộc.
* Câu lệnh cài đặt gói: npm install <package-name>

*NPX:*

* Chức năng: NPX là một công cụ đi kèm với NPM, được sử dụng để chạy các lệnh npm và các công cụ từ các gói đã được cài đặt mà không cần phải cài đặt chúng một cách toàn cục.
* Sử dụng trong ReactJS: Khi sử dụng Create React App, bạn thường sẽ thấy lệnh npx create-react-app để tạo một ứng dụng React mới mà không cần cài đặt Create React App toàn cục.
* Câu lệnh sử dụng: npx <command> hoặc npx <package-name>

*Yarn:*

* Chức năng: Yarn là một công cụ quản lý gói cũng giống như NPM, được phát triển để giải quyết một số vấn đề về hiệu suất và đồng bộ hóa khi sử dụng NPM.
* Sử dụng trong ReactJS: Yarn có thể được sử dụng làm thay thế hoặc bổ sung cho NPM trong quá trình phát triển ReactJS. Create React App hỗ trợ cả Yarn và NPM.
* Câu lệnh cài đặt gói: yarn add <package-name>

Tóm lại, NPM là trình quản lý gói chính của Node.js, NPX là công cụ giúp chạy lệnh từ các gói đã cài đặt mà không cần phải cài đặt chúng toàn cục, và Yarn là một công cụ quản lý gói tương tự như NPM nhưng với mục tiêu là cải thiện hiệu suất và đồng bộ hóa. Trong ngữ cảnh của ReactJS, bạn có thể sử dụng cả NPM và Yarn, tùy thuộc vào sự ưa thích cá nhân và yêu cầu của dự án.

a) Tạo một ứng dụng React mới:

npx create-react-app my-app

Tạo một dự án React mới với tên là my-app. Bạn có thể thay đổi my-app thành bất kỳ tên nào bạn muốn.

b) Chạy ứng dụng React:

cd my-app

npm start

Di chuyển vào thư mục dự án và chạy ứng dụng React. Trình duyệt của bạn sẽ tự động mở và hiển thị ứng dụng tại http://localhost:3000/.

c) Tạo một component mới:

npx generate-react-cli component MyComponent

Tạo một component mới có tên là MyComponent. Thay đổi MyComponent thành tên bạn muốn.

d) Tạo một trang mới:

npx generate-react-cli page MyPage

Tạo một trang mới có tên là MyPage. Tương tự như trên, bạn có thể thay đổi tên tùy ý.

e) Tạo một reducer mới (nếu bạn sử dụng Redux):

npx generate-react-cli reducer myReducer

Tạo một reducer mới có tên là myReducer. Redux là một thư viện quản lý trạng thái trong ứng dụng React.

f) Xây dựng ứng dụng để triển khai:

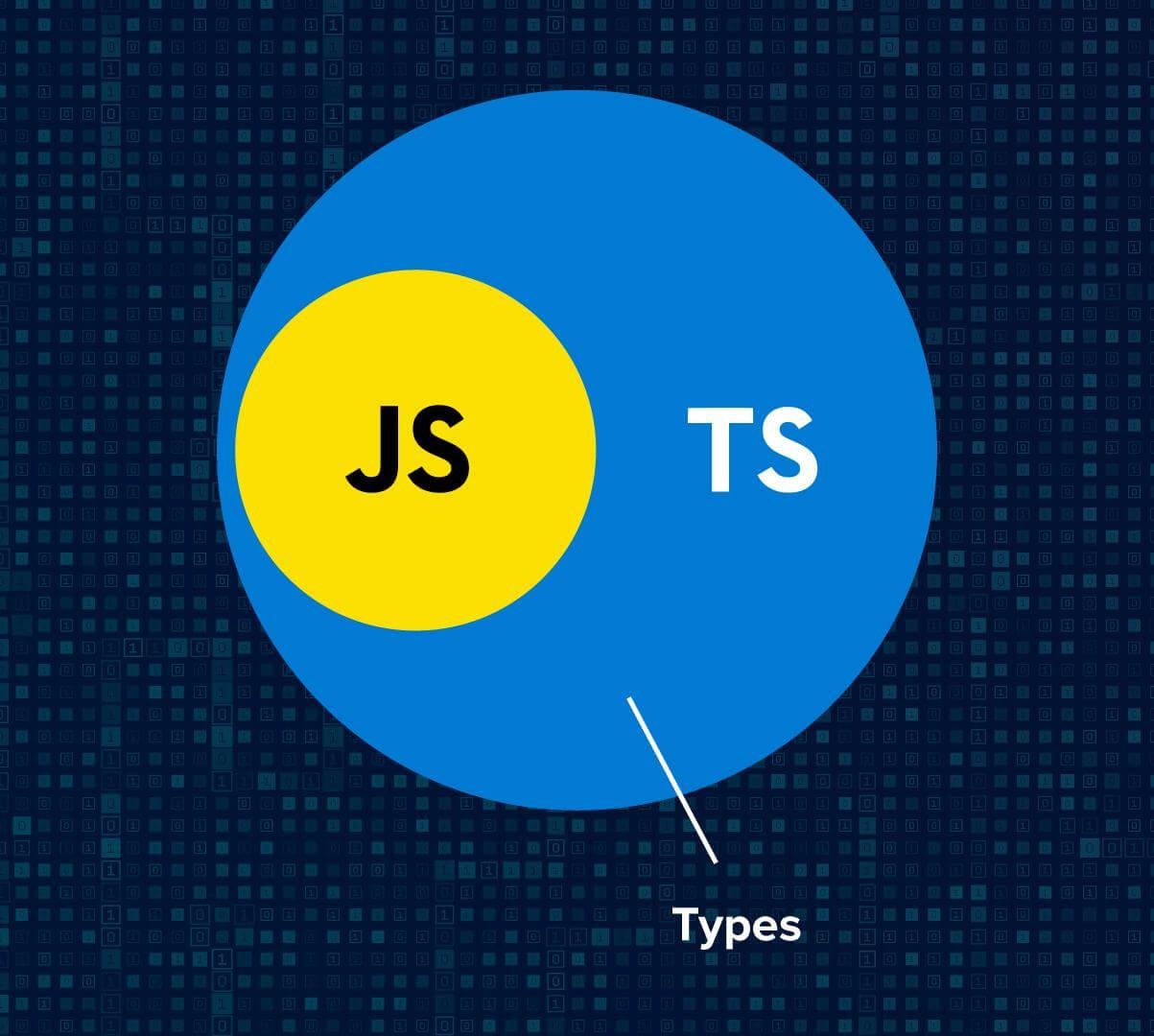
npm run build

Tạo một bản dự án đã được xây dựng (build) trong thư mục build. Bạn có thể triển khai nó lên môi trường sản phẩm.

Bài 2: Cơ bản về ngôn ngữ lập trình TypeScript

2.1 TypeScript là gì?

TypeScript là một dự án mã nguồn mở được phát triển bởi Microsoft, nó có thể được coi là một phiên bản nâng cao của Javascript bởi việc bổ sung tùy chọn kiểu tĩnh và lớp hướng đối tượng mà điều này không có ở Javascript. TypeScript có thể sử dụng để phát triển các ứng dụng chạy ở client-side (*Angular2, React*) và server-side (*NodeJS*)...



TypeScript sử dụng tất cả các tính năng của của ECMAScript 2015 (ES6) như classes, modules. Không dừng lại ở đó nếu như ECMAScript 2017 ra đời thì mình tin chắc rằng TypeScript cũng sẽ nâng cấp phiên bản của mình lên để sử dụng mọi kỹ thuật mới nhất từ ECMAScript. Thực ra TypeScript không phải ra đời đầu tiên mà trước đây cũng có một số thư viện như CoffeScript và Dart được phát triển bởi Google, tuy nhiên điểm yếu là hai thư viện này sư dụng cú pháp mới hoàn toàn, điều này khác hoàn toàn với TypeScript, vì vậy tuy ra đời sau nhưng TypeScript vẫn đang nhận được sự đón nhận từ các lập trình viên.

***Tóm lại:***

TypeScript sử dụng cú pháp của JavaScript và bổ sung thêm các cú pháp mới để hỗ trợ Type (các kiểu dữ liệu).

TypeScript = JavaScript + Type + Một số tính năng được thêm vào

Vì TypeScript code được chuyển đổi thành JavaScript code nên việc tích hợp vào các dự án JavaScript trở nên dễ dàng hơn, được thiết kế chủ yếu cho các dự án quy mô lớn.

Browser chỉ có thể hiểu HTML/CSS và JavaScript thuần, không hiểu TypeScript. Vì thế, cần có một công cụ trung gian đứng ra dịch code và đó là TypeScript Compiler.

Mỗi một dự án bạn viết bằng TypeScript cần phải có compiler giúp chuyển dịch code ra thành JavaScript để browser có thể hiểu được.

2.2 Tại sao nên sử dụng Typescript

Để hiểu tại sao nên sử dụng TypeScript thì có lẽ chúng ta nên tìm hiểu sơ lược về các ưu điểm mà TypeScritpt mang lại.

**Dễ phát triển dự án lớn**: Với việc sử dụng các kỹ thuật mới nhất và lập trình hướng đối tượng nên TypeScript giúp chúng ta phát triển các dự án lớn một cách dễ dàng.

**Nhiều Framework lựa chọn**: Hiện nay các Javascript Framework đã dần khuyến khích nên sử dụng TypeScript để phát triển, ví dụ như Angular và Ionic

**Hô trợ các tính năng của Javascript phiên bản mới nhất**: TypeScript luôn đảm bảo việc sử dụng đầy đủ các kỹ thuật mới nhất của Javascript, ví dụ như version hiện tại là ECMAScript 2015 (ES6).

**Là mã nguồn mở**: TypeScript là một mã nguồn mở nên bạn hoàn toàn có thể sử dụng mà không mất phí, bên cạnh đó còn được cộng đồng hỗ trợ.

**TypeScript là Javscript**: Bản chất của TypeScript là biên dịch tạo ra các đoạn mã javascript nên ban có thê chạy bất kì ở đâu miễn ở đó có hỗ trợ biên dịch Javascript. Ngoài ra bạn có thể sử dụng **trộn lẫn cú pháp của Javascript vào bên trong TypeScript**, điều này giúp các lập trình viên tiếp cận TypeScript dễ dàng hơn.

Như vậy bản chất của TypeScript là một trình biên dịch xuất ra mã Javascript dựa vào cấu trúc riêng của nó. Trong CSS cũng có một thư viện tương tự đó là SASS và LESS.

Như vậy ta có thể coi TypeScript là cha của Javascript bởi kết quả sau khi biên dịch TypeScript là xuất ra các đoạn mã Javascript, để các bạn hiêu rõ hơn mình sẽ lấy một ví dụ như sau:

**Code TypeScript**

*class Customer {*

*Name : string;*

*constructor (firstName: string, lastName: string)*

*{*

*this.Name = firstName + " " + lastName;*

*}*

*GetName()*

*{*

*return "Hello, " + this.Name;*

*}*

*}*

**Biên dịch thành Javascript**

*var Customer = (function () {*

*function Customer(firstName, lastName) {*

*this.Name = firstName + " " + lastName;*

*}*

*Customer.prototype.GetName = function () {*

*return "Hello, " + this.Name;*

*};*

*return Customer;*

*}());*

Với hai đoạn code trên thì rõ ràng nhìn vào mã của TypeScript rất là trong sáng và mạch lạc.

**Tóm lại:**

Các nhà phát triển đã giới thiệu JavaScrip là ngôn ngữ client-side programming. Nhưng khi các lập trình viên sử dụng thì nhận ra rằng, JavaScript có thể sử dụng làm ngôn ngữ server-side programming.

Tuy nhiên, JavaScript code trở nên phức tạp và nặng nề, thậm chí chúng không thể đáp ứng được yêu cầu của ngôn ngữ lập trình hướng đối tượng. Khiến các doanh nghiệp hạn chế sử dụng JavaScript như một công nghệ server-side.

Đó là lý do, TypeScript ra đời để quyết những vấn đề mà JavaScript đang gặp phải. TypeScript là một phần mở rộng của JavaScript, giúp tăng tính kiểm soát và dễ bảo trì cho mã nguồn JavaScript.

***Nên chọn TypeScript và JavaScript khi nào?***

Nếu bạn đang triển khai dự án với nhiều nhóm phát triển, nhiều mã nguồn mở thì TypeScript sẽ là lựa chọn tốt hơn. Vì, TypeScript có khả năng kiểm soát kiểu dữ liệu tại thời điểm biên dịch giúp tránh lỗi và dễ bảo trì hơn. Giúp dự án của bạn có thể hoạt động một cách trơn tru dễ dàng hơn và giảm thời gian sửa các lỗi.

Tuy nhiên, nếu dự án của bạn là một ứng dụng nhỏ hoặc tạo trang web đơn giản, thì JavaScript sẽ là lựa chọn. Vì, JavaScript là một ngôn ngữ lập trình linh hoạt, rộng rãi được sử dụng trên nhiều trình duyệt và hệ điều hành. Với JavaScript, bạn có thể lập trình nhanh chóng và đơn giản mà không cần phải lo lắng về việc kiểm soát kiểu dữ liệu và mã nguồn.

2.3 Các kiểu dữ liệu cơ bản và cách khai báo biến

TypeScript là một ngôn ngữ có “họ hàng” với Javascript. Đây là một bản mở rộng của ECMAScript6 (viết tắt là ES6), ES6 lại là một bản mở rộng của ECMAScript5 (viết tắt là ES5), ES5 chính là Javascript mà chúng ta thường dùng.

2.3.1 Các kiểu dữ liệu cơ bản

Bản chất của Typescript vẫn là Javascript nên các kiểu dữ liệu cơ bản của Javascript thì Typescript đều có ngoài ra Typescript còn có một số kiểu đữ liệu khác như là enum, tuple, any, void ...

Các kiểu dữ liệu cơ bản

Các kiểu dữ liệu thường dùng là string, number, boolean, array, enum, any, void.

**a) Kiểu string**

*Kiểu string đơn giản là một chuỗi (một đoạn text) được bao bọc bởi cặp ký tự ' hoặc "*

Ví dụ 1:

*var string1 : string;*

*string1 = '1001';*

*console.log(string1);*

Ví dụ 2:

*let employeeName:string = "John Smith";*

*let employeeDept:string = "Finance";*

**b) Kiểu number**

*Cũng tương tự như trong Javascript thì trong TypeScript chỉ tồn tại một kiểu Number, còn việc phân chia các kiểu nhỏ hơn như số nguyên, só thực sẽ phụ thuộc vào giá trị mà bạn gán cho nó. TypeScript cũng hỗ trợ kiểu nhị phân và bát phân được giới thiệu trong ECMAScript 2015.*

Ví dụ:

*var decimal: number = 12;*

*var hex: number = 0xf00d;*

*var binary: number = 0b1010;*

*var octal: number = 0o744;*

**c) Kiểu Array**

*Kiểu mảng, khi khai báo các phần tử trong mảng thì chúng ta phải chỉ ra cả kiểu dữ liệu nữa. Để tạo array thì chúng ta có thể dùng cú pháp Array<kiểu>**hoặc <kiểu>[].*

Ví dụ:

***Mảng string***

*var arrString : string[];*

*arrString = ['teo', 'ty', 'tun'];*

*console.log(arrString[0]);*

***Mảng number***

*var arrNumber : number[];*

*arrNumber = [1, 2, 3];*

**d) Kiểu boolean**

*Kiểu boolean có hai giá trị là true hoặc false và cả Javascript lẫn TypeScript đều gọi là boolean.*

Ví dụ:

*var boolean1 : boolean = true;*

*console.log(boolean1);*

**e) Kiểu enum**

*Tương tự như trong C#, Enum là kiểu dữ liệu đặc biệt dùng để tạo một nhóm tên tương ứng với các giá trị là những con số mà ta thiết lập cho nó, cách này sẽ giúp ta dễ dàng nhớ tên hơn.*

Ví dụ:

*enum Color {Red, Green, Blue}*

*var c: Color = Color.Green;*

**f) Kiểu tuple**

*Tuple là kiểu dữ liệu đặc biệt có thể chứa nhiều giá trị với nhiều kiểu dữ liệu con khác nhau. Thực ra Tuple là một mảng nhưng đã xác định được số phần tử và kiểu dữ liệu cho mỗi phần tử đó.*

Ví dụ:

*var x: [string, number];*

*x = ['ahihi', 10];*

*for (let i = 0; i < x.length; i++) {*

*console.log(x[i]);*

*}*

*console.log(x[0]);*

**g) Kiểu any**

*Đây là kiểu dữ liệu thoải mái nhất bởi nó cho phép bạn gán giá trị với kiểu dữ liệu bất kì, điều này giúp giải quyết rắc rối ở một số trường hợp, ví dụ ta cần lấy dữ liệu từ người dùng hoặc một thư viện khác thì ta không biết giá trị trả về sẽ ở kiểu dữ liệu nào nên ta sẽ sử dụng kiểu Any để tránh lỗi. Sau đây là một ví dụ từ trang chủ của nó.*

Ví dụ:

*var something: any = "this is a string";*

*something = 1;*

*something = [1, 2, 3];*

**h) Kiểu void**

*Trong C# thì khi muốn khai báo một hàm không có giá trị trả về thì ta sẽ sử dụng hàm void, tuy ta hay gọi là hàm void nhưng thực ra nó là một kiểu dữ liệu với giá trị là null, trong TypeScript thì có thêm giá trị undefined.*

Ví dụ:

*function showMessage(): void {*

*alert("Success!");*

*}*

*let unusable: void = undefined;*

2.3.2 Cách khai báo biến

*Ngoài từ khóa var ra TypeScript hỗ trợ thêm hai cách khai báo biến đó là sử dụng từ khóa let và const.*

***a) Khai báo biến với từ khoa var***

Ví dụ:

*var domain = 'freetuts.net';*

*var author:string = 'Nguyễn Văn Cường';*

*var series = new Array();*

Sư dụng từ khóa var đồng nghĩa với việc bạn đã khai báo trong phạm vi toàn cầu, nghĩa là bạn có thể sử dụng nó ở mọi nơi từ bên trong hàm cho đến bên ngoài hàm.

Ví dụ:

*var domain = 'utehy.edu.vn';*

*function showDomain()*

*{*

*alert(domain);*

*}*

*// Kết quả: utehy.edu.vn*

*showDomain();*

Nếu bạn khai báo một biến nằm bên trong hàm thì đương nhiên biến đó chỉ mang tính chất là biến cục bộ, nghĩa là chỉ sử dụng được trong hàm đó mà thôi.

Ví dụ:

*function showDomain()*

*{*

*var domain = 'utehy.edu.vn';*

*}*

*// Lỗi vì biến domain là biến cục bộ trong hàm showDomain*

*alert(domain);*

***b) Khai báo biến với từ khoa let***

Từ khóa let được giới thiệu ở phiên bản ECMAScript 6 (ES6). Khi sử dụng từ khóa let để khai báo biến thì biến đó chỉ hoạt động trong phạm vi khối của nó (block-scoped).

Ví dụ:

*let a = 20;*

*let b = 30;*

*let c = a + b;*

*alert(c); // 50*

Nếu sư dụng bên ngoài phạm vi của block-scoped thì sẽ bị lỗi, xem ví dụ.

Ví dụ:

*let domain = utehy.edu.vn';*

*if (domain == ‘utehy.edu.vn')*

*{*

*let author = 'Nguyễn Văn Anh';*

*}*

*alert(author); // Lỗi vì biến author nằm trong phạm vi block-scoped khác*

***c) Khai báo biến với từ khoa const***

Sử dụng từ khóa const để khai báo một biến và biến đó sẽ không thể nào thay đổi giá trị lại, trong các ngôn ngữ lập trình khác thì đây cũng có thể hiểu là khai báo hằng. Khác với các ngôn ngữ khác là khi khai báo hằng thì thường ta chỉ gán được kiểu giá trị là kiểu chuỗi hoặc kiểu số, tuy nhiên trong Javascript nói chung và trong TypeScript nói riêng thì bạn có thể gán cho nó mọi kiểu dữ liệu.

Như ví dụ sau là hoàn toàn hợp lệ vì ta gán giá trị là một object.

*const info = {*

*name : "Nguyen Van Cuong",*

*domain : "Freetuts.net"*

*};*

*console.log(info);*

Nếu bạn cố tình thay đổi giá trị thì sẽ bị báo lỗi.

*// Khai báo const*

*const domain = 'utehy.edu.vn';*

*// Thay đổi giá trị => dòng này sẽ bị lỗi*

*domain = ' fit.utehy.edu.vn ';*

*Giống như let, phạm vi hoạt động của biến const cũng bị hạn chế bởi block-scorped.*

*{*

*// Khai báo const*

*const domain = ' utehy.edu.vn ';*

*console.log(domain);*

*}*

*{*

*console.log(domain); // Bị lỗi vì biến domain nằm ở block-scoped phía trên*

*}*

2.4 Các cấu trúc điều khiển

2.4.1 Lệnh If Else trong TypeScript

Lệnh if dùng để kiểm tra một biểu thức đúng hay sai? Nếu đúng thì thực thi một nhiệm vụ nào đó.

*if (condition)*

*{*

*// thực hiện*

*}*

Trong đó condition là biểu thức muốn kiểm tra, nó có thể là một biểu thức, một giá trị hoặc là một biến.

Ví dụ: Kiểm tra nếu điểm bé hơn 5 thì thông báo thi rớt.

*var point = 4;*

*if (point < 5)*

*{*

*alert('Rớt');*

*}*

Lênh else sẽ được thực hiện nếu như biểu thức ở lệnh if không thỏa điều kiện.

*if (condition){*

*// Do Something*

*}*

*else{*

*// Do Something*

*}*

Ví dụ: Kiểm tra điểm lớn hơn 4 thì thông báo đậu, ngược lại thông báo rớt.

*var point = 2;*

*if (point){*

*alert('Đậu');*

*}*

*else{*

*alert('Rớt');*

*}*

Lệnh else if dùng để kiểm tra thêm một điều kiện và nó sẽ được chạy nếu tất cả các lệnh kiểm tra điều kiện ở trên nó không thỏa.

*if (condition1){*

*}*

*else if (condition2){*

*}*

*else if (condition3){*

*}*

*else{*

*}*

Ví dụ: Kiểm tra các mức điểm của học sinh

*var point = 10;*

*if (point >= 8){*

*alert('Giỏi');*

*}*

*else if (point >= 6){*

*alert('Khá');*

*}*

*else if (point >= 5){*

*alert('Trung bình');*

*}*

*else{*

*alert('Yếu');*

*}*

2.4.2 Lệnh Switch Case trong TypeScript

Lệnh Switch Case có chức năng tương tự như if else, tuy nhiên nó chỉ dùng để kiểm tra biểu thức có giá trị là một chuỗi hoặc một con số.

Cú pháp như sau:

*switch (expression) {*

*case case1:*

*statements1*

*[break;]*

*case case2:*

*statements2*

*[break;]*

*case case3:*

*statements3*

*[break;]*

*default:*

*default statements*

*[break;]*

*}*

Trong đó lệnh break sẽ có nhiệm vụ dừng chương trình switch case.

Ví dụ:

*var book = 'Math';*

*var msg = '';*

*switch (book) {*

*case "English":*

*msg = "Sách có giá là $12.";*

*break;*

*case "Math":*

*msg = "Sách có giá là $22.";*

*break;*

*case "Commerce":*

*msg = "Sách có giá là $12.";*

*break;*

*case "History":*

*msg = "Sách có giá là $125.";*

*break;*

*case "Physics":*

*msg = "Sách có giá là $12.99.";*

*break;*

*default:*

*msg = "Không tìm thấy mệnh giá";*

*}*

*alert(msg);*

Trong ví dụ này vì cuốn sách có tên là Math nên case "Math" sẽ được chạy, lúc này biến msg sẽ có gái trị là "Sách có giá là $22." và tiếp theo lệnh break; sẽ dừng chương trình switch lại.

2.4.3 Vòng lặp trong TypeScript

Vòng lặp rất quan trọng trong các ngôn ngữ lập trình và trong TypeScipt cũng vậy. Khi nói đến vòng lặp thì ta sẽ nhớ đến các vòng lặp thông dung như: vòng lặp for, vòng lặp while vòng lặp do while. Tuy nhiên với Javascript thì chúng ta sẽ có thêm một số vòng lặp khác nữa như vòng lặp for in và vòng lặp for as.

**a) Vòng lặp for in**

Vòng lặp for in ta còn gọi là iterates, tức là sau mỗi lần lặp nó sẽ nhớ là đang lặp tới phần tử nào để lần lặp tiếp theo nó tự động lấy phần tử tiếp theo. Giá trị của mỗi lần lặp là index chứ không phải là value.

