TRƯỜNG ĐẠI HỌC KHOA HỌC

**KHOA CÔNG NGHỆ THÔNG TIN**



BÁO CÁO MÔN HỌC

**THỰC TẬP VIẾT NIÊN LUẬN**

**REACTJS**

**Giáo viên hướng dẫn:** **Sinh viên thực hiện**:

**Ths.Trần Nguyên Phong** - Trần Đình Dương

**Nhóm** : 2

**Lớp**: TIN3142.002

**Năm học**: 2023-2024

***Huế, 8/2024***

MỤC LỤC

[MỞ ĐẦU 1](#_Toc174617405)

[I. Đặt vấn đề: 1](#_Toc174617406)

[II. Mục tiêu: 1](#_Toc174617407)

[NỘI DUNG 2](#_Toc174617408)

[I. React ES6: 2](#_Toc174617409)

[1. Arrow function: 2](#_Toc174617410)

[2. Destructuring: 2](#_Toc174617411)

[3. Spread Operator 2](#_Toc174617412)

[4. JS modules: 4](#_Toc174617413)

[II. ReactDOM 4](#_Toc174617414)

[1. document.createElement: 4](#_Toc174617415)

[2. React.createElement: 5](#_Toc174617416)

[3. ReactDOM: 5](#_Toc174617417)

[III. JSX 6](#_Toc174617418)

[IV. Component 6](#_Toc174617419)

[V. Props 7](#_Toc174617420)

[VI. Children Props 8](#_Toc174617421)

[VII. React event 9](#_Toc174617422)

[VIII. React Hooks 9](#_Toc174617423)

[1. useState hook: 10](#_Toc174617424)

[2. useEffect hook: 10](#_Toc174617425)

[3. useLayoutEffect hook: 12](#_Toc174617426)

[4. useRef hook: 13](#_Toc174617427)

[5. React.memo: 15](#_Toc174617428)

[6. useCallback Hook: 16](#_Toc174617429)

[7. useMemo Hook 19](#_Toc174617430)

[8. useReducer Hook: 22](#_Toc174617431)

[9. React useContext hook 25](#_Toc174617432)

[IX. Cài đặt SASS 26](#_Toc174617433)

[X. React CSS Styling: 26](#_Toc174617434)

[1. Inline styling: 26](#_Toc174617435)

[2. CSS styleSheets 27](#_Toc174617436)

[3. CSS module: 27](#_Toc174617437)

[XI. React Sass Styling: 28](#_Toc174617438)

[KẾT LUẬN 30](#_Toc174617439)

[PHỤ LỤC 31](#_Toc174617440)

MỞ ĐẦU

1. Đặt vấn đề:

Hiện nay, nhu cầu xây dựng các ứng dụng web tương tác với người dùng cà mượt mà ngày càng trở nên cần thiết. Với yêu cầu các ứng dụng web hoạt động nhanh chóng, giao diện người dùng thân thiện, dễ tiếp cận, sử dụng và có khả năng tương tác cao. Để đáp ứng những yêu cầu này, cần có những công cụ hỗ trợ mạnh mẽ.

Do đó, ReactJs là một tư viện JavaScript mã nguồn mở do Facebook phát triển, đã nhanh chóng trở thành một trong những công cụ phổ biến và được các nhà phát triển sử dụng rộng rãi để xây dựng giao diện người dùng

Bài nghiên cứu này nhằm mục đích cung cấp một cái nhìn tổng quan về ReactJs, giúp bản thân có thể tiếp thu các kiến thức, nguyên lý cơ bản về ReactJs. Thông quá quá trình nghiên cứu, mong muốn sẽ hiểu rõ hơn về cách sử dụng ReactJs một cách hiệu quả và giải quyết các vấn đề phổ biến khi làm việc với thư viện, từ đó bổ sung kiến thức và cải thiện kỹ năng của bản thân trong lĩnh vực phát triển ứng dụng website.

1. Mục tiêu:

Nắm được kiến thức của ReactJs như ReactJs ES6, React DOM và cấu trúc của một ứng dụng ReactJs, các kỹ thuật cần thiết để sử dụng ReactJs như sử dụng JSX, Props, các Hooks như useState, useEffect, useLayoutEffect, useRef, memo, useCallback, useMemo use reducer, useContext, … Và cuối cùng là cách React dùng CSS

NỘI DUNG

1. **React ES6:**
2. **Arrow function:**

* Arrow Function có thể tạo ra những hành vi thuộc tính this. Thường thì sẽ xảy ra những thay đổi ngay sau mỗi lần tạo Closure. Việc sử dụng Arrow Function trong phiên bản ES6 sẽ cho phép việc thực hiện lập trình mà không cần phải sử dụng that = this hoặc self = this hoặc .bind(this) hay là \_this + that.
* Trong ES6 khi kết hợp được các hàm cũ với nhau và khi một Arrow Function được sử dụng với một dòng lệnh thì nó sẽ trở thành biểu thức ấy. Bạn cần lưu ý sử dụng đến Return một cách rõ ràng đối với trường hợp bạn có nhiều hơn một dòng.
* Không dùng Arrow function

Function sum (a, b) {

return a+b;