Cú pháp:

*for (variable in object)*

*{*

*//block to execute*

*}*

Ví dụ:

*var list = [1,2,5];*

*var t = 0;*

*for (let i in list) {*

*t += list[i];*

*}*

Trong vòng lặp này thì mỗi lần lặp nó sẽ lặp một phần tử với i lần lượt là chỉ số của mảng: 0,1,2.

**b) Vòng lặp for**

Vòng lặp for dùng để lặp những trường hợp ta biết tổng số lần lặp, xem thêm tại bài vòng lặp for trong javascript.

Ví dụ1:

*for (let j = 0; j <= 10; j++)*

*{*

*document.write(j);*

*}*

Ví dụ 2:

*var numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];*

*var sum = 0;*

*for(var i = 0; i < numbers.length; i++)*

*sum += numbers[i];*

*console.log(sum);*

**c) Vòng lặp while và do while**

Khác với vòng lặp for, vòng lặp while và do while dùng để lặp cho trường hợp ta chưa biết tổng số lần lặp. Riêng đối với vòng lặp do while thì nó sẽ luôn luôn lặp ít nhất một lần. Xem thêm tại vòng lặp while và do while trong Javascript.

**Vòng lặp while**

Cú pháp:

*while (variable<=endvalue) {*

*//code block to be executed*

*}*

Ví dụ:

*var i = 10;*

*while (i <= 1000) {*

*console.log(i);*

*i++;*

*}*

**Vòng lặp do while**

Cú pháp:

*do{*

*// do something*

*}*

*while (variable<=endvalue);*

Ví dụ:

*var i = 10;s*

*do {*

*console.log(i);*

*i++;*

*} while (i <= 1000);*

**d) Vòng lặp for of**

Về bản chất vòng lặp for of giống vòng lặp for in, tuy nhiên điểm khác biệt duy nhất là vòng lặp for of sẽ trả về value chứ không phải là index.

Ví dụ:

*let numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];*

*var s = "";*

*var o = 0;*

*for (let num of numbers) {*

*s = s + "\n Array Value is - " + num;*

*}*

*alert(s);*

2.5 Hàm và cách khai báo

2.5.1 Khai báo

Giống như trong Javascript(Js) hàm trong Typescript(Ts) có cú pháp khai báo cũng rất đơn giản :

- Name function:

*function add(x: number, y: number): number {*

*return x + y;*

*};*

*- Anonymous function:*

*let myAdd = function(x: number, y: number): number {*

*return x + y;*

*};*

Bên trên là 2 cách khai báo function cơ bản nhất trong TypeScript :

+ Chúng ta bắt đầu với từ ‘function’ quen thuộc rồi đến ‘add’ (tên function) và bên trong dấu ngoặc đơn sẽ là các parameters(tham số). Đến đây chúng ta sẽ nhận thấy điều khác biệt lớn nhất trong cách khai báo hàm giữa Js và Ts là chúng ta sẽ khai báo cả kiểu cho tham số. Tiếp đến là chúng ta sẽ khai báo kiểu trả về cho hàm. Và cuối cùng là chúng ta sẽ viết code logic vào phần thân hàm.

+ Typescript có thể tìm ra kiểu trả về thông qua các câu lệnh return, vì vậy chúng ta có thể tuỳ ý bỏ qua khai báo kiểu trả về trong nhiều trường hợp.

2.5.2 Function type (kiểu hàm):

Bạn có thể khai báo một biến có kiểu là một hàm như sau:

*let varFun: (hello: number, word: number) => number = function(x: number, y: number): number {*

*return x + y;*

*};*

Một function type có 2 phần. Kiểu của các đối số về kiểu trả về. 2 phần này là bắt buộc khi chúng ta muốn khai báo một biến với function type

+ Phần đầu là mình khai báo kiểu và số lượng tham số, xếp theo thứ tự tham số chuyền vào hàm tương ứng.

+ Tiếp theo là phần chỉ ra kiểu trả về của hàm đó, bằng cách sử dụng ‘=>’ ngăn giữa các tham số và loại trả về. Vì đây là phần bắt buộc nên khi hàm không trả về gì thì ta sẽ điền là ‘void’ thay vì bỏ đi.

2.5.3 Inferring the type (suy kiểu)

Khi bạn khai báo một biến với function type như sau:

*let varFun: (hello: number, word: string) => number = function(x, y): number {*

*return x + y;*

*};*

Trình biên dịch sẽ tự động suy ra kiểu của các tham số x, y lần lượt là number và string.

Điều này được gọi là ‘contextual typing’ (suy kiểu theo ngữ cảnh). Nó giúp chúng ta giảm bớt công gõ code.

2.5.4 Optional and default parameter (tham số tuỳ chọn và mặc định)

- Khi gọi một hàm thì ta phải chắc chắn số lượng đối số được cung cấp phải khớp với số lượng tham số mà hàm mong đợi. Ví dụ như:

*function setName(nameOne: string, nameTwo: string): string {*

*return nameOne + nameTwo*

*}*

*const result1 = setName('superman') // error, too few parameters*

*const result2 = setName('ironman', 'spiderman', 'batman') // error, too many parameters*

*const result3 = setName('hello', 'word' ) // hello word*

- Trong Javascript mọi tham số là tuỳ chọn và họ có thể bỏ qua một trong số chúng nếu họ muốn. Chúng ta cũng có thế làm điều này với typescript bằng cách thêm ‘?’ vào sau tham số mà chúng ta muốn nó là tuỳ chọn. Ví dụ như:

*function setName(nameOne: string, nameTwo?: string): string {*

*if(nameTwo) return nameOne + nameTwo;*

*else return nameOne;*

*}*

*const result1 = setName('superman') // superman*

*const result2 = setName('ironman', 'spiderman', 'batman') // error, too many parameters*

*const result3 = setName('hello', 'word' ) // hello word*

**Lưu ý:** Trong hàm thì tham số tuỳ chọn không thể đứng đầu mà nó chỉ có thể viết sau tham số bắt buộc.

- Trong Typescript, chúng ta có thể set một giá trị cho một tham số khi người dùng không cung cấp. Chúng được gọi là tham số khởi tạo mặc định. Ví dụ như :

*function setName(nameOne: string, nameTwo = "man"): string {*

*return nameOne + " " + nameTwo;*

*}*

*const result1 = setName('super') // super man*

*const result2 = setName('ironman', 'spiderman', 'batman') // error, too many parameters*

*const result3 = setName('hello', 'word' ) // hello word*

*const result4 = setName('wonder', undefined) // wonder man*

**Lưu ý:** Cũng giống như các tham số tuỳ chọn, vị trí của tham số mặc định phải nên được viết sau tất cả các tham số bắt buộc. Tuy nhiên nếu chúng ta vẫn muốn viết chúng trước các tham số bắt buộc thì khi gọi hàm đó chúng ta một là phải truyền giá trị cho chúng, hai là vượt qua dùng giá trị mặc định bằng cách truyền ‘undefined’.

2.4.5 Rest Parameter

Đôi khi bạn sẽ muốn làm việc với rất nhiều tham số trong một kiểu. Hay là bạn không biết phải nên có bao nhiêu tham số cho hàm này.

Trong TypeScript chúng ta sẽ tập hợp những tham số đó lại thành 1 biến. Ví dụ như:

*function setName(nameOne: string,... arrayNameTwo: string[]): string {*

*return nameOne + " " + arrayNameTwo.join("");*

*}*

*=> Khi đó chúng ta có thể truyền bao nhiêu tham số tuỳ thích.*

*let result = setName("ironman", "superman", "batman", "spiderman").*

Rest parameter được coi là vô số các tham số tuỳ chọn. Khi truyền đối số cho rest parameter thì ta có thể truyền bao nhiêu tuỳ ý. Trình biên dịch sẽ tạo thành 1 mảng các đối số truyền vào với tên được đặt sau dấu (…), cho phép bạn sử dụng nó trong hàm của mình.

2.5.6 Arrow function

**a) Khai báo**

Ví dụ khái báo arrow function như sau:

*var hello = (name, message) => {*

*console.log("Chào " + name + ", bạn là " + message);*

*};*

*hello('Cường', 'admin freetuts.net');*

Cách viết thường

*function hello(name, message)*

*{*

*console.log("Chào " + name + ", bạn là " + message);*

*}*

*hello('Cường', 'admin freetuts.net');*

So sánh hai các trên thì rõ ràng cách thông thường sẽ đơn giản hơn rất nhiều, và cả hai đoạn code đều cho kết quả như sau:

**Nội dung là một câu lệnh đơn:**

Trường hợp trong thân của hàm chỉ có một lệnh duy nhất thì bạn có thể sử dụng theo ví dụ dưới đây.

*var hello = (name, message) => console.log("Chào " + name + ", bạn là " + message);*

Nghĩa là bạn có thể bỏ đi cặp dấu {}, điều này tuân thủ theo nguyên tắc "nếu bên thân cặp {} chỉ là một câu lệnh thì bạn có thể bỏ cặp {}".

**Trường hợp một tham số:**

Trường hợp truyền vào chỉ một tham số thì bạn có thể bỏ cặp ().

*var hello = message => {*

*console.log(message);*

*};*

*hello('Chào mừng bạn đến với utehy');*

**Trường hợp không có tham số:**

Trường hợp không có tham số truyền vào thì bạn sử dụng cặp () rỗng, xem ví dụ sau:

*var hello = () => {*

*console.log('Chào mừng bạn đến với utehyt');*

*};*

*hello();*

2.6 Khám phá chi tiết một số kiểu dữ liệu

2.6.1 Kiểu dữ liệu string

Cách khai báo

*let employeeName1:string;*

*let employeeName2:string = 'John Smith';*

*let* employeeName3:string = "John Smith";

Chuỗi là một kiểu dữ liệu nguyên thủy khác được sử dụng để lưu trữ dữ liệu văn bản. Các giá trị chuỗi được bao quanh bởi dấu ngoặc kép đơn hoặc dấu ngoặc kép.

Kể từ phiên bản TypeScript 1.4, TypeScript đã bao gồm hỗ trợ cho các chuỗi Mẫu ES6. Chuỗi mẫu được sử dụng để nhúng biểu thức vào chuỗi.

*let employeeName:string = "John Smith";*

*let employeeDept:string = "Finance";*

*// Pre-ES6*

*let employeeDesc1: string = employeeName + " works in the " + employeeDept + " department.";*

*// Post-ES6*

*let employeeDesc2: string = `${employeeName} works in the ${employeeDept} department.`;*

*console.log(employeeDesc1);//John Smith works in the Finance department.*

*console.log(employeeDesc2);//John Smith works in the Finance department.*

Ở đây, thay vì viết một chuỗi là sự kết hợp giữa văn bản và các biến với các phép nối, chúng ta có thể sử dụng một câu lệnh với dấu back-tick `. Các giá trị biến được viết là . Sử dụng các chuỗi mẫu, việc nhúng các biểu thức sẽ dễ dàng hơn và cũng ít tẻ nhạt hơn khi viết các chuỗi dựa trên văn bản dài.${}

**Một số hàm xử lý xâu**

| **Tên hàm** | **Mô tả** |
| --- | --- |
| [charAt()](https://www.tutorialsteacher.com/typescript/typescript-string#charat) | Trả về ký tự ở chỉ mục đã cho |
| [concat()](https://www.tutorialsteacher.com/typescript/typescript-string#concat) | Trả về kết hợp của hai hoặc nhiều chuỗi được chỉ định |
| [indexOf()](https://www.tutorialsteacher.com/typescript/typescript-string#indexof) | Trả về một chỉ mục xuất hiện đầu tiên của chuỗi con được chỉ định từ một chuỗi (-1 nếu không tìm thấy) |
| [replace()](https://www.tutorialsteacher.com/typescript/typescript-string#replace) | Thay thế chuỗi con phù hợp bằng một chuỗi con mới |
| [split()](https://www.tutorialsteacher.com/typescript/typescript-string#split) | Chia chuỗi thành chuỗi con và trả về một mảng |
| [toUpperCase()](https://www.tutorialsteacher.com/typescript/typescript-string#touppercase) | Chuyển đổi tất cả các ký tự của chuỗi thành chữ hoa |
| [toLowerCase()](https://www.tutorialsteacher.com/typescript/typescript-string#tolowercase) | Chuyển đổi tất cả các ký tự của chuỗi thành chữ thường |
| charCodeAt() | Trả về một số là giá trị đơn vị mã UTF-16 tại chỉ mục đã cho |
| codePointAt() | Trả về số nguyên không âm là giá trị điểm mã của điểm mã được mã hóa UTF-16 bắt đầu từ chỉ mục đã chỉ định |
| includes() | Kiểm tra xem một chuỗi có bao gồm một chuỗi khác không |
| endsWith() | Kiểm tra xem một chuỗi kết thúc bằng một chuỗi khác |
| LastIndexOf() | Trả về chỉ số xuất hiện lần cuối của giá trị trong chuỗi |
| localeCompare() | Kiểm tra xem một chuỗi đến trước, sau hay giống với chuỗi đã cho |
| match() | Khớp một biểu thức chính quy với chuỗi đã cho |
| normalize() | Trả về dạng chuẩn hóa Unicode của chuỗi đã cho. |
| padEnd() | Đệm phần cuối của chuỗi hiện tại bằng chuỗi đã cho |
| padStart() | Đệm phần đầu của chuỗi hiện tại bằng chuỗi đã cho |
| repeat() | Trả về một chuỗi bao gồm các phần tử của đối tượng được lặp lại trong thời gian đã cho. |
| search() | Tìm kiếm sự trùng khớp giữa một biểu thức chính quy và một chuỗi |
| slice() | Trả về một phần của chuỗi |
| startsWith() | Kiểm tra xem một chuỗi bắt đầu bằng một chuỗi khác |
| substr() | Trả về một chuỗi bắt đầu tại vị trí đã chỉ định và của các ký tự đã cho |
| substring() | Trả về một chuỗi giữa hai chỉ mục đã cho |
| toLocaleLowerCase() | Trả về một chuỗi chữ thường trong khi tôn trọng miền địa phương hiện tại |
| toLocaleUpperCase() | Trả về một chuỗi chữ hoa trong khi tôn trọng miền địa phương hiện tại |
| trim() | Cắt khoảng trắng từ đầu và cuối chuỗi |
| trimLeft() | Cắt khoảng trắng từ phía bên trái của chuỗi |
| trimRight() | Cắt khoảng trắng từ phía bên phải của chuỗi |

Thông tin chi tiết các em tra cứu trong địa chỉ sau: [***https://www.tutorialspoint.com/typescript/typescript\_strings.htm***](https://www.tutorialspoint.com/typescript/typescript_strings.htm)

2.6.2 Kiểu dữ liệu Arrays

Mảng là một loại dữ liệu đặc biệt có thể lưu trữ nhiều giá trị của các loại dữ liệu khác nhau theo cách sử dụng một cú pháp đặc biệt.

TypeScript hỗ trợ các mảng, tương tự như JavaScript. Có hai cách để khai báo một mảng:

***Sử dụng dấu ngoặc vuông. Phương pháp này tương tự như cách bạn khai báo mảng trong JavaScript.***

*let fruits: string[] = ['Apple', 'Orange', 'Banana'];*

***Sử dụng kiểu mảng chung, Mảng <ElementType>.***

*let fruits: Array<string> = ['Apple', 'Orange', 'Banana'];*

Cả hai phương pháp đều tạo ra cùng một đầu ra.

Tất nhiên, bạn luôn có thể khởi tạo một mảng như hiển thị bên dưới, nhưng bạn sẽ không nhận được lợi thế của hệ thống loại TypeScript.

let arr = [1, 3, 'Apple', 'Orange', 'Banana', true, false];

Mảng có thể chứa các phần tử của bất kỳ loại dữ liệu, số, chuỗi hoặc thậm chí các đối tượng.

Mảng có thể được khai báo và khởi tạo riêng.

Ví dụ: Khai báo và khởi tạo mảng

*let fruits: Array<string>;*

*fruits = ['Apple', 'Orange', 'Banana'];*

*let ids: Array<number>;*

*ids = [23, 34, 100, 124, 44];*

Một mảng trong TypeScript có thể chứa các thành phần của các loại dữ liệu khác nhau bằng cách sử dụng cú pháp kiểu mảng chung, như được hiển thị bên dưới.

Ví dụ: Mảng nhiều loại

*let values: (string | number)[] = ['Apple', 2, 'Orange', 3, 4, 'Banana'];*

*// or*

*let values: Array<string | number> = ['Apple', 2, 'Orange', 3, 4, 'Banana'];*

## Truy cập các phần tử mảng:

Các phần tử mảng có thể được truy cập bằng cách sử dụng chỉ mục của một phần tử, vd ArrayName[index]. Chỉ số mảng bắt đầu từ 0, vì vậy chỉ mục của phần tử thứ nhất bằng 0, chỉ mục của phần tử thứ hai là một và cứ thế.

Ví dụ: Các phần tử mảng truy cập

*let fruits: string[] = ['Apple', 'Orange', 'Banana'];*

*fruits[0]; // returns Apple*

*fruits[1]; // returns Orange*

*fruits[2]; // returns Banana*

*fruits[3]; // returns undefined*

Sử dụng vòng lặp for để truy cập các phần tử mảng như dưới đây.

Ví dụ: Access Array Elements bằng Loop

*let fruits: string[] = ['Apple', 'Orange', 'Banana'];*

*for(var index in fruits)*

*{*

*console.log(fruits[index]); // output: Apple Orange Banana*

*}*

*for(var i = 0; i < fruits.length; i++)*

*{*

*console.log(fruits[i]); // output: Apple Orange Banana*

*}*

**Một số hàm thông dụng trên mảng**

|  |  |
| --- | --- |
| **STT** | **Hàm và mô tả** |
| 1 | [**concat()**](https://www.tutorialspoint.com/typescript/typescript_array_concat.htm)  Trả về một mảng mới bao gồm mảng này được nối với (các) mảng và / hoặc giá trị khác. |
| 2 | [**every()**](https://www.tutorialspoint.com/typescript/typescript_array_every.htm)  Trả về true nếu mọi phần tử trong mảng này thỏa mãn chức năng kiểm tra được cung cấp. |
| 3 | [**filter()**](https://www.tutorialspoint.com/typescript/typescript_array_filter.htm)  Tạo một mảng mới với tất cả các thành phần của mảng này mà hàm lọc được cung cấp trả về true.  Ví dụ 1:  *function isBigEnough(element, index, array) {*  *return (element >= 10);*  *}*  *var passed = [12, 5, 8, 130, 44].filter(isBigEnough);*  *console.log("Test Value : " + passed );*  Kết quả: Test Value :12,130,44  Ví dụ 2:  *var passed = [12, 5, 8, 130, 44].filter(x => x>=10);*  *console.log("Test Value : " + passed );*  Kết quả: Test Value :12,130,44 |
| 4 | [**forEach()**](https://www.tutorialspoint.com/typescript/typescript_array_foreach.htm)  Gọi một hàm cho mỗi phần tử trong mảng.  Ví dụ 1:  let num = [7, 8, 9];  num.forEach(function (value) {  console.log(value);  });  Kết quả:  7  8  9  Ví dụ 2:  let result = "";  let myArray = [{name:"a"}, {name:""}, {name:"b"}, {name:"c"}];  myArray.forEach( (myObject, index) => {  if(myObject.name){  result +=myArray[index].name;  }  });  Kết quả: abc |
| 5 | [**indexOf()**](https://www.tutorialspoint.com/typescript/typescript_array_indexof.htm)  Trả về chỉ mục (ít nhất) đầu tiên của một phần tử trong mảng bằng với giá trị được chỉ định hoặc -1 nếu không tìm thấy. |
| 6. | [**join()**](https://www.tutorialspoint.com/typescript/typescript_array_join.htm)  Nối tất cả các phần tử của một mảng thành một chuỗi. |
| 7 | [**lastIndexOf()**](https://www.tutorialspoint.com/typescript/typescript_array_lastindexof.htm)  Trả về chỉ mục (lớn nhất) cuối cùng của một phần tử trong mảng bằng với giá trị được chỉ định hoặc -1 nếu không tìm thấy. |
| 8 | [**map()**](https://www.tutorialspoint.com/typescript/typescript_array_map.htm)  Tạo một mảng mới với kết quả gọi một hàm được cung cấp trên mọi phần tử trong mảng này.  Ví dụ 1:  *var numbers = [1, 4, 9];*  *var roots = numbers.map(Math.sqrt);*  *console.log("roots is : " + roots );*  Kết quả: roots is : 1,2,3  Ví dụ 2:  *const myUsers = [*  *{ name: 'shark', likes: 'ocean' },*  *{ name: 'turtle', likes: 'pond' },*  *{ name: 'otter', likes: 'fish biscuits' }*  *]*  *const usersByLikes = myUsers.map(item => {*  *const container = {};*  *container[item.name] = item.likes;*  *container.age = item.name.length \* 10;*  *return container;*  *})*  *console.log(usersByLikes);*  Kết quả:  *[*  *{shark: "ocean", age: 50},*  *{turtle: "pond", age: 60},*  *{otter: "fish biscuits", age: 50}*  *]* |
| 9 | [**pop()**](https://www.tutorialspoint.com/typescript/typescript_array_pop.htm)  Loại bỏ phần tử cuối cùng khỏi một mảng và trả về phần tử đó. |
| 10 | [**push()**](https://www.tutorialspoint.com/typescript/typescript_array_push.htm)  Thêm một hoặc nhiều phần tử vào cuối một mảng và trả về độ dài mới của mảng. |
| 11 | [**reduce()**](https://www.tutorialspoint.com/typescript/typescript_array_reduce.htm)  Áp dụng đồng thời một hàm đối với hai giá trị của mảng (từ trái sang phải) để giảm nó thành một giá trị.  Ví dụ 1:  *var total = [0, 1, 2, 3].reduce(function(a, b){ return a + b; });*  *console.log("total is : " + total );*  Kết quả: *total is :6*  Ví dụ 2:  *let array=[5,4,19,2,7];*  *function findMax(acc,val)*  *{*  *if(val>acc){*  *acc=val;*  *}*  *}*  *let biggest=arrays.reduce(findMax); // 19* |
| 12 | [**reduceRight()**](https://www.tutorialspoint.com/typescript/typescript_array_reduceright.htm)  Áp dụng đồng thời một hàm đối với hai giá trị của mảng (từ phải sang trái) để giảm nó thành một giá trị. |
| 13 | [**reverse()**](https://www.tutorialspoint.com/typescript/typescript_array_reverse.htm)  Đảo ngược thứ tự các phần tử của một mảng - cái đầu tiên trở thành cái cuối cùng và cái cuối cùng trở thành cái đầu tiên. |
| 14 | [**shift()**](https://www.tutorialspoint.com/typescript/typescript_array_shift.htm)  Loại bỏ phần tử đầu tiên khỏi một mảng và trả về phần tử đó. |
| 15 | [**slice()**](https://www.tutorialspoint.com/typescript/typescript_array_slice.htm)  Trích xuất một phần của một mảng và trả về một mảng mới. |
| 16 | [**some()**](https://www.tutorialspoint.com/typescript/typescript_array_some.htm)  Trả về true nếu có ít nhất một phần tử trong mảng này thỏa mãn chức năng kiểm tra được cung cấp. |
| 17 | [**sort()**](https://www.tutorialspoint.com/typescript/typescript_array_sort.htm)  Sắp xếp các phần tử của một mảng. |
| 18 | [**splice()**](https://www.tutorialspoint.com/typescript/typescript_array_splice.htm)  Thêm và / hoặc loại bỏ các phần tử từ một mảng. |
| 19 | [**toString()**](https://www.tutorialspoint.com/typescript/typescript_array_tostring.htm)  Trả về một chuỗi đại diện cho mảng và các phần tử của nó. |
| 20 | [**unshift()**](https://www.tutorialspoint.com/typescript/typescript_array_unshift.htm)  Thêm một hoặc nhiều phần tử vào phía trước của một mảng và trả về độ dài mới của mảng. |

Thông tin chi tiết các em tra cứu trong địa chỉ sau: [***https://www.tutorialspoint.com/typescript/typescript\_arrays.htm***](https://www.tutorialspoint.com/typescript/typescript_arrays.htm)

2.7 Lập trình hướng đối tượng

Trong các ngôn ngữ lập trình hướng đối tượng như Java và C #, các lớp là các thực thể cơ bản được sử dụng để tạo các thành phần có thể sử dụng lại. Các chức năng được truyền lại cho các lớp và các đối tượng được tạo từ các lớp. Tuy nhiên, cho đến ECMAScript 6 (còn được gọi là ECMAScript 2015), đây không phải là trường hợp của JavaScript. JavaScript chủ yếu là một ngôn ngữ lập trình chức năng trong đó tính kế thừa dựa trên nguyên mẫu. Các chức năng được sử dụng để xây dựng các thành phần có thể tái sử dụng. Trong ECMAScript 6, cách tiếp cận dựa trên lớp hướng đối tượng đã được giới thiệu. TypeScript đã giới thiệu các lớp để tận dụng lợi ích của các kỹ thuật hướng đối tượng như đóng gói và trừu tượng hóa. Lớp trong TypeScript được biên dịch thành các hàm JavaScript đơn giản bởi trình biên dịch TypeScript để hoạt động trên các nền tảng và trình duyệt.

Một lớp có thể bao gồm:

* Constructor
* Tính chất
* Phương thưcs

Sau đây là một ví dụ về một lớp trong TypeScript:

**Class**

Ví dụ:

*class Employee {*

*empCode: number;*

*empName: string;*

*constructor(code: number, name: string) {*

*this.empName = name;*

*this.empCode = code;*

*}*

*getSalary() : number {*

*return 10000;*

*}*

*}*

Trình biên dịch TypeScript sẽ chuyển đổi lớp trên thành mã JavaScript sau bằng cách sử dụng bao [đóng](https://www.tutorialsteacher.com/javascript/closure-in-javascript) :

var Employee = /\*\* @class \*/ (function () {

function Employee(name, code) {

this.empName = name;

this.empCode = code;

}

Employee.prototype.getSalary = function () {

return 10000;

};

return Employee;

}());

## Constructor

Hàm tạo là một loại phương thức đặc biệt được gọi khi tạo đối tượng. Trong TypeScript, phương thức constructor luôn được định nghĩa với tên "constructor".

Ví dụ: Employee

*class Employee {*

*empCode: number;*

*empName: string;*

*constructor(empcode: number, name: string ) {*

*this.empCode = empcode;*

*this.name = name;*

*}*

*}*

Trong ví dụ trên, Employee lớp bao gồm một hàm tạo với các tham số empcode và name. Trong hàm tạo, các thành viên của lớp có thể được truy cập bằng this từ khóa this.empCode hoặc this.name.

Không cần thiết cho một lớp để có một hàm tạo.

Ví dụ: Lớp không có Trình xây dựng

*class Employee {*

*empCode: number;*

*empName: string;*

*}*

## Tạo một đối tượng của lớp

Một đối tượng của lớp có thể được tạo bằng cách sử dụng [từ khóa mới](https://www.tutorialsteacher.com/javascript/new-keyword-in-javascript) .

Ví dụ: Tạo một đối tượng

*class Employee {*

*empCode: number;*

*empName: string;*

*}*

*let emp = new Employee();*

Ở đây, chúng ta tạo một đối tượng được gọi là emp kiểu Employee sử dụng let emp = new Employee();. Lớp trên không bao gồm bất kỳ hàm tạo tham số nào, vì vậy chúng ta không thể truyền các giá trị trong khi tạo một đối tượng. Nếu lớp bao gồm một hàm tạo được tham số hóa, thì chúng ta có thể truyền các giá trị trong khi tạo đối tượng.

*class Employee {*

*empCode: number;*

*empName: string;*

*constructor(empcode: number, name: string ) {*

*this.empCode = empcode;*

*this.name = name;*

*}*

*}*

*let emp = new Employee(100,"Steve");*

Trong ví dụ trên, chúng ta truyền các giá trị cho đối tượng để khởi tạo các biến thành viên. Khi chúng ta khởi tạo một đối tượng mới, hàm tạo của lớp được gọi với các giá trị được truyền và các biến thành viên empCode và empName được khởi tạo với các giá trị này.

## Kế thừa

Giống như các ngôn ngữ hướng đối tượng như Java và C #, các lớp TypeScript có thể được mở rộng để tạo các lớp mới có tính kế thừa, sử dụng từ khóa extends.

Ví dụ: Kế thừa

*class Person {*

*name: string;*

*constructor(name: string) {*

*this.name = name;*

*}*

*}*

*class Employee extends Person {*

*empCode: number;*

*constructor(empcode: number, name:string) {*

*super(name);*

*this.empCode = empcode;*

*}*

*displayName():void {*

*console.log("Name = " + this.name + ", Employee Code = " + this.empCode);*

*}*

*}*

*let emp = new Employee(100, "Bill");*

*emp.displayName(); // Name = Bill, Employee Code = 100*

Trong ví dụ trên, Employee lớp mở rộng từ lớp Person bằng cách sử dụng từ extends. Điều này có nghĩa là lớp Employee hiện bao gồm tất cả các thành viên của lớp Person.

Hàm tạo của lớp Employee khởi tạo các thành viên của chính nó cũng như các thuộc tính của lớp cha bằng cách sử dụng một từ khóa đặc biệt ' *super* '. Các từ khóa super được sử dụng để gọi constructor lớp cha.

**Chú ý:** Chúng ta phải gọi phương thức super () trước khi gán giá trị cho các thuộc tính trong hàm tạo của lớp dẫn xuất.

### Interface

Một lớp có thể thực hiện một hoặc nhiều giao diện.

Ví dụ: Giao diện triển khai

*interface IPerson {*

*name: string;*

*display():void;*

*}*

*interface IEmployee {*

*empCode: number;*

*}*

*class Employee implements IPerson, IEmployee {*

*empCode: number;*

*name: string;*

*constructor(empcode: number, name:string) {*

*this.empCode = empcode;*

*this.name = name;*

*}*

*display(): void {*

*console.log("Name = " + this.name + ", Employee Code = " + this.empCode);*

*}*

*}*

*let per:IPerson = new Employee(100, "Bill");*

*per.display(); // Name = Bill, Employee Code = 100*

*let emp:IEmployee = new Employee(100, "Bill");*

*emp.display(); //Compiler Error: Property 'display' does not exist on type 'IEmployee'*

## Giao diện mở rộng lớp

Một giao diện cũng có thể mở rộng một lớp để thể hiện một loại.

Ví dụ: Giao diện mở rộng lớp

*class Person {*

*name: string;*

*}*

*interface IEmployee extends Person {*

*empCode: number;*

*}*

*let emp: IEmployee = { empCode : 1, name:"James Bond" }*

Trong ví dụ trên, Iemployee là một giao diện mở rộng lớp Person. Vì vậy, chúng ta có thể khai báo một biến loại Iemployee có hai thuộc tính. Vì vậy, bây giờ, chúng ta phải khai báo và khởi tạo các giá trị cùng một lúc.

## Phương pháp ghi đè

Khi một lớp con định nghĩa việc thực hiện phương thức riêng của nó từ lớp cha, nó được gọi là phương thức ghi đè.

Ví dụ: Phương thức ghi đè

*class Car {*

*name: string;*

*constructor(name: string) {*

*this.name = name;*

*}*

*run(speed:number = 0) {*

*console.log("A " + this.name + " is moving at " + speed + " mph!");*

*}*

*}*

*class Mercedes extends Car {*

*constructor(name: string) {*

*super(name);*

*}*

*run(speed = 150) {*

*console.log('A Mercedes started')*

*super.run(speed);*

*}*

*}*

*class Honda extends Car {*

*constructor(name: string) {*

*super(name);*

*}*

*run(speed = 100) {*

*console.log('A Honda started')*

*super.run(speed);*

*}*

*}let mercObj = new Mercedes("Mercedes-Benz GLA");*

*let hondaObj = new Honda("Honda City")*

*mercObj.run(); // A Mercedes started A Mercedes-Benz GLA is moving at 150 mph!*

*hondaObj.run(); // A Honda started A Honda City is moving at 100 mph!*

Trong ví dụ trên, chúng ta có một lớp Car với thuộc tính name. Hàm tạo cho lớp này khởi tạo các biến thành viên. Lớp này cũng có một phương thức display() với tốc độ đối số được khởi tạo là 0.

Sau đó chúng ta tạo hai lớp Mercedes và Honda, mở rộng từ lớp cha Car. Mỗi lớp con mở rộng các thuộc tính của lớp cha. Hàm tạo cho mỗi lớp gọi siêu xây dựng để khởi tạo các thuộc tính của lớp cha. Mỗi lớp cũng định nghĩa một phương thức run() trong thông điệp riêng của mình ngoài việc gọi phương thức siêu lớp cho run().

Vì mỗi lớp con có cách thực hiện riêng của phương thức run(), nên nó được gọi là phương thức ghi đè, tức là các lớp con có một phương thức cùng tên với phương thức của lớp cha.

Khi chúng ta tạo các đối tượng của lớp con và gọi run()phương thức trên đối tượng này, nó sẽ gọi phương thức được ghi đè của chính nó run()và không phải là phương thức của lớp cha.

**Bài 3: JSX, Components, Props, Props trong ReactJS**

* 1. **Giới thiệu JSX trong ReactJS**

JSX là viết tắt là Javascript XML, một template languges nhưng nó lại mang hầu hết tính năng của Javascript. Nó cho phép bạn viết các đoạn mã HTML trong React một cách dẽ dàng và có cấu trúc hơn. React sử dụng JSX cho việc xây dựng bố cục thay vì javascript thông thường. JSX giúp tạo ra các React 'elements'. Việc sử dụng nó trong ReactJS rất hữu ích bởi:

* JSX giúp cho việc xây dựng các ứng dụng React một cách nhanh hơn, dễ tối ưu trong việc complie code sang javascript.
* JSX rất dễ xem các lỗi trong quá trình triển khai bởi hầu hết các lỗi sẽ được hiển thị trong quá trình compile, không như các đoạn mã HTML có thể thừa thiếu các thể div khiến giao diện bị hiển thị sai. JSX lại hoàn toàn ngược lại, khi bạn quên đóng div chẳng hạn thì nó lập tực sẽ hiển thị lỗi.
* Cú pháp khá giống với HTML nên dễ dàng cho việc viết chuyển đổi.

**Sử dụng JSX trong ReactJs**

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().

Ví dụ :

Có JSX, đoạn code sẽ như sau:

const myelement = <h1>I Love JSX!</h1>;

Không có JSX, đoạn code sẽ như sau:

const myelement = React.createElement('h1', {}, 'I do not use JSX!');

Có thể thấy, trong ví dụ đầu tiên, JSX cho phép chúng ta viết HTML trực tiếp trong JavaScript. JSX là một phần mở rộng của ngôn ngữ JavaScript dựa trên ES6 và được dịch sang JavaScript thông thường trong thời gian chạy mã.

**Biểu thức trong JSX**

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.

Ví dụ 1: Thực hiện biểu thức 5+5:

const myelement = <h1>React is {5 + 5} times better with JSX</h1>;

Ví dụ 2:

function formatName(user) {

return user.firstName + ' ' + user.lastName;

}

const user = {

firstName: 'Harper',

lastName: 'Perez'

};

const element = (

<h1>

Hello, {formatName(user)}!

</h1>

);

**JSX cũng là một biểu thức**

Sau khi biên dịch (compile), biểu thức JSX là những gọi hàm bình thường của JavaScript và thành những đối tượng JavaScript sau khi được gọi. Điều này có nghĩa là bạn có thể dùng JSX bên trong câu lệnh if cũng như vòng lặp for, gán nó cho biến, dùng như tham số hàm, và trả về JSX từ hàm.

function getGreeting(user) {

if (user) {

return <h1>Hello, {formatName(user)}!</h1>;

}

return <h1>Hello, Stranger.</h1>;

}

**Phần tử con trong JSX**

Nếu chỉ có một tag bạn chỉ cần đóng nó bằng dấu /> như ví dụ :

const element = <img src={user.avatarUrl} />;

trong trường hợp trong tag có nhiều phần tử con bạn cần phải bọc ngoài nó bằng một JSX tags:

//Đúng cú pháp

//Phải bọc nó bằng một tags

const element = (

<div>

<h1>Hello</h1>

<p>Welcome to Freetuts</p>

</div>

);

//Viết sai

//Các phần từ không được bọc

const element = (

<h1>Hello</h1>

<p>Welcome to Freetuts</p>

);

**Xác định thuộc tính của thẻ với JSX**

Bạn có thể dùng dấu nháy để khai báo một chuỗi như là thuộc tính của thẻ:

const element = <a href="https://www.reactjs.org"> link </a>;

Bạn có thể dùng dấu ngoặc nhọn để nhúng một biểu thức Javascript vào trong thuộc tính:

const element = <img src={user.avatarUrl}></img>;

Đừng để dấu nháy xung quanh ngoặc nhọn khi nhúng biểu thức JavaScript vào trong một thuộc tính. Bạn có thể dùng dấu nháy (cho giá trị chuỗi) hoặc ngoặc nhọn (cho biểu thức), nhưng không được dùng cả hai cho cùng 1 thuộc tính.

**Thuộc tính JSX trong ReactJs**

Bạn có thể sử dụng các thuộc tính tùy chỉnh của riêng mình ngoài các thuộc tính HTML thông thường. Khi muốn thêm thuộc tính tùy chỉnh của JSX trong ReactJS, bạn cần sử dụng tiền tố data-, Trong ví dụ sau, data-myattribute đã được thêm làm thuộc tính của phần tử p.

import React from 'react';

class App extends React.Component {

render() {

return (

<div>

<h1>Header</h1>

<h2>Content</h2>

<p data-myattribute = "somevalue">This is the content!!!</p>

</div>

);

}

}

export default App;

**Tạo kiểu JSX trong ReactJs**

React khuyên các lập trình viên nên sử dụng các kiểu nội tuyến (inline styles). Khi muốn thiết lập các kiểu nội tuyến, bạn cần phải sử dụng cú pháp camelcase. React sẽ tự động thêm px sau giá trị số trên các phần tử cụ thể. Ví dụ sau đây sẽ cho bạn thấy cách để thêm nội tuyến myStyle vào phần tử h1.

import React from 'react';

class App extends React.Component {

render() {

var myStyle = {

fontSize: 100,

color: '#FF0000'

}

return (

<div>

<h1 style = {myStyle}>Header</h1>

</div>

);

}

}

export default App;

**Ngăn chặn Injection Attacks**

Đây là một tính năng bảo mật của React, React DOM sẽ tiến hành escaped tất cả các giá trị bên trong JSX một cách tự động trước khi render chúng, điều này rất hữu ích cho việc ngăn chặn các hình thức tấn công bằng cách tiêm mã độc.

//Khi sử dụng trong React sẽ không nguy hiểm

const content = '<script>XSS</script>'

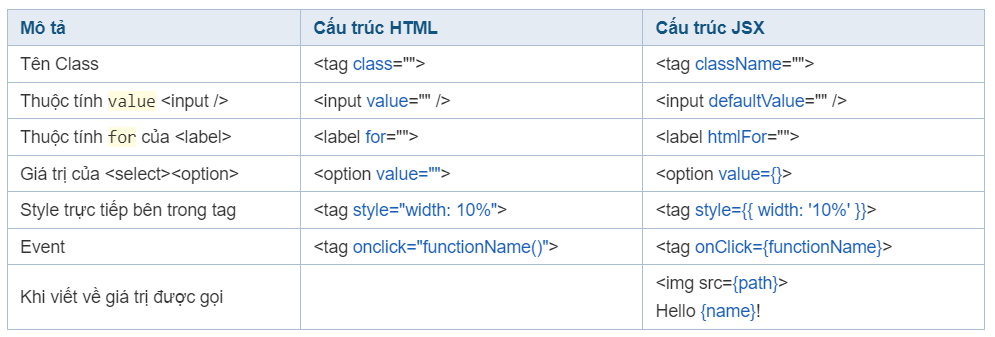
const element = <p className="welcome">{content}</p>;

**Quy ước đặt tên JSX trong ReactJS**

Các thẻ HTML luôn sử dụng tên thẻ viết thường, trong khi các thành phần React bắt đầu bằng chữ hoa. Lưu ý: Bạn nên sử dụng className và htmlFor làm tên thuộc tính XML thay vì class và for.

Trên trang web chính thức của React, điều này được giải thích là do JSX là JavaScript nên các định danh như class và for không được khuyến khích làm tên thuộc tính XML. Thay vào đó, các thành phần React DOM yêu cầu các tên thuộc tính DOM tương ứng như className và htmlFor.

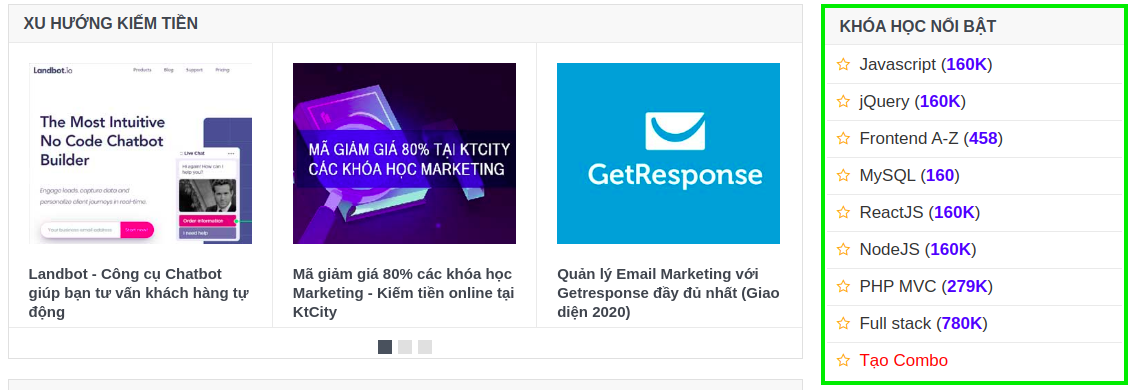
Cấu trúc JSX tương đồng với cấu trúc HTML, tuy nhiên có một số điểm khác biệt ta cần chú ý:



Kết luận: JSX trong ReactJS là một cú pháp mở rộng cho phép người dùng lập trình ứng dụng dễ dàng hơn, bạn có thể tận dụng những ưu điểm vượt trội của JSX để giúp việc lập trình nhanh chóng và hiệu quả hơn.

* 1. Components trong ReactJS

Components giúp phân chia các UI (giao diện người dùng) thành các phân nhỏ để dễ dàng quản lý và tái sử dụng. Giả sử mình có một website gồm nhiều phần bố cục khác nhau và mình muốn chia nhỏ các phần ra để dễ quản lý.



Ở hình ảnh bên trên chúng ta có thể chia nó thành 2 components, đó là phần "khóa học nổi bật" và "xu hướng kiếm tiền". Mỗi components sẽ đảm nhiệm phần hiển thị khác nhau. Khi bạn muốn làm một trang hoàn chỉnh chỉ ghép các components này lại với nhau.

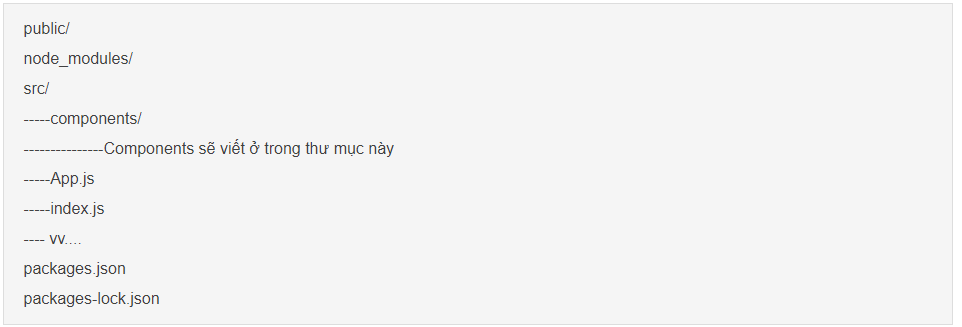
Trong mỗi React App đều có thể chứa rất nhiều components, mỗi components trong đó thường nhận về các props và trả về React elements từ đó hiển thị ra cho UI. Components trong React thường được viết theo 2 loại chính đó là functional component và class components. Bên dưới là một functional components:

const App = () => <h1>Hello Freetuts.net</h1>;

Components bên trên được viết theo cú pháp ES6, không nhận bất cứ props nào và trả về một react element

**Khởi tạo một React Component**

Ở đây mình đề cập 2 cách để khởi tạo một components, mỗi cách có ưu và nhược điểm riêng và bạn có thể lựa chọn theo từng trường hợp sử dụng. Trước khi thực hiện viết components, chúng ta nên khởi tạo một thư mục có tên components trong thư mục src để chứa tất cả các component trong dự án. Cấu trúc thư mục của dự án lúc này sẽ là:



**Functional Component**

Đây là cách viết phổ biến và được sử dụng nhiều bởi tính nhanh gọn và không quá phức tạp như các viết class component. Tiến hành tạo một file có tên Welcome.js trong thư mục src/components:

//Import react vào trong dự án

import React from "react";

const Welcome = function(props) {

return (

<div>

<h1>Welcome ! I am a functional component </h1>

</div>

)

}

export default Welcome;

Bạn cũng có thể nhận các props được truyền vào component bằng cách nhận một tham số trong funciton. Chúng ta sẽ return các react element bằng cách sử dụng cú pháp return() như ví dụ bên trên, cuối file bạn cần phải export component này để file khác có thể lấy để sử dụng. Bây giờ chúng ta sẽ tiến hành import file Welcome.js vào trong file App.js và hiển thị nó ra :

import React from 'react';

import Welcome from './components/Welcome'

function App() {

return (

<div>

<Welcome />

<p>freetuts.net</p>

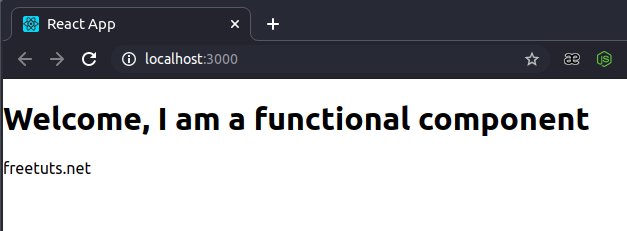
</div>

);

}

export default App;

Tiến hành khởi chạy dự án chúng ta sẽ có kết quả như hình :



Functional component rất gọn nhẹ và không quá nhiều dòng code dài dòng.

Có một vấn đề là trong functional component thì sẽ không có các khái niệm như state, life cycles, events,...nhưng trong phiên bản gần đây thì React cũng đã hỗ trợ thêm React Hooks cho phép bạn làm việc với state, lifecycles dễ dàng hơn trong funcitonal components.

Bởi vậy, bây giờ hầu như funcitonal component có hầu hết chức năng giống như class component nên bạn có thể cân nhắc sử dụng nó nhiều hơn.

**Class Component**

Cách viết này là cách viết đầy đủ của của một component, khi bạn viết một class component bạn sẽ sử dụng được hầu hết các chức năng của component như state, props, lifecycle,..Chúng ta tiếp tục với ví dụ trên, chúng ta chỉ sửa đổi file src/Welcome.js :

import React, { Component } from "react";

class Welcome extends Component {

render() {

return (

<div>

<h1>Welcome ! I am a class component </h1>

</div>

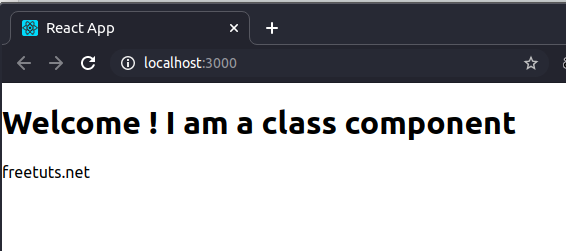
);

}

}

export default Welcome;

Khởi chạy mà bạn sẽ nhận được kết quả giống như ví dụ bên trên.



Khi bạn muốn làm việc với với các chức năng của component như events, state, lifce cycles hay tổ chức các đoạn code theo cấu trúc theo mô hình OOP thì bạn có thể cân nhắc sử dụng class components. Ngược lại, bạn có thể sử dụng functional component để thay thế.

Trong mục này chúng ta đã cùng nhau đi tìm hiểu về components trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

* 1. Tìm hiểu về Props trong ReactJS

Trong lập trình với React, các state và props được sử dụng thường xuyên để điều chỉnh các dòng dữ liệu trong component (thành phần). State và props có nhiều nét giống nhau nhưng cách sử dụng lại hoàn toàn khác biệt nên thường bị nhầm lẫn. Bài viết dưới đây sẽ giúp bạn phần biệt Props và State trong React.

ReactJs là thư viện hoạt động dựa trên các component (thành phần). Về cơ bản, khi sử dụng Reacts trong việc thiết kế UI (giao diện người dùng), ta sẽ chia các component phức tạp thành các component cơ bản. Sau khi phát triển các thành phần cơ bản, chúng ta thêm chúng lại với nhau để tạo thành một giao diện người dùng phức hợp. React kiểm soát các luồng dữ liệu trong các thành phần bằng state và props được sử dụng để kết xuất các component với dữ liệu động.