}

* Dùng Arrow function

Const sum = (a, b) => a+b

1. **Destructuring:**

* Destructuring Assignment trong ES6 sẽ thực hiện phân những giá trị trong mảng vào các biến dựa theo thứ tự nhất định. Có thể nói theo cách khác là Destructuring Assignment sẽ tách ra các phần tử Array hoặc Object chỉ bằng một đoạn code nhất định để chia thành nhiều biến nhỏ.
* Không dùng Destructuring Parameters

const vehicles = [‘mustang’, ‘f-150’, ‘expedition’];

const car = vehicles[0];

const truck = vehicles[1];

const suv = vehicles[2];

* Dùng Destructuring Parameters

const vehicles = [‘mustang’, ‘f-150’, ‘expedition’];

const [car, truck, suv] = vehicles;

* Ta có thể thấy code ngắn gọn hơn

1. **Spread Operator**

* Spread là một tính năng giúp người lập trình có thể xử lý mảng và đối tượng một cách linh hoạt. Có thể được sử dụng để mở rộng các phần tử của một mảng hoặc các thuộc tính của một đối tượng.
* Một số cách sử dụng phổ biến:
  + Copy mảng:

const vehicles = [‘mustang’, ‘f-150’, ‘expedition’];

const vehiclesCopy = […vehicles];

Lúc này mảng vehiclesCopy sẽ mang các giá trị như mảng vehicles (vehiclesCopy = [‘mustang’, ’f-150’, ‘expedition’])

* Kết hợp mảng:

const vehicles1 = ['mustang', 'f-150', 'expedition'];

const vehicles2 = ['ferrari 911', 'merc C300", 'audi R8'];

const newvehicles = [...vehicles1, ...vehicles2];

Lúc này mảng newVehicles sẽ mang giá trị của mảng vehicles1 và vehicles2 (newVehicles=[ 'mustang', 'f-150', 'ferrari 911', 'merc C300', 'audi R8'])

* Tương tự với các đối tượng:
  + Copy đối tượng:

const originalobject = {

a: 1, b: 2

｝；

const newobject = { ...originalobject };

Lúc này đối tượng newObject sẽ mang các giá trị như mảng originalObject (newObject = {a:1, b:2};)

* Kết hợp đối tượng:

const object1 = {a: 1, b: 2};

const object2 = {с: 3, d: 4};

const combinedobject - {...object1, ...object2 };

Lúc này mảng combinedObject sẽ mang giá trị của mảng object1 và object2 (combinedObject = {a:1, b:2, c:3, d:4})

* Ngoài ra chúng ta có thể truyền tham số vào hàm:

function sum (x, y, z) {return x + y + z;}

const numbers = [1, 2, 3];

sum(...numbers);

Tất cả các phần tử trong mảng numbers (1, 2, 3) sẽ được truyền vào hàm sum thực hiện tính tổng các phần thử

1. **JS modules:**

* JS modules cho phép chia nhỏ code thành những file nhỏ, điều này có thể giúp quản lí code dễ dàng, tái sử dụng nhiều lần.
* Các modules được nhập từ các tệp bên ngoài bằng câu lệnh “import”, các modules cũng phải dựa vào type=”module” trong thẻ <script>
* Để nhập một module thì phải xuất module đó trước khi nhập bằng câu lệnh “Export”
* Có hai kiểu exports đó là Named Exports và Default Exports
  + Named Export: có 2 kiểu xuất Named Export là xuất dòng riêng lẻ và xuất một lần ở cuối

// xuất dòng riêng lẻ

export const name = "Jesse";

export const age = 40;

// xuất một lần ở cuối

const name = "Jesse";

const age = 40;

export {name, age};

* + Default exports:
* Có hai kiểu import dựa trên hai kiểu xuất Named Exports và Default Exports

// Import named exports

import Paragraph from "./Paragraph"

// Import a default export

import {name, age} from "./person.js";

1. **ReactDOM**
2. **document.createElement:**

* Giúp người lập trình có thể tạo ra một DOM element, người lập trình có thể thêm element này vào trong DOM để tạo ra giao diện người dùng

const h1= document.createElement ('h1')

h1. innerText = "Hello world!"

* Đoạn mã trên tạo ra element có thẻ là h1 và gán giá trị là “Hello world!”. Nhưng element này đang nằm trong bộ nhớ và chưa được gắn vào DOM.

const h1= document createElement ('h1')

h1. innerText = "Hello world!"

document. body appendchild (h1)

* Để gắn element vào DOM ta cần khối lệnh như trên, câu lệnh document.body.appendChild(h1) gắn h1 (element vừa tạo) vào trong DOM
* Từ đó có thể xuất ra giao diện cho người dùng.
* Có thể thêm các properties như sau

const h1= document. createElement ('h1');

h1. innerText = "Hello world!";

h1. title='Hello';

h1. className='heading';

document.body.appendchild(h1);

1. **React.createElement:**

* Tạo ra React element khác với document.createElement đó là tạo ra DOM element.
* React.createElement có cú pháp như sau với type là element muốn tạo, props là các thuộc tính, children là con của element muốn tạo.

const element = React.createElement(type, props, …children)

* Ví dụ:

const h1 = React createlement("h1',{

title: 'Hello',

className: 'heading'

}, "Hello world!')

* Khối lệnh trên tạo ra thẻ h1 với các thuộc tính title: ”Hello”, className: ”heading”, với con là “Hello world!”
* React Element trả về một đối tượng
* Để render React element lên giao diện trình duyệt phải thông qua ReactDOM

1. **ReactDOM:**

* ReactDOM là một thư viện, có vai trò như là một cầu nối giữa react và DOM
* Để render một react element Sử dụng ReactDOM.render(h1, root)

const h1 = React.createElement('h1',

title: 'Hello',

className: 'heading'

}, "Hello world!')