Props là viết tắt của Properties (thuộc tính), chúng là những component chỉ đọc hoạt động tương tự như các thuộc tính HTML. Trong React, props được sử dụng để gửi dữ liệu đến các component, mọi component trong ReactJs đều được coi là một hàm JavaScript thuần túy, Props cũng đóng vai trò tương đương với các tham số của hàm này. Giá trị của Props có thể được thay đổi bằng cách truyền dữ liệu từ ngoài vào. Vì props là hàm thuần túy nên một khi props đã được truyền vào component, giá trị của nó sẽ không thể thay đổi được.

Như chúng ta đã biết, các component của React sắp xếp giao diện người dùng dưới dạng một cây, trong đó thành phần mẹ đóng vai trò là gốc và các thành phần con trở thành các nhánh con. Giả sử thành phần mẹ muốn gửi dữ liệu đến một trong những thành phần con trở thành nhánh và nhánh con. Nếu thành phần mẹ muốn gửi dữ liệu đến một trong những thành phần lồng sâu trong đó.

Do React tuân theo nguyên tắc các Props phải đi từ thành phần mẹ đến thành phần con, do vậy, bạn không thể bỏ qua bất kỳ lớp nào của các component con khi gửi một prop và component con này cũng không thể gửi ngược prop lại cho thành phần mẹ. Bạn có thể có các props mặc định trong trường hợp các component mẹ không chuyển các props xuống. Đây là lý do tại sao React có liên kết dữ liệu một chiều.

Vì vậy, trong trường hợp này, chúng ta cần gửi dữ liệu từng lớp một cho đến khi nó đến được thành phần đích. Mọi thành phần trong đường dẫn này phải nhận thuộc tính từ từ thuộc tính cha và sau đó gửi lại thuộc tính đó cho con của nó như đã nhận. Quá trình này lặp lại cho đến khi thuộc tính của bạn đạt được thành phần mục tiêu .

Ví dụ về việc di chuyển các Props

import React from 'react';

import ReactDOM from 'react-dom';

class MyComponent extends React.Component{

render(){

return(

<div>

<h1>Hello</h1>

<Header name="maxx" id="101"/>

</div>

);

}

}

function Header(props) {

return (

<div>

<Footer name = {props.name} id = {props.id}/>

</div>

);

}

const Footer = (props: any) => {

  return (

    <div>

      <h1> Welcome : {props.name}</h1>

      <h1> Id is : {props.id}</h1>

    </div>

  );

};

ReactDOM.render(

<MyComponent/>, document.getElementById('content')

);

Tóm lại: 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.

* 1. Tìm hiểu về State trong ReactJS

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.

**Khởi tạo môt state**

Chúng ta có thể khởi tạo một state bằng cách gán giá trị cho biến this.state:

this.state = { hoten : 'Nguyễn Hữu Đông' }

và lấy gía trị của state bằng cách gọi this.state:

console.log(this.state.hoten) //'Nguyễn Hữu Đông

Ví dụ bên dưới mình có một class components và mình sẽ tiến hành khởi tạo state bên trong phương thức constructor :

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

//Chỉ định một state

this.state = { website: " http://fit.utehy.edu.vn" };

}

render() {

return (

<div>

<h1>Website khoa CNTT {this.state.website} </h1>

</div>

);

}

}

export default App;

Trong hầu hết các trường hợp bạn nên khởi tạo state bên trong hàm constructor() để tránh gặp các lỗ không mong muốn. Vì đây sẽ là hàm khởi chạy đầu tiên khi một components được gọi.

**Cập nhật một state**

Trong các components bạn cần phải thao tác với state rất nhiều , ngoài thêm và lấy giá trị của state bạn còn phải cập nhật các states để ReactJS có thể tự động re-render lại components. Điều này khá quan trọng, giả sử bạn đang cho người dùng nhập vào môt form nào đó và khi click Lưu thì nội dung được điền trong form lúc này sẽ phải hiển thị ra màn hình. Đây là lúc bạn cần dùng đến cách thay đổi giá trị của một state.

Để cập nhật một state bạn sử dụng phương thức:

this.setState({

name : 'newValue'

})

Ngoài ra bạn cũng có thể lấy giá trị của state trước khi cập nhật:

this.setState((state) => {

return newValue;

});

Bên dưới mình có một ví dụ về cập nhật state index khi nhấn vào click vào button tương ứng :

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

//Chỉ định một state

this.state = { index: 1 };

}

render() {

return (

<div>

<p>Giá trị {this.state.index}</p>

<button

onClick={() => {

this.setState({

index: this.state.index + 1

})

}}

>

Tăng

</button>

<button

onClick={() => {

this.setState({

index: this.state.index -1

})

}}

>

Giảm

</button>

</div>

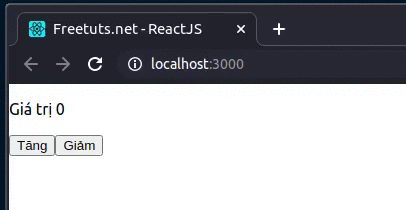
);

}

}

export default App;

Chúng ta sẽ thấy giá trị của state sẽ được thay đổi mỗi khi bạn click vào buttonTăng hoặc Giảm :



State là một khái niệm khá đơn giản nhưng cũng hết sức quan trọng trong component, mặc dù props và state hay đi đôi với nhau nhưng nó hoàn toàn khác biệt với nhau.

**Sự khác nhau giữa props và state**

Trong quá trình học React, cũng có vài trường hợp bạn chưa hiểu về 2 khái niệm props và state kĩ nên có sự nhầm lẫn ở đây.

State - Dữ liệu chỉ nằm trong phạm vi của một component. Nó được sở hữu bởi một components cụ thể mà chỉ là của component đó thôi. Ví dụ, như người yêu bạn chỉ là của bạn vậy =))). Và mỗi khi state thay đổi thì component cũng phải thay đổi theo.

Props - Dữ liệu đường truyền từ component cha cho componet con, components con khi nhận được sẽ chỉ được đọc mà không thể thay đổi dữ liệu đó.

Sự khác nhau chính của 2 khái niệm này là component sở hữu dữ liệu. State là chỉ riêng nó có thể sử dụng. Props là dữ liệu mà component con được nhận về từ một component cha.

Vì phạm vi của state chỉ nằm trong components nên việc truyền dữ liệu từ các components với nhau người ta thường dùng props. Nhưng vấn đề ở đây là props chỉ có thể truyền cho componet con của nó và khi truyền cho các componet cháu, chắt khá rắc rối. Bởi vậy chúng ta có thêm khái niệm về Redux, mình sẽ giới thiệu ở phần tiếp theo vì đây là phần khá phức tạp và đau não :))

Trong bài này chúng ta đã cùng nhau đi tìm hiểu về state trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này. Mong rằng bài viết sẽ giúp ích cho bạn.

Chú ý:

https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/class\_components/

1.4. Tìm hiểu về Component API trong ReactJS

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về các API bên trong các React Component, đây là những API được sử dụng khá nhiều trong quá trình làm việc với các component bao gồm setState(), forceUpdate(), ReactDOM.findDOMNode(),...

**Set State API**

Hàm setSate() này cho phép chúng ta cập nhật giá trị của state, tham số truyền vào của API này sẽ là giá trị của state bạn muốn update:

this.setSate({

website: 'freetuts.net'

})

hoặc là một function callback bao gồm các tham số lần lượt là state trước đó, và props có trong component.

this.setState((prevState, props) => {

return newState;

})

Trong ví dụ bên dưới mình sẽ tiến hành update state bằng 2 cách trên.

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

index: 1

};

}

countDown() {

this.setState({

index: this.state.index - 1

});

}

countUp(){

this.setState((prevState, props) => {

return {

index: prevState.index + 1

}

});

}

render() {

return (

<div>

<p>Giá trị: {this.state.index}</p>

<button onClick={() => this.countDown()}>Down</button>

<button onClick={() => this.countUp()}>Up</button>

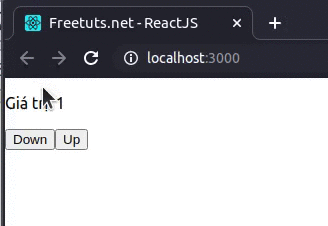
</div>

);

}

}

export default App;



**Force Update**

Component chỉ thực hiện re-render khi các state thay đổi, trong trường hợp cần phải re-render mà không cần bất cứ sự thay đổi nào của state thì chúng ta chỉ cần gọi hàm forceUpdate()

this.forceUpdate()

Bên dưới mình có ví dụ về hàm forceUpdate(), với mỗi lần component được re-render sẽ random ra một số khác nhau:

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

index: 1

};

}

render() {

return (

<div>

<p>Giá trị: {Math.random()}</p>

<button onClick={() => this.forceUpdate()}>Reload</button>

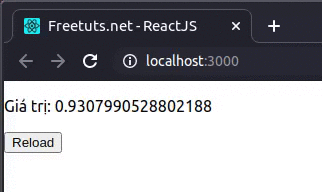
</div>

);

}

}

export default App;



**Find Dom Node**

Hàm này cho phép chúng ta làm việc với DOM trong React, nó không được khuyến khích bởi sẽ làm ảnh hưởng đến các Virtual DOM gây ra lỗi không mong muốn. Bạn cần phải gọi đối tượng ReactDOM trong thư viện react-dom bằng cách:

import ReactDOM from 'react-dom';

Để sử dụng hàm findDOMNode() chúng ta có cú pháp như sau:

ReactDOM.findDOMNode(component)

Giả sử ví dụ bên dưới mình muốn sửa đổi màu chữ của thẻ h1 có id là title thành màu đỏ.

import React from "react";

import ReactDOM from "react-dom";

class App extends React.Component {

changeColor() {

var title = document.getElementById("title");

ReactDOM.findDOMNode(title).style.color = "red";

}

render() {

return (

<div>

<h1 id="title">Tiêu đề</h1>

<button onClick={() => this.changeColor()}>Change Color</button>

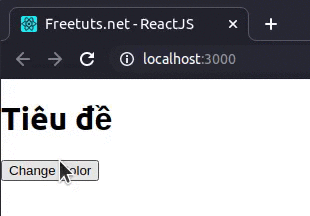
</div>

);

}

}

export default App;



1.6. Tìm hiểu Component Life Cycle trong ReactJS

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về component life cycle trong React, đây là một phần quan trọng trong React, nó cho phép bạn hiểu được cách mà một component họat động ra sao ?

**1.6.1. Component Life Cycle là gì ?**

Chúng ta có thể thấy được mọi thứ trong thế giới đều hoạt động theo một chu kì (ví dụ như con người và cây cối). Cây được mọc lên, sẽ phát triển rồi đến một khoảng thời gian nào đó là sẽ chết đi. Trong React Component cũng vậy, một chu kì cũng xuất hiện, components được khởi tạo (hiển thị ra DOM), update, và kết thúc (unmount),..đó được gọi là một component life cycle.

React cho phép chúng ta tham gia vào các giai đoạn của mỗi component bằng cách sử dụng các phương thức được xây dựng sẵn trong mỗi giai đoạn đó. Khi một components được khởi chạy nó sẽ phải trải qua 4 giai đoạn chính:

initialization

mounting

updating

unmounting

Phần tiếp theo chúng ta sẽ đi vào cách mà bạn có thể tham gia vào giai đoạn trong một components.

**1.6.2. Component Life Cycle**

Chúng ta sẽ tìm hiểu về các lifcecycle methods có trong mỗi giai đoạn.

**Initialization**

Đây là giai đoạn mà thành phần sẽ bắt đầu hành trình của mình bằng cách khởi tạo state và props. Điều này thường được thực hiện bên trong phương thức constructor. Ở đây mình có ví dụ:

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

website: 'Học ReactJS cùng Freetuts.net'

};

}

}

Ở giai đoạn này, React Component sẽ tiến hành khởi tạo các state, props hay các câu lệnh được khởi tạo trong constructor(),...

**Mounting**

Giai đoạn này được thực hiện sau khi quá trình initialization(khởi tạo) được hoàn thành. Nó thực hiện nhiệm vụ chuyển virtual DOM (DOM ảo) trong React thành DOM và hiển thị trên trình duyệt. Component sẽ được render lần đầu tiên, ở đây chúng ta có 3 phương thức để tham gia vào giai đoạn này.

**componentWillMount()**

Được khởi chạy khi một component chuẩn bị được mount (tức là trước khi thực hiện render), sau khi thực hiện xong componentWillMount() thì component mới có thể được mount.

Lưu ý: Chúng ta không nên thực hiện bất cứ thay đổi nào liên quan đến state, props hay call API ở trong hàm này, bởi thời gian chuẩn bị mount -> mount rất ngắn nên các tác vụ đó không thể hoàn thành kịp. Khiến cho component render ra kết quả không như bạn mong muốn.

**componentDidMount()**

Được gọi khi component đã được mount (render thành công ), quá trình này xảy ra sau khi componentWillMount() thực hiện xong. Trong phương thức này bạn có thể gọi API, thay đổi state, props.

Ở đây mình có ví dụ về 2 phương thức mà mình vừa để cập :

class LifeCycle extends React.Component {

componentWillMount() {

console.log('Component will mount!')

}

componentDidMount() {

console.log('Component did mount!')

this.getList();

}

getList=()=>{

/\*\*\* Gọi API, update state,vv...\*\*\*/

}

render() {

return (

<div>

<h3>Mouting Method</h3>

</div>

);

}

}

**Updating**

Đây là giai đoạn thứ ba mà các component phải thực hiện, sau giai đoạn initialization (khởi tạo ) , mount (render lần đầu),... . Trong giai đoạn này, dữ liệu của các phần (props & state) sẽ được cập nhật để đáp ứng với các sự kiện của người dùng như click, gõ, v.v. Điều này dẫn đến việc re-render lại component, ở trong giai đoạn này chúng ta sẽ có 4 phương thức chính:

**shouldComponentUpdate()**

Phương thức này xác định xem component có nên được render lại hay không ? Theo mặc định, nó trả về true. Nhưng bạn có thể thay đổi giá trị trả về của nó theo từng trường hợp. Nó sẽ nhận về 2 tham số truyền vào là nextState và nextProps.

**componentWillUpdate()**

Phương thức này được gọi trước khi tiến hành re-render, bạn có thể thực hiện các hành động như update state, props,...trong phương thức này trước khi tiến hành re-render. Giống như shouldComponentUpdate(), componentWillUpdate() sẽ nhận vào 2 tham số đó là nextState và nextProps

ComponentDidUpdate()

Phương thức này được gọi khi component đã re-render xong. Chúng ta có ví dụ về cả 3 phương thức về đề cập ở trên.

class LifeCycle extends React.Component {

constructor(props)

{

super(props);

this.state = {

date : new Date(),

clickedStatus: false,

list:[]

};

}

componentWillMount() {

console.log('Component will mount!')

}

componentDidMount() {

console.log('Component did mount!')

this.getList();

}

getList=()=>{

/\*\*\* method to make api call\*\*\*/

fetch('https://api.mydomain.com')

.then(response => response.json())

.then(data => this.setState({ list:data }));

}

shouldComponentUpdate(nextProps, nextState){

return this.state.list!==nextState.list

}

componentWillUpdate(nextProps, nextState) {

console.log('Component will update!');

}

componentDidUpdate(prevProps, prevState) {

console.log('Component did update!')

}

render() {

return (

<div>

<h3>Mounting Lifecycle Methods</h3>

</div>

);

}

}

**Unmounting**

Đây là bước cuối cùng trong mỗi component, khi tất cả các tác vụ hoàn thành và bạn tiến hành unmount DOM. Quá trình này chỉ có duy nhất 1 phương thức đó là

**componentWillUnmount() :**

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

website: 'Học ReactJS'

};

}

componentWillUnmount() {

console.log('component will unmount')

}

render() {

return (

<div>

<p>Component LifeCycle</p>

</div>

);

}

}

Đây là bước cuối cùng và sẽ kết thúc một vòng đời của components.

Trên đây chúng ta đã cùng nhau đi tìm hiểu về Component Life Cycle trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

***1.7. Handling Events (xử lý sự kiện) trong ReactJS***

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về Handling Events trong Reactjs - xử lý sự kiện trong ReactJS. Trong một website việc tương tác giữa người dùng là điều không thể thiếu như click, nhập form,..chúng ta có thể thực hiện bắt các sự kiện này trong React một cách dễ dàng.

**Handling Events**

Xử lý các sự kiện trong React rất giống với xử lý các sự kiện trên các phần tử DOM. Có một số khác biệt về cú pháp:

Các sự kiện React được đặt tên bằng camelCase, thay vì chữ thường. Ví dụ: onclick -> onClick, onchange -> onChange

Với JSX, bạn truyền một hàm để bắt sự kiện, thay vì một chuỗi như HTML thông thường.

Ở đây mình có ví dụ với HTML

<button onclick="changeName()">

Change Name

</button>

khi làm việc với JSX trong React chúng ta sẽ phải viết như sau :

<button onClick={changeName}>

Change Name

</button>

Một điểm khác biệt nữa là bạn không thể sử dụng return false để chặn các hành động mặc định được (prevent default), trong React bạn cần phải sử dụng preventDefault(). Giả sử trong HTML mình muốn dừng hành động mặc định của một form:

<form onsubmit="console.log('form submit'); return false;">

<div className="form-group">

<label htmlFor="text">Email:</label>

<input type="text" className="form-control" name="email" placeholder="Enter email" onChange={e=>

this.changeInputValue(e)}

/>

</div>

<button type="submit" className="btn btn-primary">

Submit

</button>

</div>

</form>

trong React, chúng ta cần phải sử dụng e.preventDefault():

function ActionLink() {

function submitForm(e) {

e.preventDefault();

console.log('form submit !!');

}

return (

<form onSubmit = {(e) => submitForm(e)}>

<button type="submit">Submit</button>

</form>

);

}

Ở đây, e là một object chứa tất cá event. React định nghĩa object này theo W3C spec, vì vậy chúng ta không cần phải quan tâm về khả năng tương thích giữa các trình duyệt với nhau.

**Một số điểm chú ý:**

***Sự kiện không truyền tham số :***

import React, { Component } from 'react';

import './App.css';

import Product from './components/Product';

class App extends Component {

onClickMe() {

alert('hello world cup');

}

render() {

return (

<div>

<button type="button" className="btn btn-default" onClick={ this.onClickMe }>

Click Me !

</button>

</div>

);

}

}

export default App;

Sau đó bạn mở cmd (hoặc terminal) lên vào thư mục đang chứa project chạy lện npm start và Click thử vào button "Click Me !" thì bạn sẽ nhận được một cái popup với nội dung là "hello world cup".

Vấn đề đặt ra là cái "onClick" kia lấy ở đâu ra thì bạn có thể truy cập vào trang này thì tại đây reacjs có hỗ trợ chúng ta rất là nhiều event như: onChange, onSubmit...

**Sự kiện truyền có thêm tham số :**

Cách 1: Sử dụng hàm bind để truyền tham số :

Ví dụ 1: bind tham số trực tiếp ở hàm:

import React, { Component } from 'react';

class App extends Component {

onClickMe (text) {

alert(text);

}

render() {

return (

<div>

<button type="button" className="btn btn-default" onClick={ this.onClickMe.bind(this, 'Ahihi') }>

Click Me !

</button>

</div>

);

}

}

export default App;

Ví dụ 2: bind tham số ở trên hàm constructor:

import React, { Component } from 'react';

class App extends Component {

constructor(props) {

super(props);

this.onClickMe = this.onClickMe.bind(this, 'Ahihi')

}

onClickMe (text) {

alert(text);

}

render() {

return (

<div>

<button type="button" className="btn btn-default" onClick={ this.onClickMe }>

Click Me !

</button>

</div>

);

}

}

Khi chạy lên cả hai ví dụ trên và click vào button thì đều có cùng kết quả là "Ahihi".

Cách 2: Sử dụng arrow function trong Es6

import React, { Component } from 'react';

import './App.css';

import Product from './components/Product';

class App extends Component {

onClickMe (text) {

alert(text);

}

render() {

return (

<div>

<button type="button" className="btn btn-default" onClick={ () => this.onClickMe('Ahihi') }>

Click Me !

</button>

</div>

);

}

}

**Lưu ý với this trong xử lý Events**

Ở đây mình có một ví dụ, khi click vào button tương ứng sẽ thực hiện ẩn/hiển nội dung:

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

this.state = {

isShow: true

};

}

toggleMSG() {

this.setState({

isShow: !this.state.isShow

});

}

render() {

return (

<div>

<b>Nội dung : {this.state.isShow ? "Freetuts.net - Lập trình ReactJS" : ""}</b> <br />

<button onClick={() => this.toggleMSG()}>

{this.state.isShow ? "HIDE" : "SHOW"}

</button>

</div>

);

}

}

export default App;

Trong sự kiện onClick chúng ta thực hiện gọi hàm toggleMSG(), hãy cẩn thận trong khi gọi hàm này, bạn cần phải truyền vào đó biến this, bằng các cách viết như sau:

//Sử dụng arrow function

<button onClick={() => this.toggleMSG()}>

//Sử dụng bind

<button onClick={this.toggleMSG.bind(this)}>

//hoặc bind this trong constructor()

**Xây dựng ví dụ**

Trong phần này mình sẽ đi xây dựng ví dụ về bắt sự kiện trong React và một vài cách gọi hàm trong React. Trong file src/App.js chúng ta có ví dụ:

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

//Khởi tạo inital state

this.state = {

textareaChange: "",

buttonClick : "",

mouseOver: ""

};

//bind this cho function mouseOver để tránh gặp lỗi

//không tồn tại biến this.

this.mouseOver = this.mouseOver.bind(this)

}

changeText(e) {

this.setState({

textareaChange: e.target.value

});

}

mouseOver() {

this.setState({

mouseOver: this.state.mouseOver + "mouseOver..."

});

}

render() {

return (

<div style={{marginTop: "5%"}}>

<button

onClick={() => {

this.setState({

buttonClick: this.state.buttonClick + "onClick..."

});

}}

>

Click me..

</button>{" "}

<p>

button: <code>{this.state.buttonClick}</code>

</p>

<hr />

<textarea onChange={e => this.changeText(e)} placeholder="nhập cái gì đó..."></textarea>

<p>

textarea: <code>{this.state.textareaChange}</code>

</p>

<hr />

<h6 onMouseOver={this.mouseOver}>Di chuột vào đêyyyy nè :v</h6>

<p>

mouseOver: <code>{this.state.mouseOver}</code>

</p>

</div>

);

}

}

export default App;

Bên trên mình xây dựng 2 hàm changeText() và mouseOver(), khi sự kiện được gọi nó sẽ gọi hàm tương ứng. Trong sự kiện onClick mình thực hiện viết câu lệnh trực tiếp mà không cần gọi hàm.

**Bài 4: Một số khái niệm khác trong React**

4.1 Render với điều kiện trong ReactJS

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về cách để render với điều kiện trong ReacJS. Trong React, chúng ta có thể tạo ra các component riêng biệt, và chỉ định thành phần nào được render bằng cách sử dụng biểu thức điều kiện.

Render kèm điều kiện trong React, hoạt động tương tự như Javascript thông thường, bạn chỉ cần sử dụng biểu thức điều kiện if, hay conditional operator trong Javascript, sau đó React sẽ thực hiện nhiệm vụ kiểm tra điều kiện và render theo trường hợp yêu cầu. Ở đây mình có ví dụ về kiểm tra đăng nhập của người dùng:

import React, { Component } from 'react';

export default class App extends Component {

render() {

var isLogin = true

if(isLogin) {

return (

<div>

<h3>Freetuts.net</h3>

</div>

);

}else{

return(

<div>

<h3>Vui lòng đăng nhập</h3>

</div>

)

}

}

}

Dưới đây, là một vài cách mà bạn có thể thêm điều kiện render trong các component.

**Gán element vào biến**

Chúng ta có thể lưu các element vào một biển. Đây là cách mà bạn có thể tùy chọn các element được hiển thị bằng cách thêm điều kiện vào cho nó. Giả sử, mình sẽ viết lại ví dụ ở trên bằng cách gán element vào biến.

import React, { Component } from "react";

export default class App extends Component {

render() {

const isLogin = true;

if (isLogin) {

//Gán element vào một biến

var notification = <h3>Freetuts.net</h3>;

} else {

//Gán element vào một biến

var notification = <h3>Vui lòng đăng nhập</h3>;

}

return <div>{notification}</div>;

}

}

Chúng ta sẽ nhận được kết quả giống như ví dụ ở đầu bài, ngoài ra bạn còn có thể sử dụng cú pháp conditional operator, ở đây mình có ví dụ :

import React, { Component } from "react";

export default class App extends Component {

render() {

const isLogin = true;

//Conditional operator

const notification = (isLogin) ? <h3>Freetuts.net</h3> : <h3>Vui lòng đăng nhập</h3>

return <div>{notification}</div>;

}

}

**Biểu thức điều kiện trong JSX**

Ngoài cách sử dụng các biểu thức điều kiện bên ngoài hàm return() như ví dụ bên trên, chúng ta còn có thể sử dụng nó trong JSX bằng cách viết nó trong dấu ngoặc {}, chúng ta vẫn sẽ sử dụng ví dụ ở đầu bài:

import React, { Component } from "react";

export default class App extends Component {

render() {

const isLogin = true;

return (

<div>

{isLogin ? (<h3>Freetuts.net</h3>) : (<h3>Vui lòng đăng nhập</h3>)}

</div>

);

}

}

Trong JSX bạn chỉ có thể sử dụng cú pháp conditional operator, kết quả hiển thị của ví dụ vừa rồi cũng có kết quả tương tự với ví dụ đầu tiên.

**Ngăn chặn component render**

Trong một số trường hợp nào đó, bạn sẽ muốn một component tự ẩn đi dù nó được render trong một component khác. Để làm được điều đó, ta sẽ trả về null thay vì trả về JSX.

const Demo = (props) => {

//Return về null để không hiển thị

return null

}

Giả sử mình có một component hiển thị các thông báo:

import React, { Component } from "react";

const Notification = (props) => {

//Kiểm tra giá trị của props

if(props.isShow) {

//Trả về JSX để hiển thị

return (

<ul>

<li>Thông báo 1</li>

<li>Thông báo 2</li>

</ul>

)

}else{

//Trả về null để ẩn

return null

}

}

export default class App extends Component {

constructor(props) {

super(props);

this.state = {

isShowNotification: false

};

}

render() {

const {isShowNotification} = this.state

return (

<div style={{margin: 20}}>

<p>freetuts.net - lập trình ReactJS</p>

<button onClick = {() => {

//Cập nhật lại state

this.setState({

isShowNotification: !isShowNotification

})

}}>{isShowNotification ? 'Hiển thị' : 'Ẩn'}</button>

{/\* Gọi component Notification \*/}

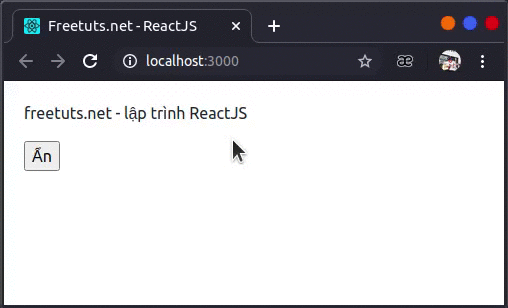
<Notification isShow = {isShowNotification}/>

</div>

);

}

}



Trên đây chúng ta đã cùng nhau đi tìm hiểu về Render với điều kiện trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

4.2 Tìm hiểu về List và Keys trong ReactJS

Trong bài viết này chúng ta sẽ cùng hau đi tìm hiểu về lists và keys trong ReactJS, đây cũng là phần mà chúng ta sẽ làm việc rất nhiều trong React.

**Lists trong React**

Việc khởi tạo các lists trong React, tương tự như khởi tạo lists trong Javascript. Ở đây mình sẽ tiến hành khởi tạo một lists các items.

import React from "react";

function ListComponent(props) {

const myList = ["php", "javascript", "python", "C++"];

const listItems = myList.map((item) =>

<li>{item}</li>

);

return (

<ul>{listItems}</ul>

);

}

export default ListComponent

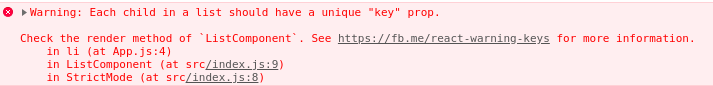
và trình duyệt sẽ hiện thị kết quả:

* php
* javascript
* python
* C++

Việc khởi tạo các lists trong React rất đơn giản, điều mình muốn tập chung nhấn mạnh trong bài viết này đó là về keys mà mình sẽ đề cập bên dưới.

**Keys trong React**

Trong quá trình làm việc với React, chúng ta phải thao tác với danh sách(lists) rất nhiều như danh sách các ảnh, danh sách các item trong giỏ hàng,...Khi các lists này có hàng tá các items thì React rất khó có thể kiểm soát được items. Bởi vậy chúng ta cần phải chỉ định cho nó một key để định danh.

Nếu bạn chạy ví dụ ở phần thứ 1, React sẽ hiển thị cảnh báo như hình bên dưới: 

Để loại bỏ cảnh báo bạn phải chỉ định cho các items trong lists một thuộc tính có tên là key. Chúng ta sẽ sửa ví dụ ở đầu bài thành:

import React from "react";

function ListComponent(props) {

const myList = [

{

id : 'p',

name : 'php'

},

{

id : 'j',

name : 'javascript'

},

{

id : 'py',

name : 'python'

},

{

id : 'c',

name : 'C++'

},

]

//Thêm thuộc tính key vào trong thẻ jsx

const listItems = myList.map((item) =>

<li key = {item.id}>{item.name}</li>

);

return (

<ul>{listItems}</ul>

);

}

export default ListComponent

Kết quả của ví dụ vẫn tương tự, nhưng chúng ta đã hoàn toàn loại được cảnh báo. React khuyên chúng ta nên chỉ định các key duy nhất trong các lists.

**Một vài lưu ý khi sử dụng Keys**

Ở đây mình có một vài lưu ý sử dụng key cho list, các lưu ý này sẽ giúp quá trình làm việc với React không gặp các lỗi không mong muốn.

Keys là duy nhất

Bạn cần chỉ định các keys này là duy nhất, các keys này không được trùng lặp trong các lists.

const myList = [

{

id : 'p',

name : 'php'

},

{

id : 'j',

name : 'javascript'

},

{

id : 'p',

name : 'python'

},

{

id : 'c',

name : 'C++'

},

]

const listItems = myList.map((item) =>

<li key = {item.id}>{item.name}</li>

);

khi các keys này trùng lặp bạn sẽ nhận được cảnh báo :

*Warning: Encountered two children with the same key, `p`.*

*Keys should be unique so that components maintain their identity across updates.*

*Non-unique keys may cause children to be duplicated and/or omitted — the behavior is unsupported and could change in a future version.*

Các keys chỉ cần là duy nhất khi so sánh với các anh/chị của nó trong lists chứa chúng.

Tránh chỉ định index làm key

Trong một vài trường hợp bạn thường chỉ định giá trị của biến index thành keys như trong ví dú này:

const listItems = myList.map((item, index) =>

<li key = {index}>{item.name}</li>

);

React khuyên chúng ta không nên sử dụng cách này. Bởi khi bạn thực hiện sắp xếp mảng thì index sẽ thay đổi, React lại phải xác định lại keys môt lần nữa, gây ra giảm hiệu xuất làm việc.

Chỉ sử dụng index làm key trong khi:

* Nếu list của bạn là tĩnh và sẽ không thay đổi.
* List sẽ không bao giờ được sắp xếp lại.
* List sẽ không được lọc (thêm / xóa các mục khỏi danh sách).
* Không có id cho các mục trong list.

Hãy chỉ sử dụng index làm key trong trường hợp đặc biệt này, và lưu ý trong quá trình sử dụng.

Trên đây chúng ta đã cùng nhau đi tìm hiểu về List và Keys trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

4.3. Kỹ thuật Lifting State Up trong ReactJS

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về Lifting State Up trong ReactJS, trong quá trình làm việc với React, các component phải re-render và thay đổi rất nhiều lần. Trong một vài trường hợp, bạn muốn khi 1 component con được thay đổi, đồng thời bạn muốn component cha cũng sẽ bắt được hành động đó. Theo tài liệu chính thức của ReactJS thì Lifting State Up được định nghĩa là rằng :

*several components need to reflect the same changing data. We recommend lifting the shared state up to their closest common ancestor*

chúng ta có thể hiểu rằng

*một vài component phải cùng nhận được sự thay đổi. Chúng tôi khuyên bạn nên chia sẻ các state cho cha, mẹ, ông bà dòng họ của chúng.*

Đọc đến đây chắc hẳn bạn cũng sẽ hiểu một phần, kỹ thuật lifting state up là cách mà có thể chia sẻ dữ liệu cho các component khác. Có nghĩa là khi bạn thay đổi dữ liêu của 1 component con thì bạn sẽ gửi dữ liệu cho component cha biết.

**Lifting State Up trong React**

Trong phần này mình sẽ chỉ ra các bước thực hiện Lifting State Up, các bước thức hiện rất đơn giản và chúng ta sẽ có ví dụ cụ thẻ ở phần tiếp theo.

Trước tiên, chúng ta sẽ truyền cho component con một props có giá trị là một function. Function này sẽ được gọi khi component con trả về dữ liệu:

function ComponentCha(props) {

//Hàm này sẽ được gọi khi nhận được dữ liệu

const receiveData = function (data) {

console.log('Data nhận được', data )

}

//Gọi compoennt con và truyền vào một props

//có giá trị là một hàm

return (

<ComponentCon onReceiveData = {receiveData}/>

)

}

Tiếp theo, ở componentCon chúng ta sẽ tiến hành gửi lại cho componentCha bằng cách truyền vào props onReceiveData giá trị cần gửi về.

function ComponentCon (props) {

return(

<div>

<button onClick={() => {

// Chúng ta sẽ gọi props có tên là

// receiveData đã được truyền từ compoentCha.

// Và truyền vào đó giá trị cần gửi

props.onReceiveData('data gửi đi')

}}>Gửi lại cho componentCha</button>

</div>

)

}

Vậy là chúng ta đã gửi thành công dữ liệu từ componentCon sang componenetCha. Lúc này componentCha có thể sử dụng dữ liệu đó. Đây là các bước cơ bản để sử dụng Lifting State Up. Chúng ta sẽ đi vào ví dụ cụ thể bên dưới.

**Ví dụ cụ thể**

Ở đây chúng ta sẽ đi vào ví dụ cụ thể về cách sử dụng kỹ thuật Lifting State Up. Mình sẽ đi xây dựng một ứng dụng cho phép chuyển đổi giữa USD và VNĐ. Chúng ta sẽ làm việc với file App.js

Trước tiên, chúng ta sẽ đi xây dựng component cha có tên Caculator, và hàm handleChange chịu trách nhận data từ component con.

export default class App extends Component {

constructor(props) {

super(props);

this.state = {

usd: 0,

vnd: 0,

};

}

handleChange = (data) => {

this.setState(data);

};

render() {

return (

<div style={{margin: "3%"}}>

<USDtoVND onHandleChange={this.handleChange} value={this.state.usd} />

<VNDtoUSD onHandleChange={this.handleChange} value={this.state.vnd} />

<hr />

<code>freetuts.net</code>

</div>

);

}

}

Mình sẽ truyền cho 2 component con có tên USDtoVND và VNDtoUSD các props như:

onHandleChange: giúp cho component con có thể gửi dữ liệu về component cha.

value: giá trị của input.

Tiếp theo, chúng ta sẽ khởi tạo các component con có tên USDtoVND và VNDtoUSD:

const USDtoVND = function (props) {

const convert = function (usd) {

return usd \* 23632;

};

return (

<div>

<span>USD </span>

<input

onChange={(e) => {

const usd = e.target.value;

const vnd = convert(usd);

props.onHandleChange({

usd,

vnd,

});

}}

value={props.value}

/>

</div>

);

};

const VNDtoUSD = function (props) {

const convert = function (vnd) {

return vnd / 23632;

};

return (

<div>

<span>VND </span>

<input

onChange={(e) => {

const vnd = e.target.value;

const usd = convert(vnd);

props.onHandleChange({

usd,

vnd,

});

}}

value={props.value}

/>

</div>

);

};

các component này có nhiệm vụ chuyển từ đổi tiền và trả về cho component cha bằng prop onHandleChange().

Cuối cùng, chúng ta sẽ có file App.js đầy đủ như sau:

import React, { Component } from "react";

const USDtoVND = function (props) {

const convert = function (usd) {

return usd \* 23632;

};

return (

<div>

<span>USD </span>

<input

onChange={(e) => {

const usd = e.target.value;

const vnd = convert(usd);

props.onHandleChange({

usd,

vnd,

});

}}

value={props.value}

/>

</div>

);

};

const VNDtoUSD = function (props) {

const convert = function (vnd) {

return vnd / 23632;

};

return (

<div>

<span>VND </span>

<input

onChange={(e) => {

const vnd = e.target.value;

const usd = convert(vnd);

props.onHandleChange({

usd,

vnd,

});

}}

value={props.value}

/>

</div>

);

};

export default class App extends Component {

constructor(props) {

super(props);

this.state = {

usd: 0,

vnd: 0,

};

}

handleChange = (data) => {

this.setState(data);

};

render() {

return (

<div style={{margin: "3%"}}>

<USDtoVND onHandleChange={this.handleChange} value={this.state.usd} />

<VNDtoUSD onHandleChange={this.handleChange} value={this.state.vnd} />

<hr />

<code>freetuts.net</code>

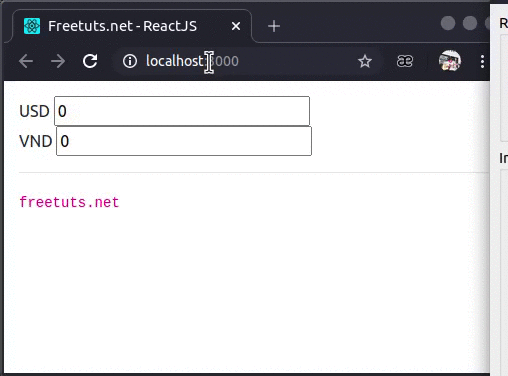
</div>

);

}

}

Kết quả:



Kỹ thuật Lifting State Up chỉ có thể sử dụng khi bạn truyền từ component cha sang con và ngược lại. Tức là bạn chỉ có thể truyền dữ liệu từ 1 component quan hệ cha con. Trong trường hợp, bạn muốn chia sẻ dữ liệu state sang component khác, chúng ta sẽ có khái niệm về Redux, cho phép chúng ta tạo ra local state, giúp việc chia sẻ dữ liệu giữa các component được dễ dàng hơn.

Trên đây chúng ta đã cùng nhau đi tìm hiểu về kỹ thuật Lifting State Up trong ReactJS Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

4.4 Tìm hiểu về Context trong ReactJS

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về Context trong ReactJS. Trong quá trình làm việc với ReactJS, các dữ liệu trong các component phải được chia sẻ với nhau.

Chúng ta thực hiện điều này bằng cách đưa các dữ liệu này lên một nơi có tên là local state. Local state chịu trách nhiệm phân phối dữ liệu tới các component. Context hỗ trợ chúng ta thực hiện điều này một cách đơn giản.

**Context trong ReactJS là gì ?**

Theo tài liệu chính thức của ReactJS định nghĩa Context là :

*Context provides a way to pass data through the component tree without having to pass props down manually at every level.*

Chúng ta có thể hiểu:

*Context cung cấp cho chúng ta cách để thực hiện chia sẻ dữ liệu tới các component trong cây mà không cần truyền dữ liệu qua props theo từng cấp bậc.*

Khi truyền dữ liệu tới các component bằng props thì bạn chỉ có thể truyền từ component cha sang component con. Nếu bạn muốn truyền sang component cháu hoặc sang component họ hàng xa thì điều này rất phức tạp. Bởi vậy context sẽ là kênh giao tiếp cho các component, cho phép bạn truyền dữ liệu một cách đơn giản hơn rất nhiều.

Ở đây mình có một ví dụ, mình muốn chuyển lời nhắn từ ComponetÔng sang Componet Cháu :

Khi sử dụng truyền dữ liệu qua props thì mình bắt buộc phải gửi lời nhắn qua (Ông -> Cha -> Cháu )

import React from "react";

const ComponentChau = (props) => {

return <h1>Ông bảo là "{props.message}"</h1>;

};

const ComponentCha = (props) => {

return <ComponentChau {...props} />;

};

const ComponentOng = () => {

const message = "Vào freetuts.net học lập trình";

return <ComponentCha message={message} />;

};

export default ComponentOng;

Còn nếu sử dụng Context thì chúng ta sẽ gửi trực tiếp từ Ông đến Cháu luôn :

import React from "react";

const MessageContext = React.createContext();

class ComponentChau extends React.Component {

render() {

return <h1>Ông bảo là : "{this.context}"</h1>;

}

}

ComponentChau.contextType = MessageContext;

const ComponentOng = () => {

return (

<MessageContext.Provider value="Vào freetuts.net học lập trình">

<ComponentChau />

</MessageContext.Provider>

);

};

export default ComponentOng;

Kết quả của 2 cách này vẫn tương tự nhưng việc sử dụng Context sẽ đơn giản hơn nhiều.

**Context API trong ReactJS**

Trước tiên, chúng ta sẽ đi tìm hiểu về Context API, sau đó sẽ đi vào ví dụ cụ thể nhé. Ở đây chúng ta có một vài API mà React cung cấp.

***React.createContext***

const MyContext = React.createContext(defaultValue);

Khởi tạo một Context Object, giá trị của defaultValue là giá trị mặc định của props value trong Provider.

***Context.Provider***

<MyContext.Provider value={/\* some value \*/}>

Mỗi Context Object phải đi kèm với một Provider, nó cho phép bạn nhận về sự thay đổi của context.

***Context.Consumer***

<MyContext.Consumer>

{value => /\* render something based on the context value \*/}

</MyContext.Consumer>

Một React component được khởi chạy mỗi khi gía trị của context thay đổi, và nhận về giá trị của context đó.

***Context.displayName***

const MyContext = React.createContext(/\* some value \*/);

MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools

<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools

Đặt tên cho Context, tên này sẽ được hiện thị trong DevTools.

**Class.contextType**

class MyClass extends React.Component {

render() {

let value = this.context;

}

}

MyClass.contextType = MyContext;

contextType là một thuộc tính của class được tạo bởi React.createContext() được dùng để lấy giá trị của context.

**Sử dụng Context trong ReactJS**

Chúng ta sẽ sử dụng context theo 3 bước cụ thể:

1. Khởi tạo object context bằng phương thức React.createContext(), sau đó chúng ta sẽ nhận được 1 object bao gồm các thuộc tính quan trọng như Provider và Consumer.
2. Sử dụng Provider bọc quanh các component, và truyền giá trị vào props value
3. Thêm Consumer vào bất cứ đâu mà bạn muốn chia sẻ context miễn là ở bên trong Provider, bạn có thể lấy gía trị của context thông qua props.chirlden.

Bây giờ chúng ta sẽ đi vào ví dụ cụ thể. Mình sẽ xây dựng 1 ứng dụng cho phép random các số và hiển thị. Trước tiên, mình sẽ tiến hành khởi tạo một object context.

import React from "react";

//Khởi tạo môt

const NumberContext = React.createContext();

Sau đó, mình sẽ bọc quanh các component cần chia sẻ bằng Provider, và Consumber :

export default class ContextComponent extends React.Component {

constructor(props) {

super(props);

this.state = {

number: 0,

};

}

updateNumber = () => {

this.setState({

number: Math.random(),

});

};

render() {

return (

<NumberContext.Provider

value={{

number: this.state.number,

update: this.updateNumber.bind(this),

}}

>

<NumberContext.Consumer>

{ () => (

<>

<ShowNumber />

<UpdateNumber />

</>

)}

</NumberContext.Consumer>

</NumberContext.Provider>

);

}

}

bên trên mình truyền vào props value trong provider giá trị là một object có các thuộc tính như:

number: Chứa giá trị của số.

update: hàm thực hiện update số.

Consumer sẽ có nhiệm vụ xem xét sự thay đổi và trả về giá trị của context trong props chirlden, mình sẽ chia sẻ props cho 2 component đó là ShowNumber và UpdateNumber.

Tiếp theo, mình sẽ đi xây dựng 2 component là ShowNumber và UpdateNumber mà mình đã gọi ở Consumer.

class UpdateNumber extends React.Component {

render() {

return (

<button onClick={() => {

//Gọi hàm update để thực hiện update số.

console.log(this.context.update())

}}>Update Number</button>

);

}

}

UpdateNumber.contextType = NumberContext;

class ShowNumber extends React.Component {

render() {

//Hiển thị ra số.

return (

<p>{this.context.number}</p>

);

}

}

ShowNumber.contextType = NumberContext;

Ở đây mình sử dụng phương thức contextType để gán giá trị của context, lúc này thuộc tính this.context trong class sẽ chứa giá trị của context.

Cuối cùng, chúng ta sẽ có file đầy đủ:

import React from "react";

//Khởi tạo môt

const NumberContext = React.createContext();

class UpdateNumber extends React.Component {

render() {

return (

<button onClick={() => {

console.log(this.context.update())

}}>Update Number</button>

);

}

}

UpdateNumber.contextType = NumberContext;

class ShowNumber extends React.Component {

render() {

return (

<p>{this.context.number}</p>

);

}

}

ShowNumber.contextType = NumberContext;

export default class ContextComponent extends React.Component {

constructor(props) {

super(props);

this.state = {

number: 0,

};

}

updateNumber = () => {

this.setState({

number: Math.random(),

});

};

render() {

return (

<NumberContext.Provider

value={{

number: this.state.number,

update: this.updateNumber.bind(this),

}}

>

<NumberContext.Consumer>

{ () => (

<>

<ShowNumber />

<UpdateNumber />

</>

)}

</NumberContext.Consumer>

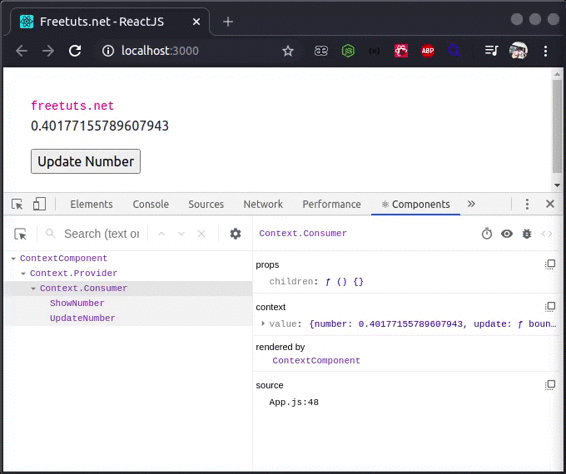
</NumberContext.Provider>

);

}

}

khởi chạy ứng dụng và xem kết quả:



Trên đây chúng ta đã cùng nhau đi tìm hiểu về Context trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

Chú ý: https://react-typescript-cheatsheet.netlify.app/docs/basic/getting-started/context/

4.5 Tìm hiểu về Fragments trong ReactJS

Trong bài này chúng ta sẽ cùng nhau đi tìm hiểu về Fragments trong ReactJS, chắc hẳn trong quá trình làm quen với React bạn gặp thông báo lỗi " JSX parent expressions must have one parent element” khi không bọc các element JSX quanh một element nào đó. Giả sử như thế này.

//Khai báo JSX sai.

return (

<h1>Hello, freetuts.net !</h1>

<p>Học lập trình ReactJS cùng Freetuts.net</p>

)

lúc này chúng ta sẽ sử dụng fragments trong React để loại bỏ lỗi.

return (

<React.Fragment>

<h1>Hello, freetuts.net !</h1>

<p>Học lập trình ReactJS cùng Freetuts.net</p>

</React.Fragment>

);

Bây giờ, chúng ta sẽ đi tìm hiểu sâu hơn về Fragments trong ReactJS, và cách nó được sử dụng như thế nào ?

**Tại sao phải sử dụng Fragments**

Thông thuờng chúng ta dùng thẻ div để bao quanh các element JSX như ví dụ:

return (

<div>

<h1>Hello, freetuts.net !</h1>

<p>Học lập trình ReactJS cùng Freetuts.net</p>

</div>

);

Nhưng trong một vài trường hợp việc thêm thẻ div vào sẽ làm đảo lộn quy ước CSS và cấu trúc của các thẻ HTML khi được render. Giả sử trong trường hợp này.

Mình có môt component có tên Table dùng để hiển thị bảng:

class Table extends React.Component {

render() {

return (

<table>

<tr>

<Columns />

</tr>

</table>

);

}

}

và component có tên Columns sẽ hiển thị nội dung của bảng đó. Nếu chúng ta thêm thẻ div để bao quanh JSX trong component Colums như thế này:

class Columns extends React.Component {

render() {

return (

<div>

<td>Hello</td>

<td>World</td>

</div>

);

}

}

Nó sẽ phá vỡ cấu trúc của tables và hiển thị sai, và đây là kết quả khi thực hiện render:

<table>

<tr>

<div>

<td>Hello</td>

<td>World</td>

</div>

</tr>

</table>

Lúc này bạn cần đến việc sử dụng fragments.