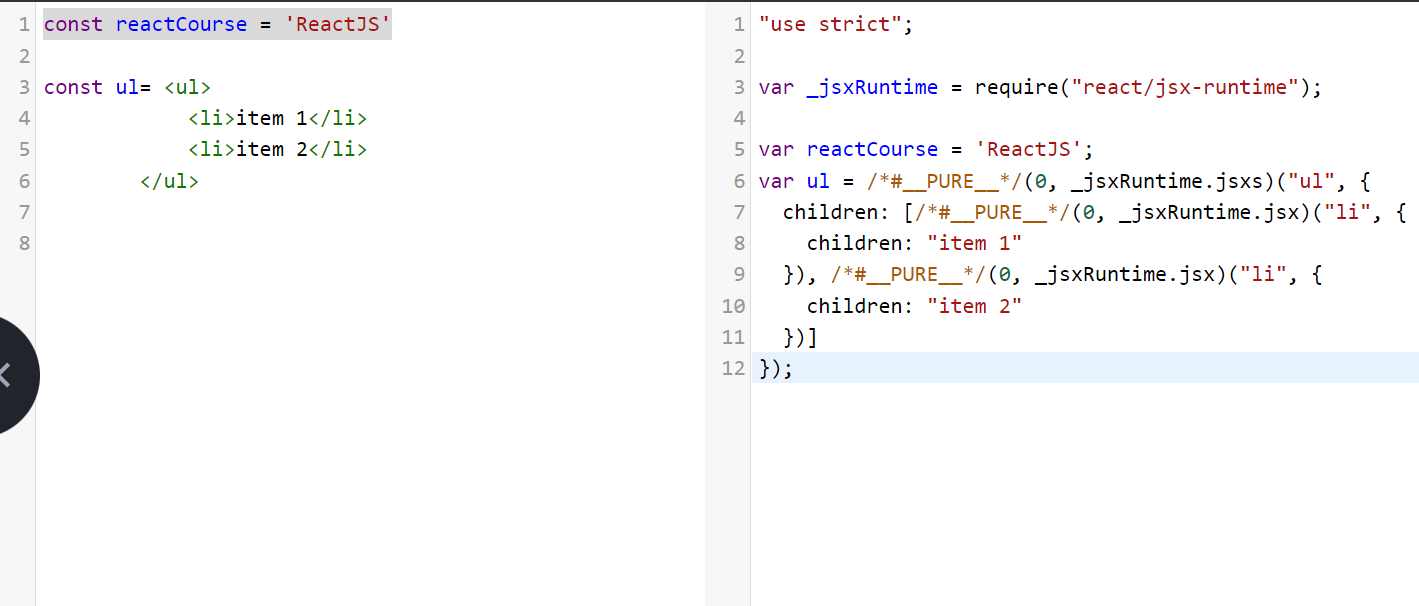
const root = document.getElementById('root')

ReactDOM. render (h1, root)

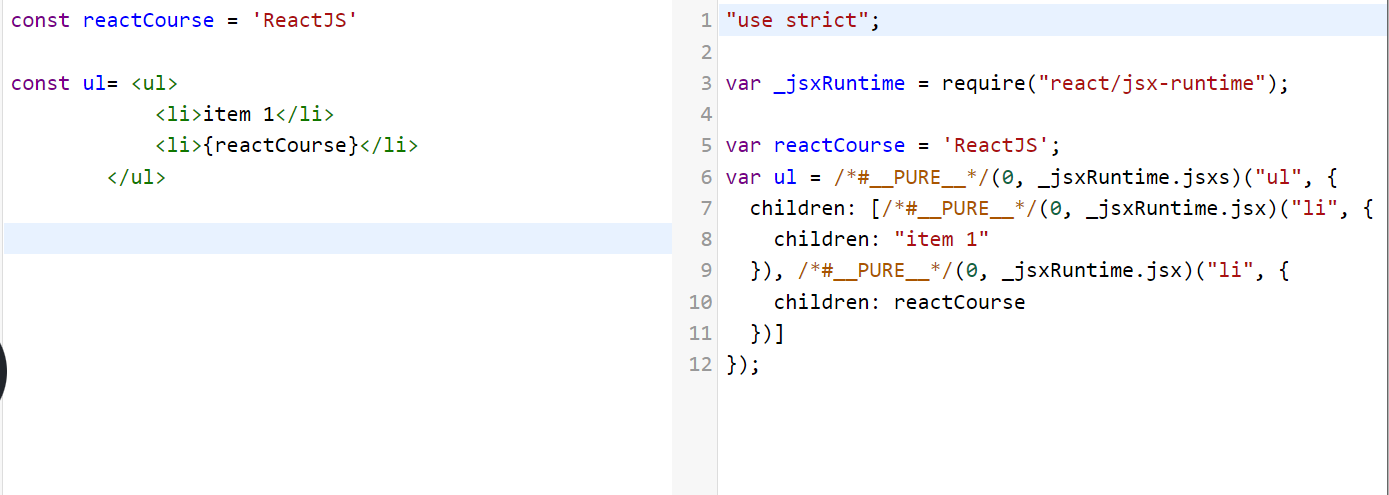
* Trong đó root là nút root trong DOM được gán vào root và h1 là react element muốn tạo
* ReactDOM.render (h1, root) gắn h1 vào root và render lên giao diện.