**Sử dụng fragments trong ReactJS**

Fragments cho phép chúng ta bọc các element JSX lại, giúp bạn triển khai các element HTML theo mong muốn. Chúng ta sẽ tiếp tục với ví dụ ở phần thứ nhất, component Columns sẽ được chỉnh sửa như sau :

class Columns extends React.Component {

render() {

return (

<React.Fragment>

<td>Hello</td>

<td>World</td>

</React.Fragment>

);

}

}

và khi component được render nó sẽ hiển thị theo cấu trúc bạn mong muốn.

<table>

<tr>

<td>Hello</td>

<td>World</td>

</tr>

</table>

**Cú pháp của Fragments**

Ở đây chúng ta có thể viết các fragments theo 2 cách, mỗi cách viết có những ưu điểm khác nhau.

**React.Fragments**

Đây là cách viết đầy đủ, nó cho phép bạn có thể thêm các key vào khi triển khai các lists. Bạn có thể tham khảo thêm về lists và keys trong ReactJS. Ở đây mình có một ví dụ, sẽ hiển thị ra một list :

function Glossary(props) {

return (

<dl>

{props.items.map(item => (

// Bạn phải chỉ định một keys cho mỗi items

// trong lists

<React.Fragment key={item.id}>

<dt>{item.term}</dt>

<dd>{item.description}</dd>

</React.Fragment>

))}

</dl>

);

}

**Viết tắt**

Bạn có thể viết cú pháp ngắn gọn của fragments bằng cách sử dụng 2 dấu ngoặc nhọn ( <> )

return (

<>

<p>freetuts.net</p>

<code>

freetuts.net

</code>

</>

);

khi sử dụng cú pháp ngắn gọn này bạn không thể chỉ định keys vào fragments. Trong trường hợp bạn muốn thêm key thì chúng ta sẽ dùng React.Fragments.

Trên đây chúng ta đã cùng nhau đi tìm hiểu về Fragments trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

4.6. Gọi API và xử lý dữ liệu trong React

React nổi tiếng với việc xây dựng các giao diện người dùng phong phú và có khả năng mở rộng cao. Có nhiều cách để lấy dữ liệu từ một external API trong React. Trong bài này, chúng ta sẽ thảo luận về các cách khác nhau để quản lý các lệnh gọi API trong React. Cuối cùng, bạn sẽ có thể chọn cách tiếp cận tốt nhất tùy theo request của ứng dụng.

**The Fetch API**

Fetch API được tích hợp vào hầu hết các trình duyệt hiện đại trên window object (window.fetch) và cho phép thực hiện các request HTTP rất dễ dàng. Các đoạn code sau đây cho thấy một ví dụ đơn giản về việc sử dụng Fetch API trong thực tế.

import {useEffect} from "react";

const fetchUsers = () => {

// Where we're fetching data from

return fetch("http://www.abc.cd/test")

// We get the API response and receive data in JSON format

.then((response) => response.json())

.then((data) => console.log(data))

.catch ((error) => console.error(error));

}

Mục tiêu duy nhất của function này là truy cập dữ liệu và convert response sang dạng json, sử dụng method response.json(). Ở đây việc sử dụng method json() để lấy response object được lưu trữ dữ liệu và được update state của users trong ứng dụng. Thực tế là Fetch dựa trên promise-nghĩa là chúng ta có thể catch errors sử dụng method .catch(). Bất kỳ lỗi nào gặp phải đều sử dụng làm giá trị để update state error. Thêm vào đó, ta thực hiện request này trong useEffect() hook với mảng phụ thuộc làm đối số thứ hai để thực hiện request một lần mà không phụ thuộc vào data khác. Dưới đây là ví dụ cách sử dụng useEffect() hook:

import {useEffect} from "react";

useEffect(() => {

fetchData()

}, []);

**Axios Library**

Axios là một ứng dụng HTTP dựa trên Promise cho JavaScript có thể được sử dụng trong ứng dụng front-end và Node.js. Bằng cách sử dụng Axios, thật dễ dàng để gửi các requests HTTP không đồng bộ đến các điểm cuối REST và thực hiện các hoạt động CRUD.

Trước tiên chúng ta phải cài đặt Axios bằng cách sử dụng npm hoặc yarn và sau đó import vào parent component, như sau:

*npm install axios*

Dưới đây là đoạn code nhỏ ví dụ việc sử dụng axios:

import axios from "axios"

const fetchData = () => {

return axios.get("http://www.abc.cd/test")

.then((response) => console.log(response.data));

}

Tương tự như Fetch API, Axios return một promise. Nhưng đối với Axios, nó sẽ return về một Json response.

**Async-Await syntax**

Async / await là một cách tương đối mới để viết code không đồng bộ một cách đồng bộ.

Từ khóa async trước một function có hai tác dụng:

* Make it always return a promise. (Luôn luôn return một promise)
* Allows await to be used in it. (Cho phép await để sử dụng trong đó)

Từ khóa await trước một promise làm cho JavaScript đợi cho đến khi promise đó hoàn thành và sau đó:

* If it’s an error, the exception is generated. (Nếu đó là một lỗi, ngoại lệ sẽ được tạo ra)
* Otherwise, it returns the result. (Nếu không, nó trả về kết quả)

Đoạn code dưới đây sẽ giúp bạn dễ hiểu hơn:

async function fetchData() {

try {

const result = await axios.get("http://www.abc.cd/test")

console.log(result.data));

} catch (error) {

console.error(error);

}

}

Khi chúng ta sử dụng useEffect(), hàm effect (đối số đầu tiên) không thể trở thành một hàm async. Vì vậy, chúng ta có thể tạo một hàm async riêng biệt trong component, hàm này chúng ta có thể gọi sync trong useEffect và lấy dữ liệu cho phù hợp.

**Custom React Hook**

Custom Hook là một hàm JavaScript có tên bắt đầu bằng "use" và có thể gọi các Hook khác. Ý tưởng đằng sau custom hooks là extract component logic thành các functions có thể sử dụng lại. Vì vậy, hãy gọi custom hook: useFetch. Hook này chấp nhận hai đối số, URL mà chúng ta cần query để lấy dữ liệu và một đối tượng đại diện cho các option mà chúng ta muốn áp dụng cho request. Bây giờ, hãy xem việc lấy dữ liệu bằng hook useEffect()của chúng ta dễ dàng như thế nào. Chúng ta sẽ sử dụng Fetch API để thực hiện request của mình. Để làm được điều đó, ta phải pass URL và các options mà chúng tôi muốn nhận. Từ đó, ta nhận được một object có thể sử dụng để render trong ứng dụng của mình.

import { useState, useEffect } from 'react';

const useFetch = (url = 'http://www.abc.cd/test', options = null) => {

const [data, setData] = useState(null);

useEffect(() => {

fetch(url, options)

.then(res => res.json())

.then(data => setData(data));

}, [url, options]);

return {data}

}

export default useFetch;

Chúng ta có thể call bất cứ khi nào ta cần lấy dữ liệu:

import useFetch from './useFetch';

const { data } = useFetch( 'http://www.abc.cd/test' );

console.log(data);

**React Query Library**

React-query là một thư viện tuyệt vời giải quyết các vấn đề quản lý server state và caching trong applications. Nó giúp việc fetching, catching, synchronizing và updating server state trong React applications trở nên dễ dàng hơn. Đầu tiên, hãy cài đặt pakage:

*npm install react-query react-query-devtools*

Note: React Query cũng có các dev tools giúp chúng ta hình dung hoạt động bên trong của React Query.

React-query cung cấp cho chúng ta một bộ nhớ cache, bạn có thể thấy bộ nhớ này bên dưới thông qua React Query Devtools. Điều này cho phép chúng ta dễ dàng quản lý các requests đã thực hiện theo key value mà chúng ta chỉ định cho mỗi request.

import React from "react";

import ReactDOM from "react-dom";

import { QueryClient, QueryClientProvider, useQuery } from "react-query";

import { ReactQueryDevtools } from "react-query/devtools";

const queryClient = new QueryClient();

export default function App() {

return (

<QueryClientProvider client={queryClient}>

<FetchData />

</QueryClientProvider>

);

}

function FetchData() {

const { data } = useQuery("UserData", () =>

fetch("http://www.abc.cd/test").then((res) => res.json())

);

return (

<div>

// data you want to show

<ReactQueryDevtools initialIsOpen />

</div>

);

}

const rootElement = document.getElementById("root");

ReactDOM.render(<App />, rootElement);

Tóm lại, chúng ta chỉ cần cho thư viện biết nơi bạn cần fetch dữ liệu và nó sẽ handle caching, background updates, và refresh data mà không cần thêm bất kỳ code hoặc configuration nào. Nó cũng cung cấp một số hooks hoặc events cho mutation và queries để handle error và các states khác của side effects, loại bỏ nhu cầu sử dụng các hook useState () và useEffect () và thay thế chúng bằng một vài dòng logic React Query. Để hiểu rõ hơn, hãy xem tài liệu gốc. Có nhiều cách khác để quản lý việc lấy dữ liệu như SWR, GraphQL API.

Tìm hiểu chi tiết tại đây:

1. <https://viblo.asia/p/reactjs-react-query-la-gi-tai-sao-nen-dung-react-query-Rk74a2GMJeO>
2. <https://codesandbox.io/p/sandbox/react-query-with-class-component-jrdy9?file=%2Fsrc%2Fuser-list-direct-client.tsx%3A24%2C62-24%2C75>

**Bài 5: Router, xử lý form**

5.1. React Router cơ bản

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về React Router trong ReactJS. Việc sử dụng Reat Router cho phép chúng ta có thể tùy biến URL trong một dự án ReactJS.

**React Router là gì ?**

SPA (single page application) hiện nay được coi là một xu thế để xây dựng một trang web bởi nhiều tính năng ưu việt, có rất nhiều thư viện cho phép xây dựng một trang SPA phổ biến nhất đó là ReactJS. Khi một trang web được xây dựng theo hướng SPA thì tất cả các UI của trang web sẽ được render ra một trang duy nhất, tùy vào từng trường hợp mà component sẽ được render.

Ngoài ra, chúng ta có thể sử dụng URL làm điều kiện xem xét rằng liệu component nào sẽ được render. Trong ReactJS, React Router là thư viện được xây dựng để thực hiện điều này.

React Router là một thư viện cho việc điều hướng URL tiêu chuẩn trong React, Nó cho phép chúng ta có thể đồng bộ UI với URL. Được thiết kế với API đơn giản, từ đó cho phép giải quyết các vấn đề về URL một cách nhanh chóng.

**Sử dụng React Router trong ReactJS**

Để sử dụng React Router chúng ta cần phải cài đặt thư viện này vào trong dự án React bằng cách sử dụng NPM:

*npm install react-router-dom*

Sau khi cài đặt thành công, trong trường hợp cần dùng đến React Router bạn chỉ cần import nó component đó.

*import { BrowserRouter, Route, Switch } from 'react-router-dom';*

Để hiểu rõ hơn về React Router chúng ta sẽ đi xây dựng một ví dụ cụ thể về nó.

**Xây dựng ví dụ**

Trong ví dụ này, chúng ta sẽ đi xây dựng một trang lading page với nhiều trang khác nhau. Cấu trúc thư mục của src sẽ như sau:

src/

---components/

--------About.js

--------Home.js

---------Shop.js

---index.js

---App.js

...more...

Trước tiên, chúng ta cần phải thiết lập app sử dụng React Router. Mọi thứ sẽ được render cần phải được bọc bên trong BrowserRouter, chúng ta sẽ lựa chọn component App bởi nó chính là component xử lí logic mặc định trong ReactJS. Trong file index.js của dự án chúng ta sẽ chỉnh sửa lại như sau:

// index.js

...

import { BrowserRouter} from 'react-router-dom';

...

ReactDOM.render(

<BrowserRouter>

<App/>

</BrowserRouter>,

document.getElementById('root')

);

Tiếp theo đó ở file src/App.js chúng ta cần phải sử dụng Switch để bọc các Router lại. Đây là điều bắt buộc, tất các các Route cần phải được bọc bởi Switch.

// Ở đây chúng ta import 4 component được xây dựng trong thư mục src/components

// đó là Home, About, Shop, Error

import React from 'react'

import { Route, Switch } from 'react-router-dom';

​import Home from './components/Home';

import About from './components/About';

import Shop from './components/Shop'

import Error from './components/Error'

export defaults function App() {

return (

<>

<Switch>

<Route path="/" component={Home} exact />

<Route path="/about" component={About} />

<Route path="/shop" component={Shop} />

<Route component={Error} />

</Switch>

</>

)

}

Route có nhiệm vụ render component theo path được chỉ định. Trong trường hợp ở trên Route có path là / có thêm một props nữa là exact bởi hầu hết các path đều thông qua /. Khi một route không có thuộc tính path thì render component khi URL không tồn tại.

Bên trên trong file App.js chúng ta đã import 3 component đó là Home, About, Shop , Error trong thư mục src/components, bây giờ chúng ta cần phải xây dựng 3 componet đó.

//file: components/Home.js

import React from "react";

import { Link } from 'react-router-dom';

export default function Home() {

return (

<div>

<h1> Home Page</h1>

<Link to="/about">About / </Link>

<Link to="/shop">Shop / </Link>

<Link to="/404">404 / </Link>

</div>

);

};

//file: components/About.js

import React from "react";

import { Link } from 'react-router-dom';

export default function About() {

return (

<div>

<h1> About Page</h1>

<Link to="/">Home Page </Link>

</div>

);

};

//file: components/Shop.js

import React from "react";

import { Link } from 'react-router-dom';

export default function Shop() {

return (

<div>

<h1> Shop Page</h1>

<Link to="/">Home Page </Link>

</div>

);

};

//file: components/Error.js

import React from "react";

import { Link } from 'react-router-dom';

export default function Error () {

return (

<div>

<h1> 404 - Error Page</h1>

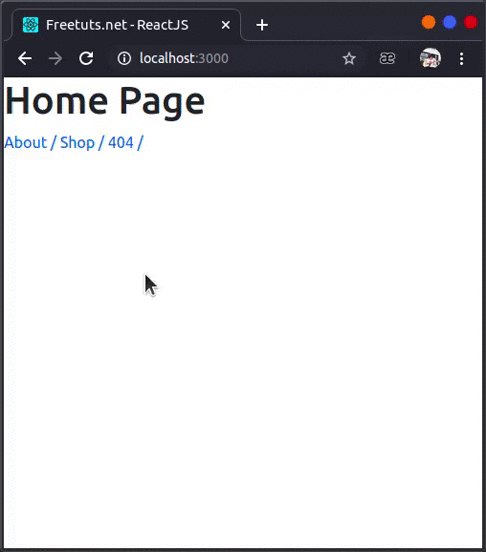
<Link to="/">Home Page </Link>

</div>

);

};

Component Link cho phép chúng ta chuyển qua lại giữa các component thông qua URL, nó tương tự như thẻ a trong html. Khởi chạy ứng dụng và chúng ta sẽ thấy sự thay đổi.



Sau khi thực hiện các bước cài đặt chúng ta có thể click vào các link để chuyển hướng qua lại trong SPA cụ thể là trong ReactJS rồi. Trên đây chúng ta đã cùng nhau đi tìm hiểu về React Router trong React Hooks. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

**Tìm hiểu chi tiết hơn về router trong reactjs**

<https://techmaster.vn/posts/37323/huong-dan-ve-react-router-moi-thu-ban-can-biet>

5.2. Xử lý Form trong ReactJS

Trong bài viết này chúng ta sẽ cùng nhau đi tìm hiểu về các thao tác xử lý form trong ReactJS.

Trong quá trình lập trình với React bạn cần phải làm việc với form nhất nhiều như đăng nhập, lấy thông tin người,vv.. Thao tác với form trong React rất đơn giản, hầu như chỉ là các kiến thức cơ bản ở phần trước như về component, state, props,...

Ngoài việc thao tác với form thông thường bạn còn có thể sử dụng các thư viện hỗ trợ việc validation form... Bởi vậy, trong bài này mình sẽ cùng nhau chỉ ra và đi xây dựng ví dụ về form trong ReactJS.

**Thao tác với Form trong ReactJS**

Dưới đây là các thao tác mà bạn có thể làm việc với form trong ReactJS, hầu như đây là những thao tác quen thuộc.

Lấy giá trị của input

Chúng ta có thể lấy giá trị của input bằng cách bắt sự kiện onChange của input.

Trước tiên, chúng ta sẽ tạo state dùng để chứa giá trị của input trong hàm constructor().

constructor(props) {

super(props);

this.state = {

email : ''

}

}

Tiếp theo, chúng ta sẽ bắt sự kiện onChange trong input :

<input type="email" onChange={(event) => this.changeInputValue(event)}/>

Khi sự kiện onChange được kích hoạt, chúng ta sẽ có một biến là event, trong đó sẽ chứa các thông tin của input như name, value,...Ở đây mình truyền biến event vào trong function changeInputValue().

Sau đó, chúng ta sẽ xây dựng hàm changeInputValue() dùng để thay đổi state (cái mà sẽ lưu giá trị của input) mà mình đã khởi tạo trước đó.

changeInputValue(event){

// Cập nhật state

this.setState({

[event.target.name]: event.target.value

})

}

Lúc này, bạn đã thực hiện xong các bước để lấy giá trị của input trong form. Tất cả các giá trị đó được lưu trong state.

Submit Form

Sau khi lấy giá trị của input, form cần được submit, bạn có thể thực hiện submit form bằng cách bắt sự kiện onSubmit trong form.

<form onSubmit={(event) => {

this.submitForm(event)

}}>

khi sự kiện onSubmit được thực thi thì biến event chứa thông tin của form sẽ tồn tại, chúng ta sẽ truyền nó vào trong hàm submitForm() để xử lý.

submitForm(event) {

//Chặn sự kiện mặc định của form

event.preventDefault()

//In ra giá trị của input trong form

console.log(this.state)

}

Validation Form

Bạn có thể thực hiện validation form trong ReactJS bằng cách xây dựng một hàm validation, giả sử mình có một hàm kiểm tra email:

validationForm() {

const re = /\S+@\S+\.\S+/;

//Kiểm tra email

if (re.test(this.state.email)) return false;

return true;

}

và thêm nó vào hàm submitForm() đã khởi tạo trước đó :

submitForm(event) {

//Chặn sự kiện mặc định của form

event.preventDefault()

//Validaton form

if(!this.validationForm()) {

alert('Email không đúng định dạng.')

}

}

**Xây dựng ví dụ form trong React JS**

Trong phần này mình sẽ đi xây dựng ví dụ về xử lý form trong ReactJS, ở đây mình sẽ đi xây dựng một trang đăng nhập đơn giản bao gồm chức năng validation.

Ở phần khởi tạo giao diện mình sử dụng Bootstrap 4 để xây dựng giao diện, bởi vậy bạn cần thêm thư viện này vào trong file public/index.html:

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css">

Từ bây giờ chúng ta sẽ làm việc với các file trong thư mục src

Trước tiên, chúng ta sẽ đi xây dựng giao diện cho form, trong file App.js chúng ta sẽ đi xây dựng giao diện và khơi tạo state chứa giá trị của input.

import React from "react";

class App extends React.Component {

constructor(props) {

super(props);

//Khởi tạo state chứa giá trị của input

this.state = {

email: "",

password: ""

};

}

render() {

return (

<div className="container" style={{ paddingTop: "5%" }}>

<form

onSubmit={e => {

this.submitForm(e);

}}

>

<div className="form-group">

<label htmlFor="text">Email:</label>

<input

type="text"

className="form-control"

name="email"

placeholder="Enter email"

onChange={e => this.changeInputValue(e)}

/>

</div>

<div className="form-group">

<label htmlFor="pwd">Password:</label>

<input

type="password"

className="form-control"

name="password"

placeholder="Enter password"

onChange={e => this.changeInputValue(e)}

/>

</div>

<button type="submit" className="btn btn-primary">

Submit

</button>

</form>

</div>

);

}

}

export default App;

Tiếp theo, xây dựng hàm changeInputValue() có nhiệm vụ lấy giá của input sau đó cập nhật vào state.

changeInputValue(e) {

this.setState({

[e.target.name]: e.target.value

});

}

Xây dựng hàm validationForm() để kiểm tra các giá trị khi submit form :

validationForm() {

let returnData = {

error : false,

msg: ''

}

const {email, password} = this.state

//Kiểm tra email

const re = /\S+@\S+\.\S+/;

if (!re.test(email)) {

returnData = {

error: true,

msg: 'Không đúng định dạng email'

}

}

//Kiểm tra password

if(password.length < 8) {

returnData = {

error: true,

msg: 'Mật khẩu phải lớn hơn 8 ký tự'

}

}

return returnData;

}

Cuối cùng, xây dựng hàm submitForm() cho sự kiện submit form:

submitForm(e) {

//Chặn các event mặc định của form

e.preventDefault();

//Gọi hàm validationForm() dùng để kiểm tra form

const validation = this.validationForm()

//Kiểm tra lỗi của input trong form và hiển thị

if (validation.error) {

alert(validation.msg)

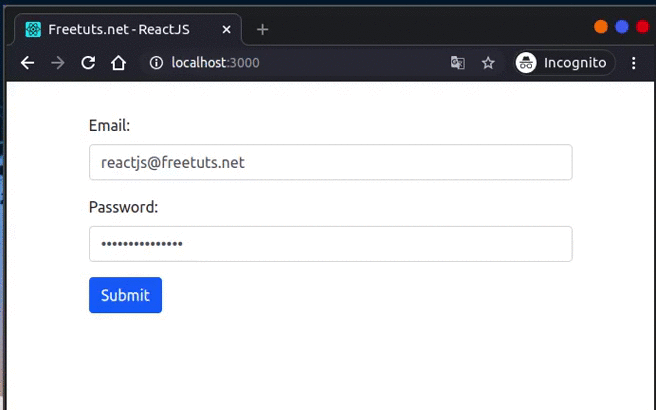
}else{

alert('Submit form success')

}

}

Khởi chạy dự án bằng cách mở terminal lên và gõ dòng lệnh:



Trên đây chúng ta đã cùng nhau đi tìm hiểu về xử lý Form trong ReactJS. Đây là kiến thức rất cơ bản về nó nhưng cũng hết sức quan trọng trong quá trình làm việc với ReactJS sau này.

5.4. Ví dụ minh họa

**Bài 6. Hooks, Recoil**

6.1.1. React Hooks là gì?

React Hooks là một tính năng trong thư viện React, được giới thiệu từ phiên bản React 16.8, giúp bạn sử dụng các tính năng của React trong các hàm component chức năng (functional components) thay vì chỉ trong các class component. Hooks giúp bạn quản lý trạng thái (state), hiệu suất và các vòng đời của component một cách dễ dàng và gọn gàng hơn.

6.1.2. Tại sao chúng ta cần sử dụng Hooks?

Quản lý state và life cycle của component dễ dàng hơn: Trước khi có React Hooks, state và life cycle của React components thường chỉ được quản lý trong Class Components. Functional Components không có khả năng quản lý trạng thái hoặc thực hiện các tác vụ sau khi render. React Hooks giúp bạn sử dụng trạng thái và vòng đời trong Functional Components một cách dễ dàng và hiệu quả. Dưới đây là cách quản lý state và life cycle của React components trong Class Component:

import React, { Component } from 'react';

class ExampleComponent extends Component {

constructor(props) {

super(props);

this.state = { count: 0 };

}

componentDidMount() {

// Thực hiện sau khi component đã được render lần đầu

console.log('Component did mount');

}

componentDidUpdate(prevProps, prevState) {

// Thực hiện sau mỗi lần component được cập nhật

console.log('Component did update');

}

componentWillUnmount() {

// Thực hiện trước khi component bị unmount

console.log('Component will unmount');

}

incrementCount = () => {

this.setState({ count: this.state.count + 1 });

}

render() {

return (

<div>

<p>Count: {this.state.count}</p>

<button onClick={this.incrementCount}>Increment</button>

</div>

);

}

}

Còn đây là cách quản lý trong Functional Component khi sử dụng Hooks:

import React, { useState, useEffect } from 'react';

function ExampleComponent() {

// Khởi tạo state bằng useState

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

// Sử dụng useEffect thay thế cho lifecycle methods

useEffect(() => {

// Thực hiện sau khi component đã được render lần đầu (tương đương với componentDidMount)

console.log('Component did mount');

// Cleanup function tương đương với componentWillUnmount

return () => {

console.log('Component will unmount');

};

}, []); // [] đại diện cho dependencies, rỗng nghĩa là sẽ chỉ thực hiện một lần sau lần render đầu tiên

// Sử dụng cú pháp arrow function để cập nhật state

const incrementCount = () => {

setCount(count + 1);

}

// Render component

return (

<div>

<p>Count: {count}</p>

<button onClick={incrementCount}>Increment</button>

</div>

);

}

export default ExampleComponent;

Trông gọn gàng và dễ hiểu hơn nhiều đúng không?

Giảm sự phức tạp của class components: Class components có thể trở nên phức tạp khi bạn cần quản lý nhiều vòng đời và trạng thái khác nhau. React Hooks giảm đi sự phức tạp này bằng cách cho phép bạn sử dụng các hook riêng lẻ để quản lý từng khía cạnh của component, làm cho mã nguồn trở nên gọn gàng hơn.

Cải thiện hiệu suất và tối ưu hóa việc render: React Hooks giúp bạn tối ưu hóa hiệu suất của ứng dụng bằng cách cho phép bạn tối ưu hóa việc render components thông qua useMemo, useCallback, và các hooks khác. Điều này giúp tránh việc render không cần thiết và cải thiện hiệu suất tổng thể.

Tái sử dụng logic và trạng thái: React Hooks cho phép bạn tái sử dụng logic và trạng thái dễ dàng hơn. Bạn có thể viết các custom hooks để chia sẻ logic giữa các components khác nhau, giúp giảm lặp lại mã nguồn và làm cho mã nguồn trở nên linh hoạt hơn.

6.1.3. Một số Hook thường dùng trong React

Sau khi đã hiểu lý do tại sao nên dùng Hooks, chúng ta cùng tìm hiểu xem những Hooks nào thường được sử dụng trong thực tế nhé. React đã cung cấp sẵn cho chúng ta rất nhiều Hooks như useState(), useEffect(), useCallback(),... Ngoài những hooks trên các bạn hoàn toàn có thể tạo ra hooks cho riêng mình để sử dụng vào những trường hợp khác nhau.

6.1.3.1 useState()

a) useState() là gì?

useState() là một hook trong React được sử dụng để khởi tạo và quản lý trạng thái (state) trong Functional Components. Hook này cho phép bạn thêm trạng thái vào Functional Components mà trước đây chỉ có thể được quản lý trong Class Components. useState() trả về một mảng với hai phần tử:

State variable (biến trạng thái): Đây là biến mà bạn sử dụng để lưu trữ giá trị trạng thái của component.

State updater function (hàm cập nhật trạng thái): Đây là hàm bạn sử dụng để cập nhật giá trị trạng thái. Khi bạn gọi hàm này với một giá trị mới, nó sẽ cập nhật giá trị trạng thái và render lại component để cập nhật giao diện người dùng.

Dưới đây là một ví dụ đơn giản về cách sử dụng useState():

import React, { useState } from 'react';

function ExampleComponent() {

// Khởi tạo trạng thái với giá trị ban đầu là 0

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

// Hàm tăng giá trị trạng thái

const incrementCount = () => {

setCount(count + 1); // Gọi hàm setCount để cập nhật trạng thái

}

return (

<div>

<p>Count: {count}</p>

<button onClick={incrementCount}>Increment</button>

</div>

);

}

Trong ví dụ này, chúng ta sử dụng useState(0) để khởi tạo trạng thái count với giá trị ban đầu là 0. Khi người dùng nhấn nút "Increment," hàm incrementCount được gọi, và chúng ta sử dụng setCount để cập nhật giá trị trạng thái. React sẽ tự động render lại component để cập nhật giao diện người dùng.

b) Một số lưu ý khi sử dụng useState()

Bắt buộc phải khởi tạo giá trị ban đầu cho useState(), điều này giúp đảm bảo rằng trạng thái không bao giờ là undefined. useState() có thể lưu trữ các giá trị dạng strings, numbers, booleans, arrays, objects. Ví dụ: const [count, setCount] = useState(0);.

Giá trị của trạng thái sẽ quay về giá trị khởi tạo sau mỗi lần refresh trang

Component sẽ render lại mỗi khi giá trị của trạng thái bị thay đổi

useState() không ghi đè các trạng thái mà thay thế trạng thái cũ bằng giá trị của trạng thái mới

Không bao giờ thay đổi trạng thái trực tiếp bằng cách gán giá trị mới, ví dụ: count = count + 1. Hãy sử dụng setCount để cập nhật lại trạng thái.

Hàm cập nhật trạng thái có thể nhận tham số là giá trị mới hoặc một hàm trả về giá trị mới dựa trên trạng thái trước đó. Ví dụ: setCount(count + 1) hoặc setCount(prevCount => prevCount + 1).

Chỉ sử dụng useState cho những trạng thái cần thiết và không tạo ra quá nhiều trạng thái không cần thiết trong component của bạn.

c) Khi nào thì nên sử dụng useState()

useState trong React nên được sử dụng khi bạn cần quản lý và theo dõi trạng thái của functional components. Dưới đây là một số tình huống nên sử dụng useState:

Quản lý trạng thái của giao diện người dùng: Khi bạn muốn lưu trữ và cập nhật thông tin trạng thái liên quan đến giao diện người dùng, như số lượng sản phẩm trong giỏ hàng, giá trị của các trường input, hoặc trạng thái hiển thị/hide của các phần tử.

Lưu trữ dữ liệu tạm thời: Khi bạn cần lưu trữ các dữ liệu tạm thời mà không cần sử dụng Redux hoặc Context API, useState là một cách tiện lợi để quản lý trạng thái tạm thời này.

Xử lý trạng thái của form: useState thường được sử dụng để lưu trạng thái của form như giá trị của các trường input, chọn các mục trong danh sách, và kiểm tra tính hợp lệ của form trước khi gửi dữ liệu.

Các trạng thái cần được cập nhật sau khi component đã render: useState cho phép bạn cập nhật trạng thái sau khi component đã render thông qua sự kích hoạt của các sự kiện hoặc lệnh gọi API. Điều này giúp bạn cập nhật giao diện người dùng khi trạng thái thay đổi.

Quản lý các hiệu ứng tương tác đơn giản: Nếu bạn cần quản lý các hiệu ứng tương tác đơn giản như sự thay đổi của nút bấm hoặc sự xuất hiện/ẩn của các phần tử, useState có thể được sử dụng để theo dõi và cập nhật trạng thái của các hiệu ứng này.

Tóm lại, useState nên được sử dụng khi bạn cần quản lý trạng thái trong functional components của React. Nó giúp bạn làm cho components trở nên tương tác và có khả năng cập nhật dễ dàng, đồng thời giúp tạo ra các ứng dụng React hiệu quả với mã nguồn dễ đọc và dễ bảo trì.

6.1.3.2. useEffect()

a) useEffect() là gì?

useEffect là một trong những hooks quan trọng trong React, được sử dụng để thực hiện các tác vụ phụ (side effects) trong Functional Components. Các tác vụ phụ bao gồm lệnh gọi API, thay đổi trạng thái, đăng ký và hủy đăng ký sự kiện, và các tác vụ không thuộc về việc render giao diện người dùng. useEffect giúp bạn thực hiện các tác vụ này tại các thời điểm cụ thể trong vòng đời của component.

Dưới đây là một ví dụ đơn giản về cách sử dụng useEffect() để thực hiện việc gọi API bằng thư viện axios:

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

import axios from "axios";

function ExampleComponent() {

const [data, setData] = useState({});

const getDataFromAPI = async () => {

const response = await axios.get(

"https://jsonplaceholder.typicode.com/todos/1"

);

setData(response.data);

};

useEffect(() => {

getDataFromAPI();

}, []);

return <div>{data.title}</div>;

}

export default ExampleComponent;

Luồng xử lý của ví dụ trên diễn ra như sau:

Khai báo state data với giá trị khởi tạo là 1 object rỗng {}

Tạo hàm getDataFromAPI() sử dụng thư viện axios để call api

Gọi hàm getDataFromAPI() bên trong useEffect(). Hàm này sẽ được gọi khi component được render lần đầu tiên

Sau khi lấy được dữ liệu thì gán lại state data với dữ liệu mới bằng setData

Dữ liệu mới được đưa lên UI để hiển thị cho người dùng

b) Dependencies trong useEffect()

Dependencies trong useEffect được sử dụng để kiểm soát cách hoạt động của useEffect() khi useEffect được gọi lại. Có 3 trường hợp sử dụng useEffect():

Không cung cấp dependencies:

import React, { useState, useEffect } from 'react';

function MyComponent() {

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

// useEffect không có dependencies

useEffect(() => {

console.log('useEffect ran.');

});

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increase Count</button>

</div>

);

}

export default MyComponent;

Trong ví dụ này, useEffect không có dependencies, vì vậy nó sẽ chạy sau mỗi lần component được render. Khi bạn nhấn vào nút "Increase Count" để tăng giá trị của count, useEffect sẽ luôn được gọi lại và in ra "useEffect ran." trên console. Điều này có nghĩa rằng nó chạy sau mỗi lần tương tác với component, dù giá trị count có thay đổi hay không.

Trong nhiều trường hợp, việc không cung cấp dependencies có thể dẫn đến việc gọi useEffect quá nhiều lần và làm cho ứng dụng của bạn hoạt động không hiệu quả. Điều này có thể gây ra tải trang chậm hoặc tạo ra các vấn đề hiệu năng khác.

Dependencies là mảng rỗng

import React, { useState, useEffect } from 'react';

function MyComponent() {

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

// useEffect với mảng dependencies rỗng

useEffect(() => {

console.log('useEffect ran.');

}, []);

return (

<div>

<p>Count: {count}</p>

<button onClick={() => setCount(count + 1)}>Increase Count</button>

</div>

);

}

export default MyComponent;

Trong ví dụ này, useEffect có một mảng dependencies rỗng, []. Điều này đồng nghĩa với việc nó chỉ chạy một lần sau khi component được render lần đầu tiên. Khi bạn nhấn vào nút "Increase Count" để tăng giá trị của count, useEffect sẽ không bao giờ được gọi lại. Điều này thích hợp cho các tác vụ mà bạn chỉ muốn thực hiện một lần và không muốn chúng phụ thuộc vào bất kỳ giá trị nào. Cách hoạt động tương tự như componentDidMount của Class Component.

Dependencies là props hoặc state

Khi bạn truyền dependencies là props hoặc state, useEffect sẽ được gọi lại mỗi khi giá trị của dependencies thay đổi. Điều này thường được sử dụng để thực hiện các tác vụ phụ thuộc vào sự thay đổi của props hoặc state. Cơ chế này tương tự như bạn sử dụng Life Cycle componentDidUpdate và shouldComponentUpdate của Class Component.

import React, {useEffect, useState} from 'react'

function ExampleComponent() {

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

// thay đổi giá trị của count

const incrementCount = () => {

setCount(count + 1);

}

// không thay đổi giá trị của count

const doNothing = () => {

setCount(count);

}

// useEffect callback được gọi khi state thay đổi so với giá trị trước đó

useEffect(() => {

console.log("useEffect ran.");

}, [count])

return (

<section>

<h1>{count}</h1>

<button onClick={incrementCount}>Tăng thêm</button>

<button onClick={doNothing}>Không có gì xảy ra</button>

</section>

);

}

Trong ví dụ này, useEffect có count là một dependency. Khi gọi đến hàm incrementCount(), giá trị của count thay đổi tăng lên 1, lúc này useEffect() được gọi lại và in dòng useEffect ran. ra console. Nhưng khi gọi đến hàm doNothing(), giá trị của state count không thay đổi nên useEffect() không được gọi lại trong trường hợp này.

c) Clean up useEffect()

Khi bạn sử dụng useEffect trong React để thực hiện các tác vụ side effect, bạn có thể cần thực hiện một số tác vụ dọn dẹp sau khi component bị unmount hoặc khi dependencies thay đổi. Điều này giúp tránh rò rỉ bộ nhớ (memory leak) hoặc xảy ra lỗi khi chuyển sang component khác.

Ví dụ, khi bạn thêm một sự kiện như sự kiện cuộn trang (scroll) bằng addEventListener trong React, bạn cũng cần dọn dẹp (remove) sự kiện khi component bị unmount để tránh rò rỉ bộ nhớ.

import React, { useEffect } from 'react';

function ScrollListenerComponent() {

// Hàm xử lý sự kiện cuộn

const handleScroll = () => {

// Xử lý sự kiện cuộn ở đây

console.log('Scrolled');

};

useEffect(() => {

// Thêm sự kiện cuộn khi component được mount

window.addEventListener('scroll', handleScroll);

// Hàm dọn dẹp, được gọi khi component bị unmount

return () => {

window.removeEventListener('scroll', handleScroll);

};

}, []);

return (

<div>

<p>Scroll down to trigger the event.</p>

</div>

);

}

export default ScrollListenerComponent;

Luồng hoạt động của ví dụ trên như sau:

tạo một hàm handleScroll để xử lý sự kiện cuộn.

Trong useEffect, chúng ta thêm một sự kiện cuộn bằng window.addEventListener khi component được mount.

Chúng ta trả về một hàm clean up từ useEffect, và trong hàm này, chúng ta gỡ bỏ sự kiện cuộn bằng window.removeEventListener khi component bị unmount. Điều này đảm bảo rằng sự kiện cuộn sẽ không được theo dõi sau khi component bị unmount.

d) Khi nào nên sử dụng useEffect()

Thực hiện các tác vụ không liên quan đến việc rendering: useEffect() thường được sử dụng để thực hiện các tác vụ như gọi API, thao tác với DOM, thiết lập các lệnh lắng nghe sự kiện (event listeners), và các tác vụ khác không liên quan trực tiếp đến việc rendering của component.

Quản lý lý thuyết hợp đồng (lifecycle): useEffect() cho phép bạn thực hiện các tác vụ tại các điểm quan trọng trong vòng đời của một component React, như sau khi component được tạo (componentDidMount) hoặc sau khi nó được cập nhật (componentDidUpdate). Điều này giúp bạn quản lý tất cả các tác vụ "side effects" một cách dễ dàng.

Theo dõi và xử lý sự thay đổi của props hoặc state: Bạn có thể sử dụng useEffect() để theo dõi sự thay đổi của props hoặc state và thực hiện các tác vụ phản ứng dựa trên những thay đổi đó.

Thực hiện các tác vụ liên quan đến dữ liệu cần lấy từ API hoặc dự báo: Nếu bạn cần lấy dữ liệu từ một API hoặc thực hiện các tính toán phức tạp, bạn có thể sử dụng useEffect() để thực hiện các tác vụ này.

Lắng nghe và xử lý sự kiện gốc (native events): useEffect() cũng thích hợp để lắng nghe các sự kiện gốc của trình duyệt như sự kiện resize, scroll, hoặc bất kỳ sự kiện nào mà bạn muốn theo dõi và xử lý.

Làm sạch tác vụ khi component unmount (componentWillUnmount): Bạn có thể sử dụng useEffect() với một hàm trả về để làm sạch các tác vụ khi component bị unmount.

6.1.3.3 useContext()

a) useContext() là gì?

useContext là một hook trong React được sử dụng để truy cập các giá trị của một Context API. Context API trong React là một cách để truyền dữ liệu từ một thành phần cha đến các thành phần con mà không cần truyền props qua nhiều lớp con trung gian. useContext giúp bạn lấy giá trị từ Context API một cách dễ dàng.

b) Cách sử dụng useContext()

Định nghĩa một Context: Trước tiên, bạn cần định nghĩa một Context bằng cách sử dụng createContext:

import React, { createContext } from 'react';

const MyContext = createContext();

Tạo một Provider: Bạn cần tạo một thành phần Provider để đặt giá trị cho Context. Thông thường, bạn sẽ làm điều này tại thành phần cha cao nhất của ứng dụng React:

function MyContextProvider(props) {

const myValue = "Giá trị mẫu từ Context"; // Đặt giá trị bạn muốn chia sẻ

return (

<MyContext.Provider value={myValue}>

{props.children}

</MyContext.Provider>

);

}

Sử dụng useContext trong thành phần con: Bây giờ bạn có thể sử dụng useContext trong các thành phần con để truy cập giá trị từ Context:

import React, { useContext } from 'react';

function MyComponent() {

const valueFromContext = useContext(MyContext);

return (

<div>

Giá trị từ Context: {valueFromContext}

</div>

);

}

Kết nối Provider và các thành phần con: Để cung cấp giá trị từ Context cho các thành phần con, hãy đảm bảo rằng bạn đã bao quanh các thành phần con bằng thành phần Provider:

function App() {

return (

<MyContextProvider>

<MyComponent />

</MyContextProvider>

);

}

c) Khi nào thì nên sử dụng useContext()

Chia sẻ dữ liệu global: Khi bạn cần truyền dữ liệu hoặc trạng thái từ thành phần cha đến các thành phần con ở nhiều mức độ sâu trong ứng dụng mà không muốn truyền props qua từng lớp con trung gian, useContext là lựa chọn tốt. Điều này giúp làm cho mã nguồn của bạn dễ đọc hơn và giảm độ phức tạp của việc truyền props.

Xác định trạng thái ứng dụng toàn cục: Nếu bạn cần theo dõi trạng thái toàn cục của ứng dụng, ví dụ: trạng thái đăng nhập, ngôn ngữ hiện tại, hoặc chế độ tối/sáng, useContext có thể giúp bạn quản lý trạng thái này một cách hiệu quả và chia sẻ nó với các thành phần con.

Giữa các thành phần không liên quan trực tiếp: Khi bạn muốn truyền dữ liệu giữa các thành phần không có mối quan hệ cha con trực tiếp, useContext giúp bạn làm điều này mà không cần truyền props qua từng thành phần trung gian.

6.1.3.4 useReducer()

a) useReducer() là gì?

useReducer là một hooks được sử dụng để quản lý trạng thái (state) và hành động (actions) của ứng dụng. useReducer gần giống với useState, tuy nhiên nó thường được sử dụng khi bạn cần quản lý một trạng thái phức tạp hơn và khi trạng thái của bạn có quá nhiều hành động gây ra sự thay đổi.

b) Cách sử dụng useReducer()

Định nghĩa reducer function: Trước hết, bạn cần xác định một reducer function. Reducer là một hàm nhận vào hai tham số: trạng thái hiện tại và hành động. Nó sẽ trả về một trạng thái mới dựa trên hành động.

const reducer = (state, action) => {

switch (action.type) {

case 'INCREMENT':

return { count: state.count + 1 };

case 'DECREMENT':

return { count: state.count - 1 };

default:

return state;

}

};

Sử dụng useReducer trong component: Trong component của bạn, sử dụng useReducer và truyền vào reducer function và trạng thái ban đầu.

import React, { useReducer } from 'react';

function Counter() {

const initialState = { count: 0 };

const [state, dispatch] = useReducer(reducer, initialState);

// Bây giờ 'state' chứa trạng thái hiện tại và 'dispatch' là một hàm để gửi hành động.

// Ví dụ về việc tăng/giảm giá trị count:

const increment = () => {

dispatch({ type: 'INCREMENT' });

};

const decrement = () => {

dispatch({ type: 'DECREMENT' });

};

return (

<div>

<p>Count: {state.count}</p>

<button onClick={increment}>Increment</button>

<button onClick={decrement}>Decrement</button>

</div>

);

}

Trong ví dụ trên, chúng ta sử dụng useReducer để quản lý trạng thái count. Hành động được xác định bởi dispatch sẽ gửi đến reducer function, và trạng thái mới sẽ được tính toán dựa trên hành động mà chúng ta gửi.

Với useReducer, bạn có thể quản lý trạng thái phức tạp hơn và loại bỏ việc trạng thái bị trôi (state drift) bởi việc xác định rõ các hành động có thể xảy ra.

c) Khi nào nên sử dụng useReducer()

Quản lý trạng thái phức tạp: Khi bạn cần quản lý một trạng thái phức tạp với nhiều thuộc tính hoặc mô hình dữ liệu, useReducer có thể giúp bạn duyệt qua các action và cập nhật trạng thái một cách dễ dàng hơn so với useState.

Tránh sự cố lỗi khi cập nhật trạng thái: Khi bạn đang làm việc với trạng thái phức tạp và muốn đảm bảo tính nhất quán trong việc cập nhật trạng thái, useReducer giúp tránh nhầm lẫn và sai sót trong quá trình cập nhật trạng thái.

Tái sử dụng logic cập nhật trạng thái: Khi bạn muốn tái sử dụng logic cập nhật trạng thái (như xác định xem một item có được thêm vào danh sách hay không) cho nhiều phần của ứng dụng, useReducer giúp bạn trừu tượng hóa logic này thành một reducer và sử dụng lại ở nhiều nơi.

6.1.3.5 useForm

Hầu hết một Front-end Developer ai cũng trải qua việc xử lý form, nếu làm theo cách thủ công sẽ tốn rất nhiều thời gian để xử lý. Vì thế mình sẽ giới thiệu tới các bạn một thư viện giúp chúng ta tiết kiệm thời gian cho việc quản lý form, đó là React Hook Form.

Là một thư viện quản lý form tốt nhất hiện nay mà bạn nên dùng. Ở bài viết này mình sẽ demo code với ReactJS.

Tải thư viện

NPM: npm install react-hook-form

YARN: yarn add react-hook-form

Ví dụ cơ bản

Ví dụ cơ bản về React Hook Form, các bạn hãy đọc comment code để có thể hiểu về nội dung câu lệnh dùng để làm gì.

import React from "react";

import { useForm } from "react-hook-form";

export default function App() {

const { register, handleSubmit, formState: { errors } } = useForm({

criteriaMode: "all"

});

return (

// Hàm handleSubmit sẽ validate trước khi gọi hàm onSubmit

<form onSubmit={handleSubmit(onSubmit)}>

// đăng kí input cho Hook vói tên example

<input defaultValue="test" {...register("example")} />

// đăng kí thẻ input với React-Hook-Form với tên "exampleRequired"

// validate là required

<input {...register("exampleRequired", { required: true })} />

// xử lý lỗi bằng đối tượng errors được trả về từ useForm

{errors.exampleRequired && <span>This field is required</span>}

<input type="submit" />

</form>

);

}

Khái niệm chính

**useForm**

Để làm việc với React-Hook-Form đầu tiên chúng ta cần khởi tạo useForm.

**Tham số truyền vào**

Ví dụ:

const { register, handleSubmit, formState: { errors } } = useForm({

mode: 'onSubmit',

reValidateMode: 'onChange',

defaultValues: {},

resolver: undefined,

criteriaMode: "firstError",

})

Một vài tham số thường dùng để truyền vào useForm:

mode : có các giá trị onChange | onBlur | onSubmit | onTouched | all . Dùng để cấu hình một chiến lược validate trước khi submit form.

onSubmit: sẽ thực hiện validate khi submit form, và những element không hợp lệ sẽ được lắng nghe sự thay đổi và sau đó tiếp tục validate những element đó bằng mode onChange.

onChange: sẽ thực hiện validate khi mỗi khi onChange element, và nó dẫn đến re-render nhiều lần ( cần cân nhắc khi sử dụng ).

onBlur: sẽ thực hiện validate mỗi khi element có sự kiện blur.

onTouch: sẽ thực hiện validate cho lần blur đầu tiên, sau đó sẽ thực hiện validate cho mỗi lần change event.

all: sẽ thực hiện validate khi blur và change event.

defaultValues: thiết lập giá trị mặc định ( lần đầu ) cho form.

criteriaMode: có các giá trị firstError | all. Đối với firstError sẽ chỉ nhận được một lỗi đầu tiên, all sẽ nhận được tất cả lỗi.

resolver: hổ trợ validate đối với thirt-party ( yup ).