1. **JSX**

* JSX là Javascript XML. Javascript XML cho phép viết được XML, HTML trong file Javascript
* JSX truyền trực tiếp ReactDOM thông qua thư viện Babel.
* Thư viện Babel có chức năng chính là phân tích cú pháp và chuyển đồi JSX trở thành Javascript ES5
* Ví dụ về thư viện babel



* Ngoài ra có thể viết xen lẫn Javasc ript vào JSX bằng cách sử dụng cặp dấu {}



* Truyền biến reactCourse vào code JSX

1. **Component**

* Component giống như các hàm trả về cấc phần tử element.
* Các Component là các đoạn mã độc lập, có thể tái sử dụng bằng cách gọi lại nhiều lần. Các Component có chức năng tương tự các hàm JavaScript nhưng lại trả về các phần tử HTML
* Có hai loại Component đó là Function Component và Class Component
  + Class Component: phải bao gồm câu lệnh mở rộng React.Component. Component này yêu cầu phương thức render(), phương thức này trả về HTML.

class Car extends React. Component {

render (){

return <h2>Hi, I am a Car!</h2>;

}

}

* + Function Component: Function component cũng trả về HTML và hoạt động tương tự Class Component, Function component được sử dụng rộng rãi hơn từ khi có hook có thể hỗ trợ function component làm các chức năng phức tạp hơn như class component

function Car) {

return <h2>Hi, I am a Car!</h2>;

}

1. **Props**

* Props là một object chứa các thuộc tính để chúng ta có thể mô tả cho react element
* Đối với React elements, cách truyền props giống như các attribute của thẻ HTML

const app = (

<div className= 'wrapper'>

<Header />

<div className="content"›Content</div>

<div className="Footer">footer</div>

</div>

);

* + Đặt tên phải tuân theo các quy ước có sẵn (className, htmlFor, ...)
* Đối với React components, sử dụng props giống đối số cho function

function app(){

return(

<div id= 'wrapper'

<PostItem

title="de men phieu luu ky"

image="https://tse3.mm.bing.net/th?id=0IP.EcgX9PbLdJYS-BAgY46eEQHaKs&pid=Api"

description= "de men phieu luu ky"

/>

</div>

)

}

* + Tên props được quy ước đặt theo camelCase, tự do đặt tên props
* Lưu ý:
  + Prop key là prop đặc biệt
  + Props cơ bản là đối số của Cpmponent nên props có thể là bất cứ kiểu dữ liệu gì và có thể dùng destructuring để truyền props

1. **Children Props**

function Button({ title }) {

return ‹button>{title}</button>

}

function App() {

return <div id="wrapper">

<Button>Click me!</Button> </div>

)

}

* Xét ví dụ trên, sẽ có lỗi hiển thị nút từ component Button.
* Khi sử dụng cặp thẻ <Button>Click me!</Button> thì Click me! trở thành prop children
* Để lấy và truyền prop children vào component ta làm như sau:

function Button({ children }){

return <button>{children}</button>

｝

function App() {

return (

<div id="wrapper">

‹Button>Click me!</Button>

</div>

)

}

* Props children sẽ mang giá trị là Click me! được truyền vào component Button và được render ra màn hình như mong muốn.

1. **React event**

* So với DOM event, các React event được viết theo cú pháp camelCase: onClick thay vì onclick
* Các trình xử lý sự kiện được viết trong dấu {} (vd: onClick ={shoot} thay vì onclick=shoot() )
* React:

<button onClick={shoot}>Take the Shot!</button>

* HTML:

<button onClick=”shoot()”>Take the Shot!</button>

* Các DOM event tham khảo: <https://www.w3schools.com/jsref/dom_obj_event.asp>
* Áp dụng các quy tắc trên cùng bảng DOM event ta có thể sử dụng React event.

1. **React Hooks**

* Hooks cho phép các function components truy cập vào trạng thái và các React features khác. Vì vậy có thể thực hiện các chức năng tương tự class components
* Hooks cho phép người lập trình “nối” vào các react features
* Một số React Hooks thường dùng:
  + useState
  + useEffect
  + useContext
  + useRef
  + useReducer
  + useCallback
  + useMemo
* Cách import Hooks từ react.

import React, { useState } from “react”;

* Ví dụ trên đang import Hook useState để theo dõi trạng thái.
* Hooks chỉ có thể được gọi trong function components
* Hooks chỉ có thể được gọi ở mức cao nhất của component
* Hooks không được có điểu kiện:
* Các hooks

1. **useState hook:**

* Để sử dụng useState, trước tiên chúng ta phải import vào component

import { useState } from “react”;

* Khai báo useState:
  + Khai báo useState bằng cách gọi trong funtion component
  + useState hook nhận 2 giá trị là trạng thái hiện tại và phương thức có chức năng để cập nhật trạng thái
  + Khi khai báo phải khai báo ở đầu component

import {useState} from “react”;

function FavoriteColor(){

const [color, setColor] = useState(“”);

}

* + Ví dụ trên là cách khai báo useState với câu lệnh import ở đầu component, useState nhận 2 tham số là “color” – trạng thái hiện tại đang chọn (ban đầu là rỗng) và phương thức setColor có chức năng cập nhật lại trạng thái cho biến “color”.
  + Sau khi khai báo chúng ta có thể sử dụng color như một biến javascript để truy xuất trạng thái hiện tại.
  + Không bao giờ nên cập nhật trực tiếp trạng thái ví dụ như color= “red”. Cập nhật trạng thái phải thông qua phương thức ví dụ như setColor(“red”)