**Giá trị trả về**

**Register**

Register là một trong những nội dung chính của React Hook Form, nó dùng để đăng kí component của bạn với hook.

function App() {

const { register, handleSubmit } = useForm();

const onSubmit = data => {

console.log(data); // { name: ... }

}

return (

<>

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register("name")} />

<input type="submit" />

</form>

</>

);

}

Ví dụ:

<input

{...register("test", {

// validate

required: true

// bắt sự kiện onChange

onChange: (e) => console.log(e)

// bắt sự kiện onBlur

onBlur: (e) => console.log(e)

})}

/>

handleSubmit

handleSubmit sẽ thực hiện validate, nếu validate thành công sẽ gọi hàm onSubmit. ví dụ:

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register("name")} />

<input type="submit" />

</form>

);

formState

import React from "react";

import { useForm } from "react-hook-form";

export default function App() {

const {

register,

handleSubmit,

formState: { errors, isDirty, isSubmitting, isSubmitted, submitCount, isValid, isValidating },

} = useForm();

const onSubmit = (data: FormInputs) => console.log(data);

return (

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register("test")} />

<input type="submit" />

</form>

);

}

Các giá trị của formState:

errors trả về một object chứa lỗi, và key của object tương ứng với name mà chúng ta truyền vào hàm Register. Nếu trường hợp không có lỗi nào thì sẽ trả về object rỗng.

isDirty so sánh data input hiện tại với defaultValue và trả về true nếu value hiện tại khác với defaultValue ( dùng để show pop-up khi user chỉnh sửa nhưng chưa submit ).

isSubmitting trả về true khi form đang submit.

isSubmitted trả về true khi form submit xong.

submitCount trả về thời gian ( dạng number ) submit form .

isValid trả về true nếu không có bất kì lỗi nào, false khi có lỗi.

isValidating trả về true nếu đang validate.

watch

watch sẽ lắng nghe sự thay đổi giá trị của ô input ( giống như sự kiện onChange ).

import React from "react";

import { useForm } from "react-hook-form";

function App() {

const { register, watch, formState: { errors }, handleSubmit } = useForm();

const watchShowAge = watch("showAge", false); // đăng kí watch cho field showAge, truyền giá trị mặc định bằng đối số thứ 2

const watchAllFields = watch(); // nếu không truyền tham số, watch sẽ lắng nghe và trả về tất cả fields

const onSubmit = data => console.log(data);

return (

<>

<form onSubmit={handleSubmit(onSubmit)}>

<input type="checkbox" {...register("showAge")} />

<input type="submit" />

</form>

</>

);

}

reset

Reset dùng để đặt lại giá trị mặc định cho Form ( thường dùng để sau khi call API chúng ta sẽ đặt lại giá trị mặc định để lắng nghe sự thay đổi của isDirty ).

import { useForm } from "react-hook-form";

export default function App() {

const { register, handleSubmit, reset } = useForm();

const onSubmit = (data) => {

// call API

const data = fetchListUser();

reset(data); // đặt lại giá trị mặc định cho Form

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register("firstName", { required: true })} />

<input {...register("lastName")} />

<input type="submit" />

</form>

);

}

setError

setError dùng để set lỗi cho một element.

import { useForm } from "react-hook-form";

const App = () => {

const { register, handleSubmit, setError, formState: { errors } } = useForm();

const onSubmit = data => {

console.log(data)

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register('firstName'}/>

{errors.firstName && <p>{errors.firstName.message}</p>}

<button onClick={() => setError("firstName",'Message Error')}} >

Set Error

</button>

</form>

);

};

clearError

Đã có setError thì phải có clearError, nó dùng để xóa một lỗi nào đó khỏi errors dựa vào name.

import { useForm } from "react-hook-form";

const App = () => {

const { register, handleSubmit, clearError, formState: { errors } } = useForm();

const onSubmit = data => {

console.log(data)

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register('firstName'}/>

{errors.firstName && <p>{errors.firstName.message}</p>}

<button onClick={() => clearError("firstName")}} >

Clear Error

</button>

</form>

);

};

setValue

Đây là function cho phép bạn set value cho một element.

import { useForm } from "react-hook-form";

const App = () => {

const { register, handleSubmit, setValue, formState: { errors } } = useForm();

const onSubmit = data => {

console.log(data)

};

return (

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register('firstName'}/>

{errors.firstName && <p>{errors.firstName.message}</p>}

<button onClick={() => setValue("test",'')}} > // set value của ô input thành '' ( rỗng )

Clear Value

</button>

</form>

);

};

setFocus

setFocus được dùng để set focus cho một element.

export default function App() {

const { register, handleSubmit, setFocus } = useForm();

const onSubmit = (data) => console.log(data);

useEffect(() => {

setFocus("firstName");

}, []);

return (

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register("firstName")} placeholder="First Name" />

<input type="submit" />

</form>

);

}

getValues

Thường được dùng để lấy value hiện tại trên element.

import React from "react";

import { useForm } from "react-hook-form";

export default function App() {

const { register, getValues } = useForm();

return (

<form>

<input {...register("test")} />

<input {...register("test1")} />

<button

type="button"

onClick={() => {

const values = getValues(); // { test: "test-input", test1: "test1-input" }

const singleValue = getValues("test"); // "test-input"

const multipleValues = getValues(["test", "test1"]); // ["test-input", "test1-input"]

}}

>

Get Values

</button>

</form>

);

}

trigger Đây là trường hợp bạn muốn re-validate một cách thủ công hãy dùng trigger(). ví dụ:

trigger("lastName"); // chỉ trigger lastName

trigger(["firstName", "lastName"]); // trigger lastName firstName

trigger(); // trigger tất cả

Validation và xử lý lỗi như thế nào ?

Validate, đây là phần khá quan trọng trong React Hook Form nó giúp các bạn thực hiện việc validate dễ dàng hơn. React Hook Form hổ trợ một số case validate như sau:

* required
* min
* max
* minLength
* maxLength
* pattern
* validate

Chúng ta cùng xem ví dụ dưới đây:

import { ErrorMessage } from "@hookform/error-message";

import { useForm } from "react-hook-form";

export default function App() {

const { register, handleSubmit, formState: { errors } } = useForm({

criteriaMode: "all"

});

const onSubmit = (data) => console.log(data);

return (

<form onSubmit={handleSubmit(onSubmit)}>

<h1>ErrorMessage</h1>

<input

{...register("multipleErrorInput", {

// xử lý lỗi khi bỏ trống

required: "This input is required.",

// xử lý lỗi khi nhập khác giá trị là số

pattern: {

value: /\d+/,

message: "This input is number only."

},

// xử lý lỗi khi nhập nhiều hơn 10 kí tự

minLength: {

value: 11,

message: "This input must exceed 10 characters"

}

})}

/>

// ErrorMessage được import từ thư viện

<ErrorMessage

// nhận vào props errors ( errors lấy từ formState ở trên ) nó sẽ trigger khi phát hiện lỗi khác.

errors={errors}

// name tương ứng với name input đã đăng kí với hook

name="multipleErrorInput"

// sẽ render ra giao diện khi có error, ở đây là thẻ p

render={({ messages }) => {

return messages.map((message,index) => (

<p key={index}>{message}</p>

))

: null;

}}

/>

<input type="submit" />

</form>

);

}

Bạn có thể tham khảo thêm tại đây

Validate với yup

React-Hook-Form hổ trợ validate với yup, các bạn cần phải hiểu rõ về yup để tạo được schema Ví dụ:

import { useForm } from "react-hook-form";

import { yupResolver } from '@hookform/resolvers/yup';

import \* as yup from "yup";

// tạo schema để validate

const schema = yup.object({

firstName: yup.string().required(),

age: yup.number().positive().integer().required(),

}).required();

export default function App() {

const { register, handleSubmit, formState:{ errors } } = useForm({

// resolver dùng để validate với yup

resolver: yupResolver(schema)

});

const onSubmit = data => console.log(data);

return (

<form onSubmit={handleSubmit(onSubmit)}>

<input {...register("firstName")} />

<p>{errors.firstName?.message}</p>

<input {...register("age")} />

<p>{errors.age?.message}</p>

<input type="submit" />

</form>

);

}

Controller

Khi dùng thư viện bên ngoài như MaterialUI, Andt, ReactSelect, ReactDapicker chúng ta sẽ sử dụng Controller để xử lý form.

control chứa các phương thức để đăng kí component với hook Ví dụ:

function App() {

const { handleSubmit, control } = useForm();

return (

<form onSubmit={handleSubmit(data => console.log(data))}>

<Controller

control={control}

name="ReactDatepicker"

render={({ field: { onChange, onBlur, value, ref } }) => (

<ReactDatePicker

onChange={onChange}

onBlur={onBlur}

selected={value}

/>

)}

/>

<input type="submit" />

</form>

);

}

Tham khảo: <https://viblo.asia/p/react-hook-form-xu-ly-form-de-dang-hon-bao-gio-het-RnB5pAdDKPG>

6.1.3.5 Custom Hooks

Ngoài việc sử dụng các Hooks có sẵn được cung cấp bởi React, bạn cũng có thể tự tạo ra Hooks của riêng mình để phục vụ yêu cầu nhất định nào đó mà các Hooks React cung cấp không đáp ứng được.

Viết Custom Hooks là một cách để bạn tái sử dụng logic và trạng thái logic liên quan trong nhiều component khác nhau. Chúng giúp bạn tránh việc lặp lại code và làm cho mã của bạn trở nên dễ quản lý hơn. Một Custom Hook là một hàm JavaScript bình thường, bắt đầu bằng từ khóa use và nó có thể gọi các hook có sẵn hoặc các Custom Hook khác. Điều quan trọng là Custom Hook không phải là một hook cơ bản mà là một phong cách tổ chức mã của bạn.

Dưới đây là ví dụ về cách tạo một Custom Hooks đơn giản:

import { useState, useEffect } from 'react';

function useCounter(initialValue, step) {

const [count, setCount] = useState(initialValue);

const increment = () => {

setCount(count + step);

};

useEffect(() => {

document.title = `Count: ${count}`;

}, [count]);

return { count, increment };

}

export default useCounter;

Còn đây là cách sử dụng nó:

import React from 'react';

import useCounter from './useCounter';

function Counter() {

const { count, increment } = useCounter(0, 1);

return (

<div>

<p>Count: {count}</p>

<button onClick={increment}>Increment</button>

</div>

);

}

export default Counter;

Ở đây, chúng ta đã tạo một Custom Hook có tên useCounter, và nó chứa logic cho việc đếm và tăng giá trị. Sau đó, chúng ta sử dụng Custom Hook này trong component Counter để hiển thị giá trị và tạo hành động tăng giá trị.

Custom Hooks có thể giúp bạn tạo ra các module chứa logic riêng biệt, dễ dàng kiểm thử, và có thể sử dụng lại trong nhiều component khác nhau. Điều này giúp tách biệt logic của bạn khỏi giao diện người dùng và làm cho mã của bạn trở nên dễ bảo trì hơn.

6.1.4. Rule of Hooks

Hooks cũng có những nguyên tắc nhất định bắt chúng ta phải tuân theo để đảm bảo component hoạt động đúng cách và ổn định, hạn chế xảy ra lỗi không mong muốn. Có 3 nguyên tắc trong việc sử dụng Hooks:

Chỉ được sử dụng Hook bên trong component: Hook không được gọi trong các hàm JavaScript thông thường, các hàm helper, hoặc bất kỳ phạm vi khác ngoài component. Chúng phải được gọi bên trong một component.

Hook chỉ được gọi ở cấp độ ngang hàng (top level): Hook phải được gọi ở cấp độ ngang hàng của hàm component. Điều này có nghĩa là bạn không nên gọi hook trong các hàm con, các câu lệnh điều kiện, hoặc vòng lặp.

Hook phải được gọi trong cùng một thứ tự mỗi lần render: Hook phải luôn được gọi trong cùng một thứ tự trong mỗi lần render. Điều này giúp React theo dõi việc cập nhật các hook và đảm bảo rằng chúng sẽ hoạt động đúng cách.

Ví dụ dưới đây mô tả về trường hợp không tuân thủ theo nguyên tắc của Hooks:

function MyComponent() {

if (someCondition) {

// Không tuân theo Rule of Hooks: Gọi hook trong hàm điều kiện

useState(0);

}

for (let i = 0; i < 5; i++) {

// Không tuân theo Rule of Hooks: Gọi hook trong vòng lặp

useEffect(() => {

// ...

});

}

// Tuân theo Rule of Hooks: Gọi hook ở cấp độ hàng ngang

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

useEffect(() => {

// ...

});

return (

// ...

);

}

Trong ví dụ trên, gọi hook trong hàm điều kiện hoặc vòng lặp là vi phạm "Rule of Hooks" và có thể gây ra lỗi. Hook phải được gọi ở cấp độ ngang hàng của component và luôn ở cùng một thứ tự.

Tham khảo: <https://200lab.io/blog/react-hooks/#:~:text=React%20Hooks%20l%C3%A0%20m%E1%BB%99t%20t%C3%ADnh%20n%C4%83ng%20m%E1%BA%A1nh%20m%E1%BA%BD%20trong%20React,g%E1%BB%8Dn%20v%C3%A0%20d%E1%BB%85%20%C4%91%E1%BB%8Dc%20h%C6%A1n>.

6.3. Quản lý state bằng Recoil

6.31. Recoil là gì?

Recoil chính là 1 thư viện dùng để quản lý state của React, bên cạnh 1 loạt thư viện hầm hố khác như: redux, mobX, flux, … hay thậm chí là context của chính React. Chà, thấy nhiều quá cũng hoang mang nhỉ, nhưng đừng lo, hãy cùng mình đánh giá và so sánh giữa Recoil với Redux nhé (vì thằng Redux này nổi tiếng nhất).

**Redux**

Ưu điểm:

Quá nổi tiếng, quá phổ biến.

Sử dụng global state (tức là nguyên app sẽ có 1 cái state bự).

Nhược điểm:

Chính vì nó là global state và cơ chế immutable, nên mỗi khi có 1 thay đổi nhỏ, nguyên cái global state đó sẽ được clone ra, cập nhật thay đổi, thay vào chỗ state cũ, sau đó là xóa state cũ ==> Sẽ có vấn đề về hiệu năng app khi cái global state của bạn quá bự và thường xuyên cập nhật.

Sử dụng hơi lằng nhằng (khá cần mấy cái middleware đi kèm).

**Recoil**

Ưu điểm:

Cú pháp gần gũi, dễ sử dụng (do giống với state mặc định trong React).

Sử dụng các state riêng lẻ (gọi là các atom nhé, phân biệt bằng key).

Hỗ trợ luôn quản lý state bất đồng bộ (không cần middleware rườm rà như Redux nhé).

Của facebook. 😃

Nhược điểm:

Khá mới mẻ, ít người biết và sử dụng.

Mới chỉ là experimental ==> chưa ổn định ==> sử dụng vào production thì nên cân nhắc nhé.

6.3.2. Các API chính của Recoil

**Recoil Root**

Là thằng bự nhất bọc bên ngoài để có thể sử dụng Recoil bên trong. (giống Context Provider của React hoặc là Provider trong Redux ấy, nhưng không cần cung cấp store lằng nhằng gì đâu nhé).

Có thể có nhiều root cùng tồn tại, và khi nested thì thằng trong cùng sẽ là thằng có tác dụng nhé.

ReactDOM.render(

<RecoilRoot>

<App />

</RecoilRoot>,

document.getElementById('root')

);

**Atom**

Nó giống như state trong React ấy. Function này sẽ trả về recoil state cho bạn. Và cũng như react state, khi recoil state thay đổi thì các component đang sử dụng nó cũng sẽ được thay đổi.

const listTodoState = atom({

key: 'listTodo', // là unique string, bắt buộc phải có nhé.

default: [], // giá trị mặc định khi khởi tạo.

});

**Selector**

Cũng trả về recoil state luôn nhé. Nhưng điểm khác với atom ở đây chính là selector sẽ nhận vào pure function và xử lý state từ atom để tạo ra 1 state biến đổi khác (giống như selector trong React-Redux ấy, nghe có vẻ rắc rối, nhưng đi vào ví dụ là hiểu ngay nè).

const newListState = selector({ // newListState này sẽ chứa danh sách các hành động có trạng thái là new.

key: 'newList', // cũng là unique string và bắt buộc phải có.

get: ({ get }) => { // đây là pure function, cũng bắt buộc phải có.

const list = get(listTodoState); // đây là cách để lấy cả list todo đã tạo với atom ở bước trên.

return list.filter((item) => item.status === 'new'); // chọn và trả về những thằng có status là new.

}

});

Hỗ trợ cả bất đồng bộ luôn

export const pkmIdState = atom({

key: 'currentPkmId',

default: 1,

});

export const pkmDetailState = selector({

key: 'pkmDetail',

get: async ({ get }) => {

const id = get(pkmIdState);

if (!id) {

return;

}

try {

const res = await fetch(`https://pokeapi.co/api/v2/pokemon/${id}`);

const pkm = await res.json();

return pkm;

} catch (err) {

return null;

}

},

});

**Hooks**

useRecoilState: giống y change useState của React luôn, nhận vào recoil state, trả về giá trị của state cùng với hàm để thay đổi giá trị state.

const [todoList, setTodoList] = useRecoilState(listTodoState);

useRecoilValue: cũng nhận vào recoil state, nhưng khác cái là chỉ trả về giá trị của state - dùng để truy xuất giá trị state.

const todoList = useRecoilValue(listTodoState);

useSetRecoilState: giống cái useRecoilState, nhưng chỉ trả về hàm để thay đổi giá trị state.

const setTodoList = useRecoilValue(listTodoState);

useResetRecoilState: trả về hàm để reset state về giá trị default.

const resetTodoList = useResetRecoilState(newListState);

6.3.3. Code cái Todo-App với Recoil thử nào

Khởi tạo một React project với create-react-app

npx create-react-app recoil-example

Vào trong project vừa khởi tạo và cài Recoil thôi nào

cd ./recoil-example && yarn add recoil && yarn start

Kế tiếp, mình sẽ tạo thư mục recoil trong src, để chứa các recoil state, dạng thế này

Cấu trúc thư mục

Sau đó, mình sẽ tạo file listState.js trong folder recoil (như hình trên), dùng để chứa list todo state.

import { atom } from 'recoil';

const defaultData = [

{

id: 1,

content: 'Action 1',

status: 'new',

},

{

id: 2,

content: 'Action 2',

status: 'inprogress',

},

];

const listTodoState = atom({

key: 'listTodo',

default: defaultData,

});

Tiếp đó là tạo new list state để trả về danh sách những action có status là new

export const newListState = selector({ // newListState này sẽ chứa danh sách các action có trạng thái là new.

key: 'newList',

get: ({ get }) => {

const list = get(listTodoState); // đây là cách để lấy cả list todo đã tạo với atom ở bước trên.

return list.filter((item) => item.status === 'new'); // chọn và trả về những thằng có status là new.

}

});

Làm tương tự để tạo in-progress list và completed list thôi

export const inprogressListState = selector({ // inprogressListState này sẽ chứa danh sách các action có trạng thái là inprogress.

key: 'inprogressList',

get: ({ get }) => {

const list = get(listTodoState);

return list.filter((item) => item.status === 'inprogress');

}

});

export const completedListState = selector({ // completedListState này sẽ chứa danh sách các action có trạng thái là completed.

key: 'completedList',

get: ({ get }) => {

const list = get(listTodoState);

return list.filter((item) => item.status === 'completed');

}

});

Giờ thì tạo 3 components: NewList, InProgressList và CompletedList để hiện 3 danh sách tương ứng ra màn hình

import React from 'react';

import { useRecoilValue } from 'recoil';

import { newListState } from './recoil/listState';

const NewList = () => {

const newList = useRecoilValue(newListState); // ở đây chỉ hiện data, nên useRecoilValue là đủ.

return (

<div className='col'>

<h3>New</h3>

<ul>

{newList.map((item) => (

<li key={item.id}>

{item.content}

<button>In-Progress</button>

</li>

))}

</ul>

</div>

);

};

export default NewList;

import React from 'react';

import { useRecoilValue } from 'recoil';

import { inprogressListState } from './recoil/listState';

const InProgressList = () => {

const inProgressList = useRecoilValue(inprogressListState);

return (

<div className='col'>

<h3>In-Progress</h3>

<ul>

{inProgressList.map((item) => (

<li key={item.id}>

{item.content}

<button>Completed</button>

</li>

))}

</ul>

</div>

);

};

export default InProgressList;

import React from 'react';

import { useRecoilValue } from 'recoil';

import { completedListState } from './recoil/listState';

const CompletedList = () => {

const completedList = useRecoilValue(completedListState);

return (

<div className='col'>

<h3>Completed</h3>

<ul>

{completedList.map((item) => (

<li key={item.id}>{item.content}</li>

))}

</ul>

</div>

);

};

export default CompletedList;

Chỉnh App.js lại để sử dụng 3 components vừa tạo, và cùng với đó là bọc cái RecoilRoot vào vì mình có sử dụng Recoil bên trong nhé

import React from 'react';

import { RecoilRoot } from 'recoil';

import NewList from './NewList';

import InProgressList from './InProgressList';

import CompletedList from './CompletedList';

import './App.css';

function App() {

return (

<RecoilRoot>

<div className='App'>

<header className='App-header'>

<h1>To-do List</h1>

</header>

<div className='content'>

<NewList />

<InProgressList />

<CompletedList />

</div>

</div>

</RecoilRoot>

);

}

export default App;

Chà, vậy là đã hiện được cái danh sách lên màn hình rồi. Tiếp tới là phải thêm những hành động mới vào cái list to-do này chứ nhỉ. Đầu tiên là thêm hàm set vào newListState để thêm 1 hành động vào danh sách

export const newListState = selector({

key: 'newList',

get: ({ get }) => {

const list = get(listTodoState);

return list.filter((item) => item.status === 'new');

},

set: ({ get, set }, newValue) => {

const list = get(listTodoState);

// tạo 1 cái hành động mới

const newTodo = {

id: new Date().getTime(), // id tạo vậy để khỏi lo bị duplicate

content: newValue, // tên hành động được truyền vào

status: 'new', // hành động mới nên có status là new

};

set(listTodoState, [...list, newTodo]); // hàm set dùng để thay đổi giá trị của recoil state từ atom

},

});

Giờ thì tạo 1 cái component để người dùng nhập nội dung hành động thôi

import React, { useState } from 'react';

import { useSetRecoilState } from 'recoil';

import { newListState } from './recoil/listState';

const NewActionInput = () => {

const addNewTodo = useSetRecoilState(newListState); // dùng useSetRecoilStat do chỉ set chứ không cần đọc

const [text, setText] = useState(''); // dùng nội bộ trong component thì xài state của React là đủ

const handleAddNewAction = () => {

if (!text) {

return;

}

addNewTodo(text); // đây là chỗ truyền giá trị cho hàm set ở newListState lúc nãy (cái biến newValue á)

setText('');

};

const handleChangeText = (e) => {

setText(e.target.value);

};

return (

<div className='action-input'>

<input

type='text'

placeholder='Enter new to-do action'

value={text}

onChange={handleChangeText}

/>

<button onClick={handleAddNewAction}>Add</button>

</div>

);

};

export default NewActionInput;

Thêm cái NewActionInput vào App.js luôn để còn xài chứ

<header className='App-header'>

<h1>To-do List</h1>

</header>

<NewActionInput />

<div className='content'>

<NewList />

<InProgressList />

<CompletedList />

</div>

Giờ mình sẽ thêm chức năng chuyển trạng thái từng hành động từ new -> in-progress -> completed nhé. Nếu bạn để ý thì trong component NewList và InProgressList đã có tạo sẵn button trong đó rồi, nên tiếp theo là tạo hàm set cho inprogressListState và completedListState

export const inprogressListState = selector({

key: 'inprogressList',

get: ({ get }) => {

const list = get(listTodoState);

return list.filter((item) => item.status === 'inprogress');

},

set: ({ get, set }, id) => { // để set 1 cái hành động có id này thành trạng thái inprogress

const list = get(listTodoState);

set(

listTodoState,

list.map((item) =>

item.id === id ? { ...item, status: 'inprogress' } : item

)

);

},

});

export const completedListState = selector({

key: 'completedList',

get: ({ get }) => {

const list = get(listTodoState);

return list.filter((item) => item.status === 'completed');

},

set: ({ get, set }, id) => { // để set 1 cái hành động có id này thành trạng thái completed

const list = get(listTodoState);

set(

listTodoState,

list.map((item) =>

item.id === id ? { ...item, status: 'completed' } : item

)

);

},

});

Gắn vào cái button trong 2 cái component NewList và InProgressList

import React from 'react';

import { useRecoilValue, useSetRecoilState } from 'recoil';

import { newListState, inprogressListState } from './recoil/listState';

const NewList = () => {

const newList = useRecoilValue(newListState);

const setInprogress = useSetRecoilState(inprogressListState);

const handleClick = (id) => () => {

setInprogress(id); // truyền id vào

};

return (

<div className='col'>

<h3>New</h3>

<ul>

{newList.map((item) => (

<li key={item.id}>

{item.content}

<button onClick={handleClick(item.id)}>In-Progress</button>

</li>

))}

</ul>

</div>

);

};

export default NewList;

import React from 'react';

import { useRecoilValue, useSetRecoilState } from 'recoil';

import { inprogressListState, completedListState } from './recoil/listState';

const InProgressList = () => {

const inProgressList = useRecoilValue(inprogressListState);

const setCompleted = useSetRecoilState(completedListState);

const handleClick = (id) => () => {

setCompleted(id); // truyền id vào

};

return (

<div className='col'>

<h3>In-Progress</h3>

<ul>

{inProgressList.map((item) => (

<li key={item.id}>

{item.content}

<button onClick={handleClick(item.id)}>Completed</button>

</li>

))}

</ul>

</div>

);

};

export default InProgressList;

Tham khảo: <https://viblo.asia/p/recoil-thu-vien-quan-ly-state-cho-react-924lJPB6KPM>

6.4. Ví dụ minh họa