1. **useEffect hook:**

* useEffect Hook cho phép thực hiện các “side effect” trong component (“side effect” là tác động trong chương trình phần mềm mà khi tác động đó xảy ra dẫn tới việc dữ liệu chương trình bị thay đổi)
* Để sử dụng useEffect, phải import vào component

import { useEffect } from “react”;

* Cú pháp sử dụng useEffect

useEffect(callback,[dependence])

* Trong đó callback là hàm để thực hiện sideEffect và là đối số bắt buộc. Callback luôn được gọi sau khi component được mount
* Các dependence là mảng chứa các biến dữ liệu thể hiện sự phụ thuộc về mặt dữ liệu, mảng dependence là dối số không bắt buộc
* Có 3 trường hợp sử dụng đối số mảng dependence
  + Không có dependence:

useEffect (()=>{

// Runs on every render

});

* + - Ở trường hợp này, callback luôn được gọi mỗi khi component re-render
    - Callback được gọi sau khi component thêm element vào DOM
  + Dependence là mảng rỗng:

useEffect(()=>{

//Runs only on the first render

},[]);

* + - Chỉ gọi 1 lần sau khi component được mount
  + Dependence là mảng chứa phần tử props và state

useEffect(()=>{

//Runs only on the first render

//And any time any dependency value changes

},[props, state]);

* + - Callback được gọi khi compponent được mount và được gọi lại mỗi khi dependence (props hoặc state) thay đổi.
    - Khi component re-render useEffect kiểm tra denpendence (props hoặc state) trước và sau khi render, nếu khác nhau thì mới gọi đến callback
* **Effect Cleanup:**
  + Effect Cleanup giúp dọn dẹp và giảm rò rỉ bộ nhớ
  + Timeouts, subscriptions, event listeners, và một số effects khác sẽ bị loại bỏ khi không còn cần thiết
  + Effect Cleanup luôn được gọi trước khi component unmount

import i usestate, useEffect } from "react";

import ReactDOM from "react-dom/client";

function Timer() {

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

useEffect(() =› {

let timer = setTimeout (() => {setcount((count) => count + 1);

},1000;

return () => clearTimeout(timer)

}, []);

return ‹h1>I've rendered {count) times!</h1>;

}

const root = ReactDoM. createRoot (document.getElementById(“root"));

root. render (<Timer/>);

* + Như ví dụ trên, hàm được trả về của callback trong useEffect được gọi là cleanup function

return () => clearTimeout(timer)

* + Hàm này clearTimeOut khi setTimeOut không còn cần thiết nữa

1. **useLayoutEffect hook:**

* Cách thức hoặt động của useLayoutEffect gần giống useEffect, đều thực hiện các side effect
* Để sử dụng useLayoutEffect, phải import vào component

import { useLayoutEffect } from “react”;

* Cú pháp sử dụng useLayoutEffect
* useLayoutEffect(callback,[dependence])
* useLayoutEffect và useEffect khác nhau ở thứ tự hoạt động:

|  |  |
| --- | --- |
| **useEffect** | **useLayoutEffect** |
| * Cập nhật lại state * Cập nhật lại DOM * Render lại giao diện * Gọi cleanUp function nếu dependence thay đổi * Gọi callback của useEffect | * Cập nhật lại state * Cập nhật lại DOM * Gọi cleanUp function nếu dependence thay đổi * Gọi callback của useLayoutEffect * Render lại giao diện |

* Điều này có thể giúp giải quyết các logic bằng các gọi callback trước khi render lại giao diện, từ đó giảm đi độ trễ khi cập nhật lại giao diện cải thiện trải nghiệm của người dùng.

1. **useRef hook:**

* useRef giúp chúng ta có thể lưu một giá trị bất kì qua một tham chiếu bên ngoài component, có thể lưu một giá trị thay đổi mà không gây ra hiện tượng re-render
* Để sử dụng useRef, phải import vào component

import { useRef } from “react”;

* Cách sử dụng useRef: gọi phương thức useRef()
* Ví dụ về sử dụng useRef:

import { usestate, useEffect, useRef } from "react";

import ReactDOM from "react-dom/client";

function App() {

const [inputValue, setInputValue] = usestate(“ ”);

const count = useRef(0);

useEffect(() => {

count. current = count. current + 1;

});

return (

<>

<input

type="text"

value={ inputvalue)}

onChange={ (e) => setInputvalue(e.target.value) }

/>

<h1>Render Count: {count.current)</h1>

</>

}

const root = ReactDoM. createRoot (document.getElementById(“root”));

root.render(<App />);

* Sử dụng useRef để đếm số lần render giao diện bằng cách sử dụng useRef, nếu sử dụng useState trong trường hợp này có thể dẫn tới lặp vô hạn.
* useRef trả về môt đối tượng current
* Có thể sử dụng useRef để truy cập các DOM Element

import { useRef } from "react";

import ReactDOM from "react-dom/client";

function App) {

const inputElement = useRef);

const focusInput = () => { inputElement.current.focus);

};

return(

<>

<input type="text" ref={inputelement} />

<button onclick={focusInput}>Focus Input</button>

</>

);

const root = ReactDOM.createRoot(document.getElementById(‘root’));

root.render (<App />);

* Với ví dụ trên khi ấn nút Focus Input thì có thể focus vào ô để nhập input
* Trong React, có thể thêm thuộc tính ref vào một phần tử để dùng useRef truy cập vào DOM như ví dụ trên.
* useRef có thẻ được sử dụng để theo dõi các giá trị giữa các lần thay đổi trạng thái bằng cách kết hợp useState, useEffect.

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

import ReactDoM from "react-dom/client";

function App) {

const [inputValue, setInputValue] = usestate(“”);

const previousInputValue = useRef (“”);

useEffect(() => {

previousInputValue.current = inputvalue;

}, [inputvalue]);

return(

<>

<input

type="text"

value={inputvalue}

onChange={(e) => setInputvalue(e.target.value)}

/>

<h2>Current Value: {inputvalue)</h2>

<h2>Previous Value: {previousInputValue.current)</h2>

</>

);

}

const root = ReactDoM. createRoot (document-getElementById(\*root'));

root.render (<App />);

1. **React.memo:**

* memo() là một Higher Order Component là một component bậc cao. Sử dụng memo sẽ khiến React bỏ qua việc render component nếu như các props không có sự thay đổi, có thể tăng hiệu suất.
* Để sử dụng memo, phải import vào component

import {memo} from “react”;

* Cú pháp sử dụng memo:

export default memo(Todos);

* Xét ví dụ:

import { useState } from “react”

import ReactDOM from ”react-dom/client”;

import Todos from “./Todos”;

const App = () => {

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

const [todos, settodos] = usestate([“todo 1”, “todo 2”]);

const increment = () => {

setcount ( (c) => c + 1);

};

return(

<>

<Todos todos={todos) />

<hr />

‹div>

Count: {count)

<button onclick={increment}>+</button>

</div>

</>

);

}

const root = ReactDOM. createRoot (document.getElementById(‘root’));

root.render (<App />);

* Ứng dụng đếm số trên, mỗi lần biến count được tăng lên component được re-render nhưng component Todos là component con cũng phải re-render gây tốn tài nguyên và không tối ưu hiệu năng

import { memo } from “react”;

const Todos = ({ todos }) => {

console.log(“child render”);

return (

<>

<h2>My Todos</h2>

{todos. map ((todo, index) => {

return <p key={index}>{todo}</p>;

})}

</>

);

};

export default memo (Todos);

* Vì thế ta sử dụng memo ở component Todos có thể giải quyết vấn đề trên.
* Khi component App re-render, vì các props của component Todos không có sự thay đổi nên component Todos không re-render, giải quyết vấn đề tối ưu hiệu năng.

1. **useCallback Hook:**

* useCallback hook giúp tránh tạo các hàm mới không cần thiết trong function component không cần thiết
* useCallback lưu callback của hook ra ngoài và trả về hàm đó
* Để sử dụng useCallback, trước tiên phải import vào component

import { useCallback } from “react”;

* Cú pháp sử dụng useCallback

useCallback(callback(),[dependence])

* useCallback sử dụng tham số [dependence] tương tự như useEffect
* Xét ví dụ:
  + index.js:

import { usestate ) from “react”;

import ReactoM from “react-dom/client”;

import Todos from “./Todos”;

const App = () => {

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

const [todos, setTodos] = usestate([1]);

const increment = () => {

setCount ((c) => c + 1);

};

const addTodo = () => {

setTodos ((t) →> [...t, “New Todo”]);

};

return (

<>

<Todos todos={todos) addTodo={addTodo} />

<hr />

<div>

Count: {count)

<button onclick={increment}>+</button>

</div>

</>

);

};

const root = ReactDOM. createRoot (document.getElementById(‘root’));

root.render (<App />);

* + Todos.js

import { memo } from “react”;

const Todos = ({ todos, addTodo }) => {

console.log("child render");

return(

<>

<h2>My Todos</h2>

{todos.map ((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onclick={addTodo}>Add Todo</button>

</>

);}

export default memo (Todos);

* Với ví dụ trên mặc dù có sử dụng React.memo nhưng component Todos vẫn re-render một cách không cần thiết. Lý do bởi vì mỗi lần click nút “addTodo” thì component App sẽ re-render, khi đó hàm addTodo sẽ thuộc một phạm vi khác, khi memo so sánh bằng toán tử “===” thì 2 props không giống nhau nên component Todos sẽ bị re-render
* Cách khác phục: sử dụng useCallback hook:
  + index.js:

import { usestate, useCallback } from “react”;

import ReactDOM from “react-dom/client”;

import Todos from “./Todos”;

const App = () => {

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

const [todos, setTodos] = useState([]);

const increment = () =› {

setCount ((c) => c + 1);

};

const addTodo = useCallback( => t

setTodos ((t) => [...t, "New Todo"]);

},[todos]);

return(

<>

<Todos todos={todos} addTodo={addTodo} />

<hr />

‹div>

Count: {count)

‹button onClick={increment}>+</button>

</div>

</>

);

};

const root = ReactDOM.createRoot (document.getElementById('root')) ;

root.render (<App />);

* + Todos.js:

import { memo } from “react”;

const Todos = ({todos, addTodo }) => {

console.log("child render");

return (

<>

<h2>My Todos</h2>

{todos. map ((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onclick={addTodo}›Add Todo</button>

</>

);

};

export default memo (Todos);

* useCallback sẽ lưu callback ra ngoài một tham chiếu rồi trả về cho addTodo và truyền vào nút Add Todo nên khi memo so sánh props thì không có sự thay đổi nên component không phải re-render chỉ khi dependence todos thay đổi thì component mới phải re-render

1. **useMemo Hook**

* khác với memo, useMemo là một hook. useMemo giúp trách thực hiện lại một logic nào đó không cần thiết
* useMemo và useCallback tương tự nhau, chỉ khác useCallback trả về một hàm calback đã được lưu trữ còn useMemo trả về một giá trị được lưu trữ
* Để sử dụng useMemo, trước tiên phải import vào component:

import {useMemo} from “react”;

* Cú pháp sử dụng useMemo:

UseMemo(callback(),[dependence])

* useMemo sử dụng tham số [dependence] tương tự như useEffect
* Xét ví dụ bên dưới: Ta thấy rằng hàm expensiveCalculation() được gọi mỗi lần component re-render, tức là mỗi lần bấm nút “Add Todo” và “+” trong khi chỉ cẩn gọi tính toán khi biến count thay đổi, sự lặp lại không cần thiết này làm giảm hiệu năng của hệ thống, có thể gây chậm.

import { useState } from "react";

import ReactDOM from "react-dom/client";

const App = () => {

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

const [todos, setTodos] = useState([]);

const calculation = expensiveCalculation(count);

const increment = () => {

setCount((c) => c + 1);

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<div>

<div>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</div>

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

<h2>Expensive Calculation</h2>

{calculation}

</div>

</div>

);

};

const expensiveCalculation = (num) => {

console.log("Calculating...");

for (let i = 0; i < 1000000000; i++) {

num += 1;

}

return num;

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

* useMemo sinh ra để giải quyết vấn đề này

import { useState, useMemo } from "react";

import ReactDOM from "react-dom/client";

const App = () => {

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

const [todos, setTodos] = useState([]);

const calculation = useMemo(() => expensiveCalculation(count), [count]);

const increment = () => {

setCount((c) => c + 1);

useMemo

};

const addTodo = () => {

setTodos((t) => [...t, "New Todo"]);

};

return (

<div>

<div>

<h2>My Todos</h2>

{todos.map((todo, index) => {

return <p key={index}>{todo}</p>;

})}

<button onClick={addTodo}>Add Todo</button>

</div>

<hr />

<div>

Count: {count}

<button onClick={increment}>+</button>

<h2>Expensive Calculation</h2>

{calculation}

</div>

</div>

);

};

const expensiveCalculation = (num) => {

console.log("Calculating...");

for (let i = 0; i < 1000000000; i++) {

num += 1;

}

return num;

};

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<App />);

* Hàm expensiveCalculation() chỉ được gọi để tính toán lại mỗi khi biến count thay đổi (bấm nút “+”), giúp tối ưu hiệu năng, tránh lặp lại những đoạn logic nhiều lần mà không cần thiết

1. **useReducer Hook:**

* Cách hoạt động của useReducer Hook giống với useState Hook, đều có thể thay đổi các logic trạng thái. Nhưng khi dùng nhiều useState để theo dõi nhiều trạng thái hoặc các kiểu dữ liệu phức tạp như mảng hay đối tượng thì useReducer hữu ích hơn.
* Để sử dụng useReducer thì phải import vào component

import { useReducer } from "react";

* Cú pháp sử dụng useReducer

useReducer(<reducer>, <inittalState>)

* Trong đó:
  + <reducer> là hàm thay đổi trạng thái
  + <initialState> là trạng thái bắt đầu
* userReducer trả về trạng thái hiện tại (current state) và một phương thức điều phối (dispatch method)
* Xét ví dụ:

import { useReducer } from "react";

import ReactDOM from "react-dom/client";

const initialTodos = [

{

id: 1,

title: "Todo 1",

complete: false,

},

{

id: 2,

title: "Todo 2",

complete: false,

},

];

const reducer = (state, action) => {

switch (action.type) {

case "COMPLETE":

return state.map((todo) => {

if (todo.id === action.id) {

return { ...todo, complete: !todo.complete };

} else {

return todo;

}

});

default:

return state;

}

};

function Todos() {

const [todos, dispatch] = useReducer(reducer, initialTodos);

const handleComplete = (todo) => {

dispatch({ type: "COMPLETE", id: todo.id });

};

return (

<>

{todos.map((todo) => (

<div key={todo.id}>

<label>

<input

type="checkbox"

checked={todo.complete}

onChange={() => handleComplete(todo)}

/>

{todo.title}

</label>

</div>

))}

</>

);

}

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(<Todos />);

* Ở ví dụ trên, useReducer nhận 2 đối số đó là reducer là hàm thay đổi trạng thái và initialTodos trạng thái đầu là một đối tượng.
* useReducer trả về 2 giá trị current state và dispatch method được gán vào [todos, dispatch]
* Khi truyền type: “COMPLETE” vào dispatch thì sẽ thực hiện khối lệnh của case “COMPLETE” trong switch case của reducer.

1. **React useContext hook**
   1. **React Context:**

* React Context là một cách để quản lý trạng thái toàn cục, có thể truyền dữ liệu từ component cha đến các component con mà không cần sử dngj đến props
* Cách truyền qua props thông thường gây khó trong việc thay đổi (xóa bỏ) component trong source code
* Để sử dụng React Context phải import vào component

import { createContext } from “react”;

* Create Context:

const UserContext = createContext()

* createContext trả về 1 object
* Để component cha truyền đi được dữ liệu thì phải dùng Context Provider
* Ví dụ về sử dụng Provider

function Component1() {

const [user, setUser] = useState("Jesse Hall");

return (

<UserContext.Provider value={user}>

<h1>{`Hello ${user}!`}</h1>

<Component2 user={user} />

</UserContext.Provider>

);

}

* Tất cả các component con của UserContext sẽ nhận được dữ liệu là user
  1. **useContext Hook:**
* Để sử dụng Context trong các component con, cần phải truy cập bằng useContext hook
* Đầu tiên import useContext vào component để sử dụng useContext hook:

import { useContext } from "react";

* + Export đối tượng cần dùng ở component cha

export const UserContext = createContext()

* + Sau đó ở component con, import vào component con

import {UserContext} from ‘./Component1’

* + Bây giờ có thể truy cập UserContext với mọi component con

import {UserContext} from ‘./Component1’

import { usecontext } from “react”;

function Component2() {

const user = useContext(UserContext);

return (

<>

<h1>Component 2</h1>

<h2>(`Hello ${user} again!`}</h2>

</>

);

}

export default Component2

1. **Cài đặt SASS**

* Cài đặt SASS theo cách command line

npm install -g sass

1. **React CSS Styling:**

* Có nhiều cách để CSS cho React, tìm hiểu về 3 cách phổ biến nhất hiện nay:
  + Inline styling
  + CSS styleSheets
  + CSS module

1. **Inline styling:**

* Để tạo kiểu cho một phần tử bằng thuộc tính inline style thì giá trị trong tag style bắt buộc là một đối tượng javaScript.

const Header = () => {

return (

<>

<h1 style={{color: "red"}}>Hello Style!</h1>

<p>Add a little style!</p>

</>

);

}

* Tên các thuộc tính phải dược viết theo quy tắc camelCase.
* Ví dụ: background-color sẽ được viết thành backgroundColor.
* Có thể tạo một đối tượng JavaScript ở bên ngoài và dùng nó để tạo kiểu cho phần tử bằng cách gọi đối tượng đó

const Header = () => {

const myStyle = {

color: "white",

backgroundColor: "DodgerBlue",

padding: "10px",

fontFamily: "Sans-Serif"

};

return (

<>

<h1 style={myStyle}>Hello Style!</h1>

<p>Add a little style!</p>

</>

);

}

1. **CSS styleSheets**

* CSS styleSheets là cách viết CSS trong một file riêng biệt có phần mở rộng .css sử dụng như css thông thường và phải import vào component mới có thể sử dụng được

import './App.css';

* Phía trên là ví dụ cho import file App có phần mở rộng là .css vào component
* Cú pháp được viết như css thông thường

1. **CSS module:**

* CSS module giúp chúng ta tạo ra các file css hoạt động độc lập và không ảnh hưởng tới nhau. CSS trong module chỉ áp dụng cho component đã import nó, sẽ không bị xung đột về tên class.
* Cách dùng CSS module:
  + Tạo file có phần mở rộng là .module.css (vd: test.module.css với class bigblue)

.bigblue {

color: DodgerBlue;

padding: 40px;

font-family: Sans-Serif;

text-align: center;

}

* + Import vào component

import styles from './my-style.module.css';

const Car = () => {

return <h1 className={styles.bigblue}>Hello Car!</h1>;

}

export default Car;

* + Sau đó gọi bigblue thông qua đối tượng style đã import

1. **React Sass Styling:**

* Sass là ngôn ngữ tiền xử lý
* Các file Sass được thực thi trên sever và gửi CSS tới trình duyệt
* Để sử dụng Sass, cài đặt như phần I
* Đầu tiên, Sass được viết trong file có phần mở rộng .sass.
* Cách import file Sass tương tự như import CSS

import ‘.my-sass.scss’;

* Trong file Sass có thể sử dụng các biến hoặc hàm Sass

$myColor: red;

h1 {

color: $myColor;

}

* Khi file Sass biên dịch, nó sẽ lấy biến myColor và xuất ra file css thông thường với các giá trị của biến trong CSS.
* Sass cho phép lồng các CSS selectors
* Ví dụ:

nav{

ul{

margin: 0;

padding: 0;

list-style: none;

}

li{

display: inline-block;

}

a{

display: block;

padding: 6px 12px;

text-decoration: none;

}

}

# KẾT LUẬN

Qua quá trình tự nghiên cứu đã đuc kết được cho bản thân nhiều kết quả quan trọng. Biết được ReactJs là gì và công dụng của ReactJs, nắm được các khái niệm cơ bản của ReactJs như component, props, state, nắm được cách sử dụng các Hook cơ bản như useState, useEffect, useLayoutEffect, useRef, useCallback, useMemo, useReducer, useContext,… giúp ích rất nhiều trong quá trình xây dựng giao diện người dùng.

Tuy nhiên, trong quá trình nghiên cứu, em cũng nhận ra một số tồn tại và hạn chế. Như là làm sao để tối ưu hóa hiệu suất cho ứng dụng, giảm thiểu số lần re-render không cần thiết, tối ưu tài nguyên và trải nghiệm người dùng.

# PHỤ LỤC

* Sản phẩm minh họa: <https://github.com/Duong247/userMangerment>
* Ứng dụng web mô phỏng lại ứng dụng quản lý tài khoản người dùng được lấy từ API (<https://reqres.in/>) với các chức năng như thêm tài khoản người dùng, xóa tài khoản người dùng, sửa thông tin tài khoản,…
* Công nghệ sử dụng: ReactJs, HTML/CSS, bootstrap5
* Hình ảnh minh họa sản phẩm:

