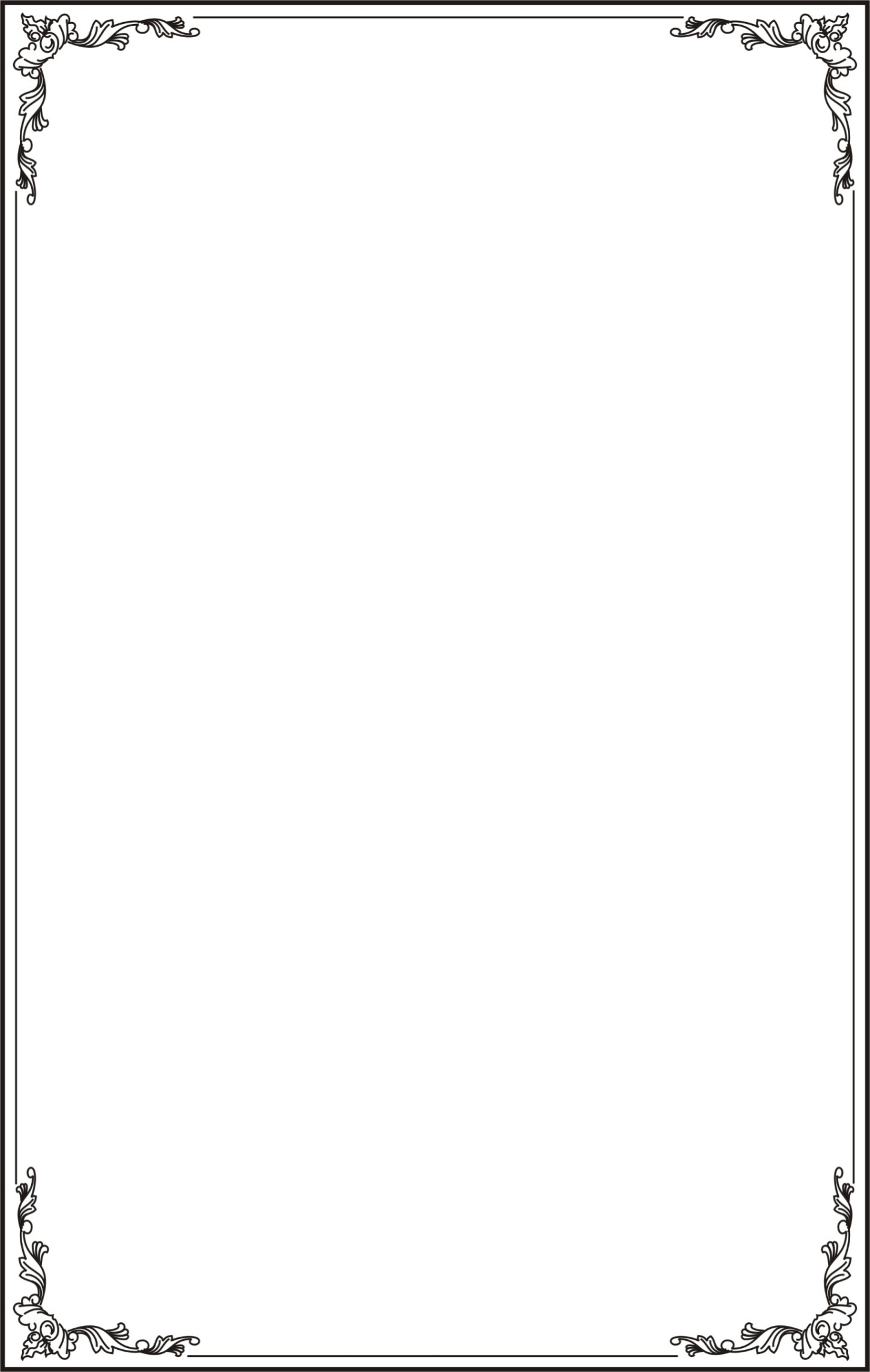
****

**ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH**

**ĐẠI HỌC KHOA HỌC TỰ NHIÊN TPHCM**

**KHOA: CNTT – VIỆT PHÁP**



**BÁO CÁO QUÁ TRÌNH THỰC TẬP TẠI FPT TELECOM**

|  |  |
| --- | --- |
| Người hướng dẫn: | **TRƯƠNG TẤN SANG** |
| Người thực hiện: | **TRẦN KHÁNH GIA UY** |
| Đơn vị công tác: | **Trung tâm giám sát và đảm bảo dịch vụ (SCC)** |
| Vị trí: | **Open Source Web Software Developer** |
| Bộ phận: | **SCC** |

***Thành phố Hồ Chí Minh, tháng 09 năm 2021***

**CHƯƠNG 1: TÌM HIỂU VỀ NHỮNG THAY ĐỔI TỪ ES6 – ES11**

**2.1 ES6**

**2.1.1 Arrow function**

* Trước đây ta thường khai báo hàm dạng như sau:

return function func() {

    //code

}

* Còn bây giờ ta sử dụng => để chỉ ra function :

return () => {

    //code

}

* Việc sử dụng arrow function giúp tiết kiệm thời gian code hơn

**2.1.2 Class**

* ES6 cung cấp các lớp cho người dùng sử dụng:

class Student {

    constructor(id, name) {

        this.id = id;

        this.name = name;

    }

    func() {

        //code

    }

}

var objStudent = new Student(1, "Uy");

objStudent.func();

**2.1.3 Let and const**

* Tầm vực của let là block scope chứ không phải globally hay locally scoped. Chính vì thế biến let chỉ có hiệu lực trong block code. Ngoài ra biến let không thể tái khai báo
* Const là biến không thể tái khai báo hoặc cập nhật giá trị của biến. Tuy nhiên các thuộc tính của biến vẫn có thể cập nhật được giá trị

**2.1.4 Template String**

* Trước đây, khi muốn tạo các string từ các biến ta sử dụng dấu +. Ví dụ

var a = 5;

var str = "a is " + a;

* Bây giờ việc tạo string từ các biến chỉ cần đặt vào trong ${}

var t = 10;

var temp = `t: ${t} `;

**2.1.5 Promise**

* Promise là một cơ chế trong JavaScript giúp thực thi các tác vụ bất đồng bộ mà không rơi vào callback hell hay pyramid of doom, là tình trạng các hàm callback lồng vào nhau ở quá nhiều tầng. Ví dụ:

function readFilePromise(path) {

    return new Promise(function(resolve, reject) {

        fs.readFile(path, function(err, data) {

            if(err) {

                reject(err);

            } else {

                resolve(data);

            }

        });

    });

}

readFilePromise('file.txt')

    .then(function(file) {

        console.log(file);

    })

    .catch(function(err) {

        console.log(err);

    })

* Khi .then(…) có return, thì giá trị trả về sẽ được bọc bên trong 1 promise. Cho phép kết nối nhiều promise với nhau. Ví dụ:

promise()

  .then(() => {

    return 'foo'

  })

  .then(result1 => {

    console.log(result1) // 'foo'

    return anotherPromise()

  })

  .then(result2 => console.log(result2)) // `result2` sẽ là kết quả của anotherPromise()

  .catch(err => {})

**2.2 ES7**

**2.2.1 Array.prototypes.include**

* Được sử dụng để kiểm tra phẩn tử có nằm trong array hay không

var arr = [1, 2, 3]

console.log(arr.includes(1)); //true

console.log(arr.includes(0)); //false

**2.2.2 Exponential operator**

* Là toán tử mũ:

var a = Math.pow(3, 2)

if(3\*\*2 === a) {

    console.log(true);

}

else {

    console.log(false);

}

**2.3 ES8**

**2.3.1 Object.values() và Object.entries()**

* Ta có object sau :

var obj = {

    firstName: "Uy",

    lastName: "Tran"

}

* Với hàm Object.entries(), nó sẽ trả về mảng các cặp key – value của object:
* Tương tự Object.entries(), Object.values() trả về mảng các value của object

**2.3.2 String.prototype.padStart() và String.prototype.padEnd()**

* Hai hàm này được sử dụng để thêm chuỗi đệm vào đầu hoặc vào cuối của một string
* padStart:

const str1 = '5';

console.log(str1.padStart(2, '0'));

// expected output: "05"

* padEnd:

const str1 = '5';

console.log(str1.padEnd(2, '0'));

// expected output: "50"

**2.3.3 Async, await**

* Async: Khai báo hàm bất đồng bộ
  + Tự động biến đổi một hàm thông thường thành một Promise
  + Khi gọi tới hàm async nó sẽ xử lý mọi thứ và trả về kết quả trong hàm của nó
  + Async cho phép sử dụng await
* Await: tạm dừng tất cả các hàm async
  + Khi đặt trước Promise, nó sẽ đợi Promise kết thúc và trả về kết quả
  + Await chỉ làm việc với Promise, nó không hoạt động với callback
  + Await chỉ có thể được sử dụng bên trong các function async

**2.4 ES9**

**2.4.1 Async iteration**

* Async iteration dùng để sử dụng vòng lặp với async, await
* Cú pháp:

for await (let item of listItem) {

    //code

}

**2.4.2 Promise.prototype.finally**

* Đây là bước cuối cùng thực hiện trong Promise dù nó được resolve hay reject. Hàm này trả về một Promise. Ví dụ :

let isLoading = true;

fetch(myRequest).then(function(response) {

    var contentType = response.headers.get("content-type");

    if(contentType && contentType.includes("application/json")) {

      return response.json();

    }

    throw new TypeError("Oops, we haven't got JSON!");

  })

  .then(function(json) { /\* process your JSON further \*/ })

  .catch(function(error) { console.log(error); })

  .finally(function() { isLoading = false; });

**2.4.3 Rest operator**

* Toán tử “…” được sử dụng để chọn tất cả các phần tử còn lại của mảng. Ví dụ:

const fruits = { orange: 1, melon: 2, apple: 10, banana: 4 }

const { orange, ...r } = fruits;

console.log(r); // {melon: 2, apple: 10, banana: 4 };

**2.5 ES10**

**2.5.1 Optional catch binding**

* Sử dụng try catch mà không có tham số truyền vào. Ví dụ:

try {

    //code

} catch {

    // code

}

**2.5.2 Object.fromEntries()**

* Ngược lại với Object.entries(). Object.fromEntries() trả về một Object từ mảng các cặp key – value

const entries = [

    ['foo', 'bar'],

    ['baz', 42]

];

const obj = Object.fromEntries(entries);

console.log(obj);

// expected output: Object { foo: "bar", baz: 42 }

**2.5.3 Array.flat()**

* Hàm sử dụng để flat các mảng lồng nhau. Ví dụ:

var arr = [1, 2, [3, 4, [5, 6]]];

arr.flat();

// [1, 2, 3, 4, [5, 6]]

* Nếu truyền vô tham số cho hàm này, nó sẽ là số mảng lồng nhau mà mình muốn flat. Ví dụ :

var arr1 = [1, 2, [[3, 4, [5, 6]]]];

arr1.flat(2);

// [1, 2, [3, 4, 5, 6]]

var arr2 = [1, 2, [3, 4, [5, 6, [7, 8, [9, 10]]]]];

arr2.flat(Infinity);

// [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

**2.5.4 Array.flatMap()**

* Đây là hàm flat mảng sau khi map.

let arr = ["it's Sunny in", "", "California"];

arr.map(x => x.split(" "));

// [["it's","Sunny","in"],[""],["California"]]

arr.flatMap(x => x.split(" "));

// ["it's","Sunny","in", "", "California"]

**2.5.5 String.trimStart() và String.trimEnd()**

* Xóa các khoảng trắng trước hoặc sau chuỗi. Ví dụ:

var greeting = '   Hello world!   ';

console.log(greeting.trimStart());

// expected output: "Hello world!   ";

console.log(greeting.trimEnd());

// expected output: "   Hello world!";

**2.5.6 Dynamic import**

* Ta có static import như sau :

export function getMessage() {

    return "Hello world";

}

import { getMessage } from './mymodule.js';

let message = getMessage();  // output  "Hello world";

* Với dynamic import, nó sẽ trả về một Promise của module request :

import(moduleSpecifier)

**2.5.7 globalThis object**

* globalThis xuất hiện nhằm mục đích củng cố các cách ngày càng bị phân mảnh để truy cập đối tượng toàn cầu bằng cách xác định một thuộc tính toàn cầu tiêu chuẩn.

**2.6 ES11**

**2.6.1 Optional chaining ?.**

* Cho phép đọc giá trị nằm sâu bên trong object mà không cần kiểm tra xem có valid hay không
* Ví dụ:

const adventurer = {

    name: 'Alice',

    cat: {

        name: 'Dinah'

    }

};

const dogName = adventurer.dog?.name;

console.log(dogName);

// expected output: undefined

**2.6.2 Private field #**

* Bây giờ ta có thể khai báo các thuộc tính private trong class bằng cách thêm dấu #. Ví dụ:

class Foo {

    #b = 15;

    a = 10;

    get() {

      return this.#b;

    }

}

const obj = new Foo();

obj['#b']; // undefined

**2.6.3 Undefined JS, Nullish Coalescing ??**

* Với toán tử ?? ta có đoạn code sau :

a() ?? b()

* Nếu a() là null hoặc undefined, toán tử sẽ trả về vế phải. Để đề phòng vế trái là falsy ta có thể sử dụng a() || b()

**2.6.4 Top level await**

* Cho phép sử dụng từ khóa await bên ngoài async function. Nó hoạt động giống như một hàm async lớn khiến các module khác phải đợi trước khi thực hiện các lệnh bên trong

await Promise.resolve(console.log('temp'));

// → temp

**CHƯƠNG 2: TÌM HIỂU VỀ SPA, VIRTUAL DOM VÀ REALDOM**

**3.1 Single Page Application (SPA)**

**Table, timeline

Description automatically generated3.1.1 Concept**

* SPA là một ứng dụng web sử dụng HTML5 và AJAX giúp nâng cao trải nghiệm người dùng. Khi tải một trang web, SPA sẽ load toàn bộ resource của trang bao gồm HTML, CSS, JS và cấu trúc webpage của khung trang web. Sau đó client sẽ gửi những request AJAX để get những dữ liệu cần thiết (thường là nội dung của trang).
* Ví dụ:

A person sitting on a bench

Description automatically generated with medium confidence

* Vùng màu vàng được coi là khung hay là master của một trang
* Vùng màu đỏ là nội dung trang.

**3.1.2 So sánh giữa SPA và multi-page application**

* Với multi-page application, mỗi khi người dùng gửi yêu cầu, toàn bộ trang web sẽ được tải lại hoàn toàn. Do đó nếu có nhiều yêu cầu thì việc trả về một trang web sẽ diễn ra lâu hơn do cần tính toán nhiều hơn.
* Với SPA thì chỉ có nội dung cần thay đổi thì mới cần tải lại thôi, nên sẽ không có hiện tượng load lại toàn trang web và nó giúp tiết kiệm thời gian chờ đợi.

**3.1.3 Ưu điểm của SPA**

* Như đã đề cập ở trên, SPA giảm rất nhiều thời gian chờ đợi. Bên cạnh đó, nó giúp cho trải nghiệm moble tốt hơn, tạo cảm giác liền mạch.
* Ngoài ra, SPA có độ tin cậy cao hơn do mọi link trong SPA đều trỏ về trang chủ

**3.1.4 Nhược điểm của SPA**

* Trang web sẽ bị giới hạn nội dung hiển thị và không hỗ trợ tạo format bài viết dài (Facebook).
* Có những kỹ thuật SEO nâng cao không sử dụng được trên SPA. Ví dụ như kỹ thuật cấu trúc website thành cái Category và Sub-Category để hiển thị nội dung tốt nhất cho người dùng.

**3.2 Virtual DOM và Real DOM**

**3.2.1 Virtual DOM**

* Virtual DOM sẽ copy toàn bộ Real DOM và lưu lại trên ram. Nếu như có sự thay đổi ở một node nào đó trên cây DOM thì Virtual DOM sẽ update lại node đó và render lại trên DOM thật.
* Ví dụ: ta có một DOM như sau :

**Diagram

Description automatically generated**

* Khi đó Virtual DOM sẽ sao chép toàn bộ DOM này. Giả sử node 1 bị thay đổi state (kí hiệu màu đỏ) thì Virtual DOM sẽ dò lại từng node để so sánh và đối chiếu những sự thay đổi, rồi cập nhật lại node bị thay đổi và bỏ qua những node không liên quan.
* Cuối cùng nó sẽ render lại trên DOM thật để thay đổi dữ liệu cần thiết

**3.2.2 So sánh giữa Virtual DOM và Real DOM**

* Virtual DOM chỉ render lại những node bị thay đổi chứ không phải toàn bộ DOM như real DOM.

**CHƯƠNG 3: TÌM HIỂU VỀ SEVER SIDE RENDERING & CLIENT SIDE RENDERING, NEXTJS, GASBY VÀ GENERATOR FUNCTION**

**4.1 Server side rendering & client side rendering**

**4.1.1 Server side rendering**

* Đây là cơ chế được sử dụng từ lâu. Các xử lý về logic sẽ được thực hiện ở phía server và client chỉ nhận các kết quả trả về từ phía server này.
* Cụ thể hơn như sau:
  + Khi người dùng truy cập vào một trang web. Ngay lập tức trình duyệt sẽ gửi GET request đến server.
  + Server sau khi nhận request sẽ xử lý các yêu cầu và lấy dữ liệu từ database để xử lý.
  + Cuối cùng server sẽ render HTML rồi trả về cho trình duyệt để hiển thị.

Diagram

Description automatically generated

* Ưu điểm của server side rendering:
  + Việc khởi tạo các thành phần khá nhanh và dễ optimize vì toàn bộ dữ liệu đã được xử lý ở server. Client chỉ làm nhiệm vụ hiển thị.
  + Đa số cái web framework đều hỗ trợ cơ chế này
  + Dễ hiểu và dễ code hơn do không cần phải chia ra frontend và backend
  + Chạy được trên phần lớn các trình duyệt. Kể cả disable JS vẫn chạy được.
* Nhược điểm của server side rendering:
  + Do mỗi lần chuyển trang thì client lại phải gửi request cho server trả về HTML nên phải mất rất nhiều thời gian tải lại mỗi lần chuyển trang. Do đó, nó cũng gây khó chịu cho người dùng.
  + Nặng cho phía server vì phải gánh nhiều thành phần xử lý logic và dữ liệu.
  + Tốn băng thông do server gửi nhiều thành phần trùng lặp (khung trang, menu, header, footer, …).

**4.1.2 Client side rendering**

* Đây là cơ chế được sử dụng trong Single Page Application (SPA). Ngược lại với server side rendering, client side rendering sẽ xử lý một số thành phần đơn giản (render html, css, js)
* Cụ thể hơn như sau:
  + Những logic đơn giản (validation, đọc dữ liệu, sorting, filtering) nằm ở phía client
  + Logic để chuyển trang (routing), render dữ liệu sẽ nằm ở phía client, trừ một số routing đặc biệt do server quản lý
  + Một số logic phức tạp (thanh toán, phân quyền) vẫn được xử lý ở phía server

Diagram

Description automatically generated

* Ưu điểm của client side rendering
  + Page chỉ cần load 1 lần duy nhất. Khi user chuyển sang trang khác hoặc thêm dữ liệu, client sẽ gửi request và lấy dữ liệu từ phía server qua AJAX. Từ đó, client có dữ liệu mới mà không cần chuyển trang
  + Do chuyển một số logic cho client nên phía server sẽ đỡ nặng hơn
  + Giảm được băng thông do client chỉ cần một phần dữ liệu, không cần toàn bộ dữ liệu của cả trang
  + Với ứng dụng cần nhiều tương tác, single page application hoạt động mượt mà hơn trên browser mà không cần load đi load lại nhiều
* Nhược điểm:
  + Việc khởi tạo trang mất nhiều thời gian hơn vì browser phải tải toàn bộ JS về, gọi API để lấy dữ liệu về server rồi mới render dữ liệu
  + Đòi hỏi cấu trúc của phần mềm phải có 2 phần riêng biệt là Front end và Back end nên việc code sẽ phức tạp hơn
  + Không chạy được nếu như JS bị disable hoặc ở các trình duyệt cũ không nhận được JS ES6
  + Nếu phía client sử dụng mobile có cấu hình yếu thì sẽ bị chậm khi load.

**4.2 Nextjs**

**4.2.1 Dẫn nhập**

* Cơ chế Client side rendering được sử dụng chính trong React, chính vì thế với mỗi trang đơn giản (chẳng hạn như About) thì client phải tải một lượng lớn JS về. Chính vì sự bất tiện này Server side rendering dành cho React ra đời và điều này làm tăng hiệu suất hơn cho các điện thoại với cấu hình yếu

**4.2.2 Giới thiệu**

* Trong Next.js có những giải pháp sau:
  + Tốt cho việc SEO website
  + Tích hợp bộ nhớ đệm và tối ưu hóa tĩnh tự động
  + Các trang hoàn toàn do máy chủ hiển thị
  + 100% React hỗ trợ
  + Tinh chỉnh cấu hình webpack / babel của bạn nếu cần

**4.2.3 Các tính năng**

* Tạo các nội dung tĩnh bằng cách động: khi Next.js tìm nạp URL cụ thể, nó sẽ lưu nó dưới dạng một trang tĩnh và phân phát nó một cách tĩnh bất cứ khi nào ai đó truy cập đường dẫn. Đồng thời, nó sẽ sẵn sàng chấp nhận các paths mới một cách linh hoạt.
* Webpack 5 support
* Trong React thông thường sẽ có một hàm duy nhất thực thi cả 2 môi trường server và client, đó là getInitialProps. Tuy nhiên Next.js có giải pháp là đưa ra 2 phương thức là getServerSideProps (đưa các props vào trang từ server) và getStaticProps (tạo ra các đầu ra tĩnh tại thời điểm build).
* Hỗ trợ bộ nhớ đệm liên tục cho các trang không bị thay đổi
* Hỗ trợ cho SCSS

**4.3 Gasbyjs**

**4.3.1 Giới thiệu**

* Gastby JS là nền tảng dùng để xây dựng website và web app tĩnh để hoạt động ở hiệu suất rất cao. Với các web động, việc tạo cache ở thời điểm runtime vì thế server phải chạy liên tục để có đầy đủ cơ sở dữ liệu. Còn với Gasby.js thì khi web build xong sẽ ra tập tin tĩnh thi máy chủ có thể tắt đi giúp tiết kiệm chi phí
* Với Gasby.js khi nhấn vào liên kết để chuyển trang thì chỉ cần tải tập tin JSON của nội dung mà trang khác khác cần hiển thị.

**4.3.2 Ưu điểm**

* Triển khai với chi phí thấp và dễ dàng
* Tốc độ cao
* Tối ưu SEO
* Hỗ trợ Progressive Web App
* An toàn

**4.3.3 Nhược điểm**

* Với việc triển khai hệ thống bình luận, tìm kiếm phải dựa vào các dịch vụ bên thứ 3
* Thời gian build khá lâu
* Triển khai cho người dùng bình thường còn phức tạp. Thật khó để sử dụng chức năng hẹn giờ xuất bản, hoặc chỉ là những cập nhật nho nhỏ.
* Với wordpress, hay 1 nguồn nội dung khác, thì phần “hình ảnh trong bài viết” có thể không tận dụng được khả năng xử lý hình ảnh của Gatsby Js.
* Mất đi một số tính năng hay của các nền tảng khác cung cấp. Ví dụ với WordPress mình kết hợp với OneSignal, khi mỗi bài viết được xuất bản thì sẽ tự động gởi thông báo đến tất cả các người dùng đang theo dõi (subscribers).

**4.4 Generator function**

**4.4.1 Giới thiệu**

* Generator function là một hàm có thể được thực thi nhiều lần liên tiếp mà các số lượng biến, giá trị biến, trạng thái các thành phần bên trong hàm, … đều có thể lưu lại sử dụng sau mỗi phiên. Generator function có thể dừng lại ở bất kì khi nào đợi một điều kiện nào đó xảy ra rồi mới thực thi.

**4.4.2 Cú pháp**

function\* name([param[, param[, ...param]]]) {

  //code

}

* Trong đó: name là tên hàm, param : các tham số truyền vào (tối đa 255 tham số)

**4.4.3 Giá trị trả về**

* Với generator function, nó không trả về kiểu dữ liệu cơ bản mà thay vào đó, nó trả về một iterator object. Hàm next() của iterator object được sử dụng để truy xuất các node dữ liệu sau mỗi bước resume lại generator function. Khi đó generator function sẽ thực thi hàm cho đến khi gặp từ khóa yield, hoặc return kế tiếp chưa được duyệt ở bước trước.

**4.4.4 Yield**

* Yeild là từ khóa dùng để tạm dừng và cũng để tiếp tục việc thực thi bên trong generator function
* Ví dụ:

function\* generatorFunc(index) {

  while (index < 2) {

    yield index++;

  }

}

const iterator = generatorFunc(0);

console.log(iterator.next());

// log output: {value : 0, done : false}

console.log(iterator.next());

// log output: {value : 1, done : false}

console.log(iterator.next());

// log output: {value : underfined, done : true}

**4.4.5 Yield\***

* Yield\* là một dạng ủy quyền thực thi. Nó có thể nhúng mã của một generator function ngay sau nó hoặc là ủy quyền trực tiếp cho một iterator object. Ngoài ra, yield\* là một biểu thức chứ không phải một câu lệnh
* Cú pháp:

yield\* [[expression]]

expression ở đây luôn là dạng iterator object nghĩa là nó có thể là mảng, string, hay là generator function.

* Ví dụ:

function\* g2() {

  yield\* [1, 2];

  return 'abc';

}

var rs;

function\* g() {

   rs = yield\* g2();

}

var iterator = g();

console.log(iterator.next());

// {value: 1, done: false}

console.log(iterator.next());

// {value: 2, done: false}

console.log(iterator.next());

// {value: undefined, done: true},

// g2() đã trả về {value: 'abc', done: true}

**CHƯƠNG 4: TÌM HIỂU VỀ LIFECYCLE, HOOK**

**5.1 Vòng đời của component**

* Trong React, có 3 giai đoạn (phase) sẽ được thực hiện khi chạy chương trình:
  + Render phase: đây là giai đoạn bao gồm các công việc render components và tính toán các thay đổi cần apply. Giai đoạn này có thể tạm ngưng, hủy hoặc chạy lại bởi React
  + Pre-commit phase: Giai đoạn này sẽ đọc DOM
  + Commit phase: Giai đoạn áp dụng các thay đổi vào DOM thật.
* Trong 3 phase này sẽ có các nhóm phương thức như là *Mounting, Updating, Unmounting* và *Error handling*

Diagram

Description automatically generated with medium confidence

**5.1.1 Mounting**

**5.1.1.1 Constructor()**

* Phương thức constructor() luôn luôn được gọi đầu tiên khi component được khởi tạo và nó là nơi để cài đặt khởi tạo state và các giá trị khác.

constructor(props) {

  super(props);

  this.state = {favoritecolor: "red"};

}

**5.1.1.2 getDerivedStateFromProps()**

* Phương thức này được cọi ngay trước khi render các elements trong DOM. Có thể return một object để update state hoặc null
* Rất hiếm khi sử dụng, chỉ khi giá trị của state phụ thuộc vào prop

static getDerivedStateFromProps(props, state) {

  return {favoritecolor: props.favcol };

}

**5.1.1.3 render()**

* Đây là phương thức bắt buộc duy nhất khi tạo ra một component, trả về một trong những giá trị sau:
  + React element
  + Arrays và fragment
  + Portals
  + String và numbers
  + Boolean hoặc null

render() {

  return (

    <h1>This is the content of the Header component</h1>

  );

}

**5.1.1.4 componentDidMount()**

* Đây là phương thức được gọi sau khi render. Được thực hiện để kết nối React với một ứng dụng bên ngoài ví dụ như web API. Ngoài ra, đây cũng là nơi thích hợp để thực hiện các hàm setInterval hoặc setTimeout

componentDidMount() {

    setTimeout(() => {

      this.setState({favoritecolor: "yellow"})

    }, 1000)

  }

**5.1.2 Updating**

**5.1.2.1 getDerivedStateFromProps()**

* Phương thức này sẽ được gọi xuống trước khi render, có thể return được một object để update state hoặc null
* Rất hiếm khi sử dụng, chỉ khi giá trị của state phụ thuộc vào prop
* Nên hạn chế sử dụng method này vì logic hiển thị khá khó hiểu.

class Header extends React.Component {

  constructor(props) {

    super(props);

    this.state = {favoritecolor: "red"};

  }

  static getDerivedStateFromProps(props, state) {

    return {favoritecolor: props.favcol };

  }

  render() {

    return (

      <h1>My Favorite Color is {this.state.favoritecolor}</h1>

    );

  }

}

ReactDOM.render(<Header favcol="yellow"/>, document.getElementById('root'));

**5.1.2.2 shouldComponentUpdate()**

* Đây là phương thức sẽ xác định rằng component có được update hay không. Mặc định sẽ trả về là true. Có thể re-render lại component bằng cách thêm một số ràng buộc điều kiện trong method này.
* Giả sử muốn re-render component khi props thay đổi thì method sẽ nhận nextProps và nextState để giúp chúng ta so sánh với prop hiện tại

shouldComponentUpdate(nextProps, nextState) {

  return this.props.clicks !== nextProps.clicks;

}

**5.1.2.3 componentWillUpdate()**

* Phương thức này sẽ được gọi giữa shouldComponentUpdate() và render(). Nó nhận 2 tham số truyền vào là nextProps và nextState.
* Phương thức này không thể gọi được **this.setState** bởi vì mục tiêu của method này là tương tác với những thứ bên ngoài React

componentWillUpdate(nextProps, nextState) {

  if (nextState.open == true && this.state.open == false) {

    this.props.onWillOpen();

  }

**5.1.2.4 getSnapshotBeforeUpdate()**

* Gọi ngay trước khi render xuống DOM, cho phép lấy một số thông tin của DOM, các giá trị return từ hàm này sẽ đưa cho **componentDidUpdate()**

getSnapshotBeforeUpdate(prevProps, prevState) {

  document.getElementById("div1").innerHTML =

  "Before the update, the favorite was" + prevState.favoritecolor;

}

componentDidUpdate() {

  document.getElementById("div2").innerHTML =

  "The updated favorite is " + this.state.favoritecolor;

}

**5.1.2.5 componentDidUpdate()**

* Phương thức này sẽ chỉ được gọi sau khi re-render component. Sau mỗi lần update mới, component sẽ được update vào Dom và phương thức này sẽ được thực hiện. Method này sẽ nhận vào các đối số như prevProps và prevState.

componentDidUpdate() {

  document.getElementById("mydiv").innerHTML =

  "The updated favorite is " + this.state.favoritecolor;

}

**5.1.2.6 componentWillReceiveProps**

* Phương thức này sẽ được gọi trước khi một component nhận props mới.
* Nếu component cha làm các components con phải re-render lại thì phương thức này sẽ được gọi kể cả khi props không thay đổi

componentWillReceiveProps(nextProps){

  if(nextProps.someValue!==this.props.someValue){

    //Perform some operation

    this.setState({someState: someValue });

    // ....

  }

}

**5.1.3 Unmounting**

**5.1.3.1 componentWillUnmount**

* Có thể sử dụng để remove các listener, các hàm setInterval, cancel network request và nó sẽ được gọi trước khi unmount component

componentWillUnmount() {

  document.removeEventListener("click", this.closeMenu);

}

**5.1.4 Error handling**

**5.1.4.1 componentDidCatch()**

* Nếu một component nào đó bị lỗi nó sẽ không chết nguyên cái app nữa mà sẽ bắt lỗi ở đây

componentDidCatch(error, errorInfo) {

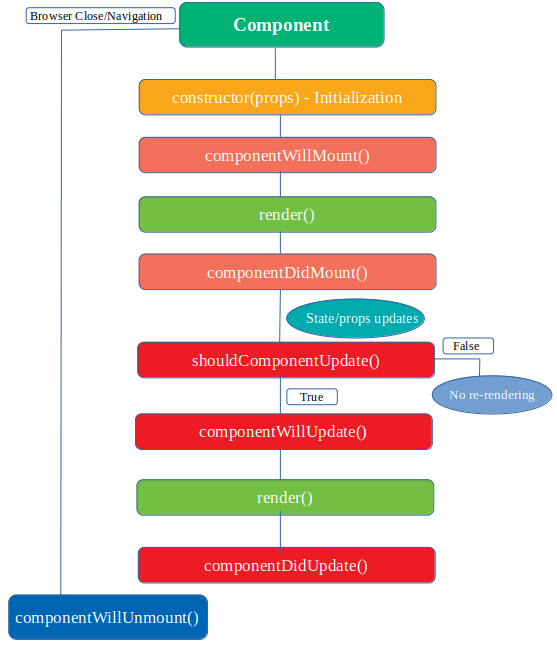
  // You can also log the error to an error reporting service

  logErrorToMyService(error, errorInfo);

}

**5.1.5 Tổng kết về lifecycle component**

* Đây là sơ đồ thứ tự thực hiện của các method trong một lifecycle



**5.2 Hook**

Với mục đích sử dụng state và các chức năng của một class component mà không cần phải tạo class nên hook đã ra đời

**5.2.1 useState**

* Giúp sử dụng state trong functional componnet
* Truyền vào là initialState (giá trị hoặc function)
* Trả về một mảng có 2 phần tương ứng là state và setState. **state:** định nghĩa tên của state nó có thể là đơn giá trị hoặc object,.. (là tham số của useState), **setState**: định nghĩa tên function dùng cho việc update state (là tham số của useState)
* Ví dụ:

const [count, setCount] = useState(0)

const handleClick = () => setCount(age + 1)

**5.2.2 useEffect**

* **useEffect** cho phép chúng ta sử lý logic trong lifecycle methods.
* Đây là sự kết hợp của componentDidMount, componentDidUpdate và componentWillMount
* Nếu muốn dùng như componentDidUpdate thì ta dùng như sau: useEffect(function)

const [count, setCount] = useState(0)

const handleClick = () => setCount(count + 1)

useEffect(() => {

  document.title = 'Count is: ' + count

})

* Còn nếu muốn dùng như componentDidMount thì ta truyền vô thêm một mảng rỗng :

const [count, setCount] = useState(0)

const handleClick = () => setCount(count + 1)

useEffect(() => {

  document.title = 'Count is: ' + count

}, [])

* Còn với componentWillUnmount thì ta sẽ trả về một function và function này sẽ được gọi ngay trước khi component unmount

useEffect(() => {

  const clickWindow = () => console.log('1')

  window.addEventListener('click', clickWindow)

  //return 1 function, sẽ được gọi ngay trước khi component unmount

  return () => {

    window.removeEventListener('click', clicked)

  }

}, [])

**5.2.3 useContext**

* **useContext** là một hooks trong React Hooks cho phép chúng ta có thể làm việc với React Context trong một functional component.

const value = useContext(AppContext);

**5.2.4 useLayoutEffect**

* Method này khác với useEffect ở chỗ là thời điểm gọi (không hiểu lắm)

useLayoutEffect(() => {

  //Do something and either return undefined or a cleanup function

  return () => {

    //Do some cleanup here

  };

}, [dependencies]);

**5.2.5 useReducer**

* Đây là một phiên bản nâng cao của useState, dùng trong trường hợp local state của component phức tạp, có nhiều action làm thay đổi state đó.
* Hàm có 2 tham số là state, action và trả về new state sau khi thực hiện một action với cú pháp như sau (state, action) => newState

**5.2.6 useCallback**

* useCallback có chức năng tránh re-render ở component con, ta xét ví dụ sau:

function Parent({ ... }) {

  const [a, setA] = useState(0);

  const onChangeHandler = useCallback(() => {

    doSomething(a);

  }, [a]);

  ...

  return (

    ...

    //Pure là component con có sử dụng React.memo

    <Pure onChange={onChangeHandler} />

  );

}

* Mỗi lần component **Parent** re-render thì hàm **onChangeHandler** sẽ không cần phải tạo mới nữa mà nó chỉ tạo mới khi biến **a** thay đổi.

**5.2.7 useMemo**

* Tương tự như useCallback, useMemo cũng có chức năng tránh re-render, ngoài ra cũng tránh được những tính toán nặng. Ta xét ví dụ sau:

const componentA = () => {

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

  const getArray = useMemo(() => {

    // tưởng tượng một hàm phức tạp, filter,

    // sort một mảng 100 phần tử, tốn 2s để chạy

    const result = filterAndSortAndDoSomething(...);

    return result;

  }, []);

  return (

    <div>

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

      <div>mảng phức tạp: {getArray}</div>

    </div>

  )

}

* Khi bấm nút và tăng biến **count** thì hàm **getArray** sẽ trả ra giá trị **result** trước đó mà không cần mất 2 giây để tính toán

**2.5.8 useRef**

* Công dụng của useRef là cố định dữ liệu giữa các lần re-render, truy xuất giá trị đó qua thuộc tính current
* Một ứng dụng rất phổ biến của useRef là truy xuất đến DOM node. Thí dụ để set focus của input

function Form() {

  const nameRef = React.useRef();

  const emailRef = React.useRef();

  const passwordRef = React.useRef();

  const handleSubmit = (e) => {

    e.preventDefault();

    const name = nameRef.current.value;

    const email = emailRef.current.value;

    const password = passwordRef.current.value;

    console.log(name, email, password);

  };

  return (

    <React.Fragment>

      <label>

        Name:

        <input placeholder="name" type="text" ref={nameRef} />

      </label>

      <label>

        Email:

        <input placeholder="email" type="text" ref={emailRef} />

      </label>

      <label>

        Password:

        <input placeholder="password" type="text" ref={passwordRef} />

      </label>

      <button onClick={() => nameRef.current.focus()}>Focus Name Input</button>

      <button onClick={() => emailRef.current.focus()}>

        Focus Email Input

      </button>

      <button onClick={() => passwordRef.current.focus()}>

        Focus Password Input

      </button>

      <button onClick={handleSubmit}>Submit</button>

    </React.Fragment>

  );

}

**CHƯƠNG 5 : DEMO REACT APPLICATION**

**6.1 Ý tưởng**

Phần này em sẽ xây dựng ứng dụng todo dùng để ghi lại những công việc cần làm. Có các chức năng cơ bản như :

* Thêm công việc
* Xóa công việc
* Sửa công việc
* Đánh dấu đã làm
* Lọc các công việc chưa làm, đã làm
* Xóa các công việc đã làm

Em sẽ sử dụng Class component và lifecycle component để quản lý các sự kiện trong app. Ngoài ra, em sử dụng state là nơi để em lưu dữ liệu hiện ra màn hình, local storage để lưu tất cả dữ liệu. Như vậy em sẽ có các component sau :

* TodoItem : là component thể hiện những công việc
* TodoInput : là component dùng để thêm công việc
* TodoFilter : là component dùng để lọc công việc

Một state cơ bản sẽ chứa các object công việc, nó bao gồm :

* id : mã công việc
* title : tên công việc
* isEditing : công việc này có đang được sửa hay không
* isComplete : công việc này có hoàn thành chưa

**6.2 Cách thực hiện**

**6.2.1 App.js**

Đây là nơi em viết các logic chính trong app. Class App sẽ trả về các thẻ <TodoItem>, <TodoInput>, <TodoFilter> để render ra màn hình.

* Với <TodoItem> em sẽ truyền các props cho nó như là:
  + item: đây là tham số chứa object công việc
  + onClick: truyền vô hàm khi nhấn vào dấu tích để đánh dấu là công việc đã hoàn thành
  + onDoubleClick: truyền vô hàm để sửa tên công việc
  + onChange: truyền vô hàm để có thể lưu khi sửa công việc
  + onBlur: truyền vô hàm để lưu công việc khi không nhập nữa
  + onDelete: truyền vô hàm để xóa công việc
* Với <TodoInput> em sẽ truyền vô các props như sau:
  + onKeyUp: truyền vô hàm để khi nhập xong công việc và nhấn Enter công việc sẽ được thêm vào
  + chooseAll: truyền vô hàm để đánh dấu tất cả các công việc đã hoàn thành
* Với <TodoFilter> em sẽ truyền vô các props như sau:
  + numItem: truyền vô số lượng công việc còn lại chưa hoàn thành
  + todoItems: truyền vô các công việc từ state
  + allFunc: truyền vô hàm hiển thị tất cả công việc
  + activeFunc: truyền vô hàm hiển thị tất cả công việc chưa hoàn thành
  + completeFunc: truyền vô hàm hiển thị tất cả công việc đã hoàn thành
  + clearCompleted: truyền vô hàm để xóa tất cả các công việc đã hoàn thành

**6.2.2 TodoInput.js**

Component này chỉ render ra một khung input cho nhập công việc và một nút để đánh dấu tất cả công việc là đã hoàn thành. Ngoài ra, component này sẽ nhận các props truyền giống như ở mục **6.2.1**

**6.2.3 TodoFilter.js**

Component này render ra các nút như:

* Hiển thị tất cả các công việc
* Hiển thị các công việc chưa hoàn thành
* Hiển thị các công việc đã hoàn thành
* Xóa tất cả các công việc

Component này nhận các props truyền giống như ở mục **6.2.1**

**6.2.4 TodoItems.js**

Component này render ra tên công việc, nút để đánh dấu công việc đã hoàn thành và nút để xóa công việc. Ngoài ra, component này nhận các props truyền giống như ở mục **6.2.1**

**6.3 Source code**

Tất cả source code nằm ở trong link github sau:

[internSCC/React/todo-list at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/todo-list)

**6.4 Demo**

[React App (internscc-todo.herokuapp.com)](https://internscc-todo.herokuapp.com/)

Một số hình ảnh:

Text, table

Description automatically generated with medium confidence

Graphical user interface, text, application

Description automatically generated

**CHƯƠNG 6: TÌM HIỂU VỀ AXIOS, ROUTER, ANT DESIGN VÀ REDUX**

* 1. **Axios**

**1.1.1 Get request**

* + - Đây là phương thức để lấy toàn bộ data về, chính vì thế get request thường được đặt trong life cycle componentDidMount để lấy dữ liệu :

  componentDidMount() {

    axios

      .get("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data")

      .then((res) => {

        this.setState({ todoItems: res.data });

      })

      .catch((err) => {

        console.error(err);

      });

  }

**1.1.2 Post request**

* + - Post request dùng để tạo dữ liệu mới

Axios.post("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data", {

      id: n,

      title: event.target.value,

      isEditing: false,

      isCompleted: false,

    })

    .then((res) => {

      console.log(res);

    })

    .catch((err) => {

      console.log(err);

});

**1.1.3 Put request**

* + - Put request dùng để cập nhật lại dữ liệu theo id

axios.put("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" + item.id,

{

title: text,

})

.then((res) => {

console.log(res.data);

})

.catch((err) => {

console.log(err);

});

**1.1.4 Delete request**

* + - Delete request dùng để xóa dữ liệu theo id

axios.delete("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

item.id

)

.then((response) => {

console.log(response);

});

**1.1.5 Vấn đề về too many request**

* + - Khi thực hiện gửi đồng loạt nhiều request trong một lúc cho server thì sẽ gặp hiện tượng lỗi too many request (error 429). Để khắc phục vấn đề này ta sẽ setTimeout cho mỗi lần lặp gửi request.

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

  setTimeout(() => {

    axios.put(

      "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

        todoItems[i].id,

      {

        isCompleted: todoItems[i].isCompleted,

      }

    )

    .then((res) => {

      console.log(res.data);

    })

    .catch((err) => {

      console.log(err);

    });

  }, 5000);

}

* + - Tuy nhiên trong JS, hàm setTimeout chỉ delay được duy nhất 1 lần trườc khi thực hiện hàm chính vì thế vẫn xuất hiện lỗi too many request
    - Để giải quyết vấn đề trên, ta sẽ tạo luôn một hàm để lặp và gọi lại hàm đó trong mỗi lần lặp.

function myLoop() {

      setTimeout(function () {

        axios

          .put(

            "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

              todoItems[i].id,

            {

              isCompleted: todoItems[i].isCompleted,

            }

          )

          .then((res) => {

            console.log(res.data);

          })

          .catch((err) => {

            console.log(err);

          });

        i++; //  increment the counter

        if (i < n) {

          myLoop();

        }

      }, 700);

    }

* + - Khi đó, hàm myLoop sẽ được lặp lại sau 700ms
    - Tuy nhiên cách làm này chưa được tối ưu, bởi vì khi muốn thay đổi thời gian timeout phải sửa lại toàn bộ hàm. Chính vì thế ta sẽ sửa lại hàm này với async await.

async function wait(ms) {

      return new Promise((resolve) => {

        setTimeout(resolve, ms);

      });

    }

    async function updateAPI(todoList) {

      for (const item of todoList) {

        await axios

          .put(

            "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/todo\_data/" +

              item.id,

            {

              isCompleted: item.isCompleted,

            }

          )

          .then((res) => {

            console.log(res.data);

          })

          .catch((err) => {

            console.log(err);

          });

        await wait(5000);

      }

    }

* + - Với hàm wait, ta có thể tùy ý cài đặt thời gian timeout mà không cần viết lại hàm
  1. **Router**

**1.2.1 Cài đặt**

* + - Thực hiện lệnh sau trong terminal: npm install react-router-dom
    - Để điều hướng chúng ta sử dụng các thẻ Router, Switch, Route, Link. Ví dụ:

<Router>

  <div>

    <Link to="/">Home</Link>

    <Link to="/about">About</Link>

    <Link to="/topics">Topics</Link>

  </div>

  <Switch>

    <Route exact path="/about">

      <About />

    </Route>

    <Route exact path="/topics">

      <Topics />

    </Route>

    <Route exact path="/">

      <Home />

    </Route>

  </Switch>

</Router>;

* + - Tất cả các phần để điều hướng được đặt trong thẻ Router. Thẻ Switch dùng để điều hướng khi nhấn vào các thẻ Link

**1.2.2 Sử dụng trong Todo**

* + - Trong Todo có 3 loại filter là All (tất cả các công việc), Active (các công việc chưa hoàn thành), Completed (các công việc đã hoàn thành). Vậy nên ta sẽ chuyển 3 nút này thành 3 route.

**Trong app.js**

<Router>

<TodoInput onKeyUp={this.onPressEnter} chooseAll={this.chooseAll} />

   <Switch>

    <Route exact path="/">

    <TodoGeneral path="" todoState={this.state} />

    </Route>

    <Route exact path="/active">

    <TodoGeneral path="active" todoState={this.state} />

    </Route>

    <Route exact path="/completed">

    <TodoGeneral path="completed" todoState={this.state} />

    </Route>

   </Switch>

   <TodoFilter numItem={numItem} clearCompleted={this.clearCompleted} />

</Router>

**Trong TodoFilter.js**

<div className="TodoFilter">

<p className="numItem">{numItem} left</p>

   <div className="filter" onClick={routeFunc}>

    <Link to="/">All</Link>

        <Link to="/active">Active</Link>

        <Link to="/completed">Completed</Link>

   </div>

   <button className="clearCompleted" onClick={clearCompleted}>Clear completed</button>

</div>

* 1. **Ant Design**

**1.3.1 Cài đặt vào project**

* + - Ta sẽ sử dụng Ant Design để tạo table và navbar điều hướng cho app.
    - Table ở đây sẽ lấy data từ Mock api về và cho phép người dùng thêm xóa sửa cũng như tìm kiếm trên table
    - Trước hết ta định nghĩa table gồm các cột sau :
      * ID : mã todo
      * Title : tên todo
      * Is editing ? : trạng thái isEditing của todo (true/false)
      * Is completed ?: trạng tháng isCompleted của todo (true/false)
      * Action : sửa hay xóa todo
    - Sau đó ta tạo các nút thêm và tìm kiếm trong table
    - Cuối cùng ta tạo các modal để sửa và thêm todo

Graphical user interface, application, Teams

Description automatically generated**1.3.2 Demo**

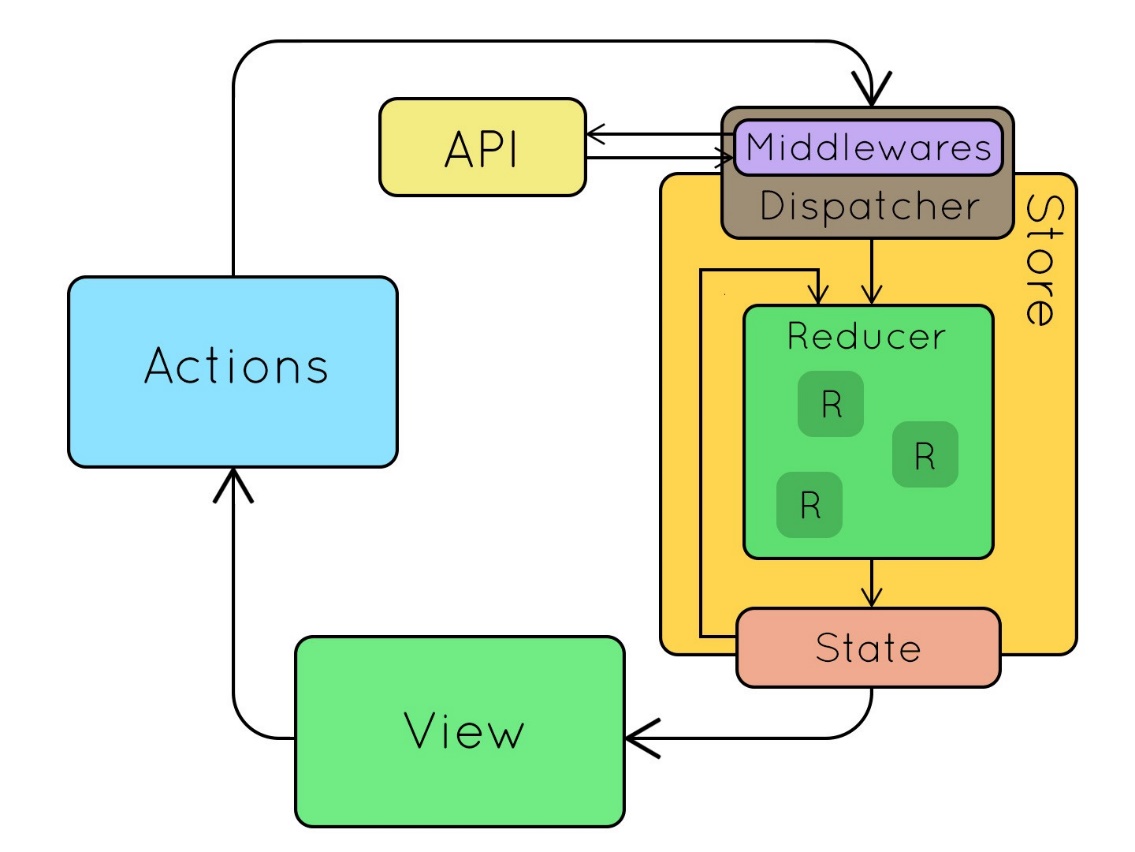
**Graphical user interface, text, application, email

Description automatically generated**

* 1. **Redux**

**1.4.1 Giới thiệu**

* + - Redux ra đời với mục đích quản lý state có thể dự đoán được. cung cấp một vùng chứa state duy nhất và các quy tắc nghiêm ngặt về cách có thể thay đổi state.
    - Có thể sử dụng Redux cho React, Angular, …



**1.4.2 Concept chính**

* + - Store : đây là nơi lưu trữ state và chỉ nên có 1 store trong 1 app
    - Action: một đối tượng JS đơn thuần. Action mô tả rõ ràng lý do tại sao thay đổi state và có thể thực hiện ở bất kỳ đâu trong app. Ví dụ:

{

  type: "ADD\_STUDENT",

  name: "Thu Hương"

}

* + - Reducer: là thành phần để liên kết store với action. Nó nhận state và action làm đối số và trả về state tiếp theo của ứng dụng.
    - Một reducer có thể xử lý nhiều action chính vì thế thường sử dụng cấu trúc switch trong reducer
    - Tốt nhất là cho mỗi reducer quản lý 1 phần state
    - Ví dụ:

function studentsApp(state, action) {

  // Câu lệnh switch

  // Kiểm tra type và trả về sate mới

  switch (action.type) {

    case "ADD\_STUDENT":

      return [ ...state,  action.name ]

    default:

      // Không có case nào khớp

      // thì trả về state hiện tại

      return state

}

**CHƯƠNG 7: TỐI ƯU TODOS VÀ TÌM HIỂU VỀ PHÂN TRANG TRONG ANT DESIGN**

* 1. **Tối ưu todos**

**2.1.1 Vấn đề về component cha và con và thêm proptypes vào các component**

* Hiện tại các xử lý logic chính nằm ở trong component con, vì thế để tiện cho việc dò lỗi và update, nên chuyển tất cả các logic đó cho component cha và các component con chỉ nhận props từ component cha.
* Để đảm bảo cho việc mỗi props truyền vô component là chính xác, việc thêm proptypes vào các components là cần thiết. Ví dụ :

import PropTypes from "prop-types";

TodoFilter.propTypes = {

  numItem: PropTypes.number,

  clearCompleted: PropTypes.func,

  onClickAll: PropTypes.func,

  onClickActive: PropTypes.func,

  onClickCompleted: PropTypes.func,

};

**2.1.2 Vấn đề về tối ưu logic update và search**

* Mỗi lần update và search trong table đều phải get data lại để hiển thị trong màn hình. Điều này làm chương trình chạy rất chậm và không hiệu quả.
* Trong Mockapi, khi ta thực hiện một method bất kì nó sẽ trả về data của method đó, chính vì thế, việc lấy data từ response của Mockapi để hiển thị data sẽ giúp cải thiện tốc độ rất nhiều

**2.1.3 Vấn đề về validate data**

* Trong Ant Design đã cũng cấp 1 số tính năng để validate data khi nhập vào bảng. Ví dụ trong một form thì props “rules” sẽ quy định data khi nhập vô sẽ thuộc kiểu nào.

rules={[

{

   required: true,

    pattern: new RegExp("([0-9]\\s\*)+"),

    message: "please input phone num",

  },

* 1. **Phân trang trong Ant Design**

**2.2.1 Tại sao phải phân trang ?**

* Trong table của Ant Design đã tích hợp sẵn công cụ phân trang. Chỉ cần truyền dữ liệu cho table thì lập tức nó sẽ phân trang theo dữ liệu đó.
* Tuy nhiên với một dữ liệu cực lớn (bảng có 1 triệu dòng chẳng hạn) việc load toàn bộ data này sẽ rất mất thời gian khi tải trang. Chính vì thế nên phân thành những trang nhỏ để mỗi lần chuyển trang sẽ lấy đúng data mà trang đó có.

**2.2.2 Cách phân trang khi lấy dữ liệu từ api**

* Phân trang trong Ant Design cần thuộc tính “pagination”, ngoài ra, cần thêm thuộc tính “onChange” để có thể xử lý logic khi chuyển trang.

<Table

        columns={columns}

        dataSource={data}

        pagination={tablePagination}

        loading={loading}

        onChange={handleTableChange}

      />

* Đây là hàm xử lý khi thay đổi một trang. Hàm fetch() có nhiệm vụ lấy dữ liệu theo trang được truyền vô

const handleTableChange = (page, pageSize) => {

    let currentPage = {

      current: (page.current - 1) \* page.pageSize,

      pageSize: pageSize,

    };

    console.log(page);

    fetch(currentPage);

  };

const tablePagination = {

    pageSize: paramsContact.limit,

    page: paramsContact.offset / paramsContact.limit + 1,

    total: paramsContact.total,

    pageSizeOptions: ["5", "10", "15", "20", "25"],

    showSizeChanger: true,

  };

* 1. **Source code**

Tất cả source code nằm trong link github sau:

[internSCC/React/todo-list-optimize at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/todo-list-optimize)

* 1. **Demo**

**Graphical user interface, application, Teams

Description automatically generated**

**CHƯƠNG 8: THỰC HIỆN DEMO REDUX**

* 1. **Ý tưởng**
* Em sẽ xây dựng app cho người dùng đăng ký, đăng nhập. Nếu đăng nhập thành công thì sẽ chuyển đến trang chính để có thể thêm, xóa, sửa dữ liệu.
* Phần đăng ký em sẽ sử dụng thư viện “bcrypt” để mã hóa mật khẩu và lưu vô database. Đồng thời chuyển cặp giá trị username, password thành một token bằng jwt và lưu lại trong database.
* Khi người dùng đăng nhập, mật khẩu sẽ được so sánh với mật khẩu đã mã hóa. Nếu đúng sẽ cho người dùng quyền truy cập vào trang chính đồng thời đưa cho người dùng token và lưu vào trong local storage.
* Khi đăng xuất thì app sẽ xóa token đó trong local storage và chuyển về trang đăng nhập
* Trang chính sẽ hiển thị thông tin người dùng bao gồm họ tên, số điện thoại, giới tính

**3.2** **Cấu trúc thư mục**

**Graphical user interface, application

Description automatically generated**

* Thông thường sẽ có 2 cách chia thư mục: Function-first và Feature-first. Nhưng ở đây em làm theo Function-first.
* Trong *index.js*, em sẽ tạo store đồng thời apply middle *Thunk* để xử lý async await function vào project.

**3.2.1 Actions**

* Đây là nơi đặt tất cả các actions, xử lý logic của app. Em chia ra 2 file action chính là *auth.js* để xử lý đăng ký, đăng nhập và *contacts.js* để xử lý logic các chức năng thêm xóa sửa dữ liệu.
* Với action đăng ký em sẽ trả về tên của action này và state của nó sau khi thực hiện action

export const register = (user) => {

  return {

    type: types.REGISTER,

    user,

  };

};

* Đồng thời, em sẽ khai báo một hàm xử lý logic và gọi action này mỗi khi nhấn đăng ký

export const registerRequest = (user) => {

  return (dispatch) => {

    const salt = bcrypt.genSaltSync(10);

    const hassPassword = bcrypt.hashSync(user.password,salt);

    const token = jwt.sign(

      {

        username: user.username,

        password: hassPassword,

      },

      "SECRET\_TOKEN"

    );

    const data = {

      username: user.username,

      password: hassPassword,

      token: token,

    };

    axios

      .post("https://60b0f8b91f26610017fff943.mockapi.io/api/v1/users", data)

      .then((response) => {

        dispatch(register(response.data));

      })

      .catch((err) => {

        console.log(err);

      });

  };

};

* Tương tự như vậy với đăng nhập và các actions khác.

**3.2.2 Components**

* Đây sẽ là nơi khai báo tất cả các component và nếu muốn gọi một action thì em sẽ gọi thông qua props. Để làm được điều này em sẽ thêm một hàm là *mapDispatchToProps* và sử hàm *connect()* để nối dispatch props với lại component. Ví dụ cho form đăng nhập:

const mapDispatchToProps = (dispatch) => {

  return {

    loginRequest: (user) => {

      dispatch(loginRequest(user));

    },

  };

};

export default connect(null, mapDispatchToProps)(FormLogin);

**3.2.3 Constants**

* Đây là nơi khai báo tên của những actions có trong app, mục đích là để mỗi lần gọi tên thì sẽ tránh trường hợp gọi nhầm tên và dễ kiểm soát lỗi hơn.

export const SET\_CURRENT\_USER = "SET\_CURRENT\_USER";

export const LOGIN = "LOGIN";

export const REGISTER = "REGISTER";

export const FETCH\_CONTACT = "FETCH\_CONTACT";

export const ADD\_CONTACT = "ADD\_CONTACT";

export const EDIT\_CONTACT = "EDIT\_CONTACT";

export const DELETE\_CONTACT = "DELETE\_CONTACT";

export const SEARCH\_CONTACT = "SEARCH\_CONTACT";

**3.2.4 Reducers**

* Đây sẽ là nơi quản lý state đầu vào, đầu ra. Bởi vì không nên để một file quản lý tất cả các state nên em sẽ chia thành nhiều file nhỏ quản lý những nhóm state. Cuối cùng em sẽ gộp lại bằng hàm *combineReducer()*

export default combineReducers({

  loginReducer,

  registerReducer,

  contactReducer,

  searchContactReducer,

  auth

});

* 1. **Source code**

Tất cả source code nằm trong link github sau:

[internSCC/React/redux-app at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/redux-app)

* 1. **Graphical user interface, text, application, email

     Description automatically generatedDemo**

**Graphical user interface, text, application, email

Description automatically generated**

**Graphical user interface, text, application, Teams

Description automatically generated**

**CHƯƠNG 9: TỐI ƯU DEMO REDUX VÀ XÂY DỰNG APP SỬ DỤNG REDUX SAGA VIẾT BẰNG TYPESCRIPT**

* 1. **Tối ưu demo Redux và chuyển từ function-first thành feature-first**
     1. **State trong reducers:**
* Trong contactReducer.js:

const contactReducer = (state = initialState, action) => {

  switch(action.type) {

    case types.FETCH\_CONTACT:

      const contacts = action.contacts;

      return { ...state, contacts };

    case types.ADD\_CONTACT:

      state = action.contacts;

      return state;

    case types.DELETE\_CONTACT:

      state = action.contact;

      return state;

    case types.EDIT\_CONTACT:

      state = action.contact;

      return state;

    default:

      return state;

  }

};

* Với action ADD\_CONTACT, việc trả về một state thì sẽ mất những state trước đó, trong khi đó chỉ cần thêm contact vừa được add và state cũ là được. Tương tự như các action khác. Vì thế sửa lại contactReducer như sau:

const contactReducer = (state = initialState, action) => {

  switch (action.type) {

    case types.FETCH\_CONTACT:

      const contacts = action.contacts;

      return { ...state, contacts };

    case types.ADD\_CONTACT:

      state.contacts.push(action.contact);

      return { ...state };

    case types.DELETE\_CONTACT:

      let index = state.contacts.length;

      while (index--) {

        if (state.contacts[index] === action.contact) {

          state.contacts.splice(index, 1);

          break;

        }

      }

      const delContact = action.contact;

      return { ...state, delContact };

    case types.EDIT\_CONTACT:

      for(let i = 0; i < state.contacts.length; i++) {

        if(state.contacts[i].id === action.contact.id) {

          state.contacts[i] = action.contact;

          break;

        }

      }

      const contactEdit = action.contact;

      return { ...state, contactEdit };

    default:

      return state;

  }

};

* + 1. **Form**
* Trong AddModal.js:

const ho\_va\_ten = document.getElementById("nest-messages\_ho\_va\_ten").value;

    const sdt = document.getElementById("nest-messages\_sdt").value;

    const gender = document.getElementById("nest-messages\_gender").value;

    const age = document.getElementById("nest-messages\_age").value;

* Không nên sử dụng document.getElementById vì nó ảnh hưởng đến DOM thật mà trong Ant Design có hỗ trợ lấy value trong form nên em sẽ sửa lại như sau:

const ho\_va\_ten =  form.getFieldValue('ho\_va\_ten');

const sdt = form.getFieldValue('sdt');

const gender = form.getFieldValue('gender');

const age = form.getFieldValue('age');

* + 1. **Cấu trúc thư mục**

Graphical user interface, text, application, chat or text message

Description automatically generated

* Ta có thể thấy AddModal, EditModal và TableContacts là những component cùng cấp với nhau chính vì thế em sẽ cấu trúc lại như sau:

Graphical user interface, text, application

Description automatically generated

* + 1. **API**
* Trong mỗi action gọi api, việc viết lại đường dẫn sẽ mất rất nhiều thời gian nên em sẽ tạo sẵn một constant để chứa đường dẫn api để mỗi lần sử dụng sẽ tiện hơn
* urlApi.js:

export const contactData =

  "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/contact\_data/";

export const userData =

  "https://60b0f8b91f26610017fff943.mockapi.io/api/v1/users/";

* Khi đó em chỉ cần khai báo mỗi lần gọi api như sau :

axios

.get(contactData)

.then((res) => {

   dispatch(fetchContact(res.data));

   })

   .catch((err) => {

    console.log(err);

   });

* + 1. **Chuyển từ function-first thành feature-first**
* Tùy theo chức năng mà em sẽ chia nó thành một thư mục riêng (ví dụ như chức năng đăng kí đăng nhập em sẽ để thành folder *session*).
* Khi đó em tách cái file cũ trong folder action và reducer và bỏ vào các feature tương ứng
* Như vậy, với feature-first, em sẽ cấu trúc lại thư mục như sau

A screenshot of a computer screen

Description automatically generated with medium confidence

* 1. **Xây dựng app sử dụng Redux Saga viết bằng TypeScript**
     1. **Ý tưởng**
* Em sẽ chuyển app redux lần trước được viết bằng JavaScript thành TypeScript đồng thời sẽ quản lý các action trong redux bằng redux saga
  + 1. **Cách thực hiện**
* Đầu tiên, em sẽ thiết lập Redux Saga vào project react với folder tên là saga như sau :

**A screenshot of a computer screen

Description automatically generated with medium confidence**

* Thư mục này sẽ chứa các logic xử lý side effect trong redux cũng như thiết lập saga :
* Bên trong ***rootSata.ts :***

import { all } from "redux-saga/effects";

import { sessionSaga } from "./sessionSaga";

import { contactSaga } from "./contactSaga";

export default function\* rootSaga() {

  yield all([sessionSaga(), contactSaga()]);

}

* Lệnh yield all() sẽ gọi tất cả các file xử lý logic mỗi khi nghe được một action nào đó được gọi.
* Em sẽ sử dụng generator function để xử lý các logic trong project. Với mỗi hàm, em sẽ sử dụng lệnh yield. Ví dụ như sau:
* Bên trong ***sessionSaga.ts :***

function\* handleLogin(action: any) {

  const user = action.user;

  yield axios.get(userData).then((response) => {

    for (const userInt of response.data) {

      if (user.username === userInt.username) {

        bcrypt.compare(user.password, userInt.password).then((valid: any) => {

          if (valid) {

            const token = userInt.token;

            window.localStorage.setItem("jwtToken", token);

          }

        });

        break;

      }

    }

  });

  yield delay(500);

  yield put({

    type: types.SET\_CURRENT\_USER,

    auth: {

      username: user.username,

      password: user.password,

      token: window.localStorage.getItem("jwtToken"),

    },

  });

}

function\* handleRegister(action: any) {

  const user = action.user;

  const salt = bcrypt.genSaltSync(10);

  const hassPassword = bcrypt.hashSync(user.password, salt);

  const token = jwt.sign(

    {

      username: user.username,

      password: hassPassword,

    },

    "SECRET\_TOKEN"

  );

  const data = {

    username: user.username,

    password: hassPassword,

    token: token,

  };

  yield axios

    .post(userData, data)

    .then((response) => {

      console.log(response);

    })

    .catch((err) => {

      console.log(err);

    });

}

export function\* sessionSaga() {

  yield takeEvery(types.LOGIN, handleLogin);

  yield takeEvery(types.REGISTER, handleRegister);

}

* Tất cả các logic gọi Api đều được xử lý mỗi khi action tương ứng được thực hiện thông qua lệnh takeEvery. Ví dụ : Khi người dùng bấm vào nút đăng nhập, lập tức saga sẽ lắng nghe được và xử lý nhờ vào hàm handleLogin.
* Tương tự như ***sessionSaga***, ***contactSaga*** cũng xử lý logic khi nghe được action.
* Như vậy bên trong file ***sessionActions.ts*** và ***contactActions.ts*** chỉ còn lại những action trả về type và dữ liệu chứ không còn hàm xử lý logic nữa.
  + 1. **Vấn đề của redux thunk**
* Redux thunk cho phép sử dụng **mapDispatchToProps** rồi gọi các action thông qua props. Điều này làm code nở ra rất nhiều nếu có nhiều action.
* Chính vì thế em sẽ sử dụng **useDispatch** để gọi action và code sẽ trở nên gọn hơn.
  + 1. **Source code**
* Tất cả source code nằm trong link github sau:

[internSCC/React/redux-saga-ts at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/redux-saga-ts)

* + 1. **Demo**

**Graphical user interface, text, application, email

Description automatically generated**

**Graphical user interface, text, application, Teams

Description automatically generated**

**CHƯƠNG 10: TÌM HIỂU VỀ WEBPACK VÀ JEST**

* 1. **Webpack**
     1. **Webpack là gì?**
* Webpack là công cụ giúp compile các module Javascript. Nó hay được gọi là “module bundler”.
  + 1. **Ưu điểm**
* Dành cho các dự án lớn, dễ dàng phát triển, quản lý và customize
* Tăng tốc độ cho project
* Phân chia các module và chỉ load khi thật sự cần
* Đóng gói tất cả các file nguồn thành 1 file duy nhất. Nhờ vào loader mà nó có thể biên dịch các loại file khác nhau.
* Biến các tài nguyên tĩnh (image, css,..) thành module.
* Biến đổi các mã nguồn: js, less, sass -> js, css or es6 -> es5 (nhằm hỗ trợ các trình duyệt)
* Đóng gói các module => sẽ giải duyết được các đường dẫn trong react
* JSX -> JS ( Browser sẽ không hiểu JSX nên webpack sẽ giúp chuyển đổi)
* Áp dụng Css preprocessor để xây dựng stylesheet (ví dụ: SASS)
  + 1. **Cài đặt**
       1. **Khởi tạo các file cơ bản:**
* Đầu tiên, em sẽ init npm vào project và tạo ra các file như sau:

**Text

Description automatically generated**

* File ***index.html***chỉ có head và body rỗng, đồng thời import file ***bundle.js*** mà lát nữa em sẽ tạo sau thông qua webpack.
* File ***math.js*** em sẽ viết một hàm cơ bản trả về 1 số và export nó ra:

const sum = (a, b) => {

    return a + b;

}

export default sum;

* File ***index.js*** sẽ import file ***math.js*** và sẽ đưa kết quả tính toán vào trong trang html:

import sum from "./math";

const total = sum(3, 5);

document.write(total);

console.log(total);

* Bây giờ em sẽ cài đặt webpack và viết file config cho nó
  + - 1. **Thiết lập file config**
* Em sẽ sử dụng các module sau:
  + Babel: dùng để chuyển đổi ES6, 7, 8,.. thành ES5
  + styleLoader, cssLoader, MiniCssExtractPlugin: tải và tách file css trong webpack
* Em sẽ tạo file tên là ***webpack.config.js*** trong đó sẽ chứa thông tin các cài đặt cho webpack, các module, plugins của webpack. Ở đây em sẽ tạo tên folder chứa file là *dist* và file được tạo ra sau khi chạy webpack là ***bundle.js***
* Em sẽ chạy lệnh webpack và nó sẽ tạo ra folder như sau

A screenshot of a computer

Description automatically generated with medium confidence

* Trong đó bundle.css là những cài đặt về css trong project và bundle.js là tất cả xử lý logic cũng như khai báo trong js của project.
  + 1. **React với webpack**
* Tương tự với cách làm trên, em sẽ áp dụng webpack vào project em đã tạo với React sử dụng Ant Design.
* Để sử dụng được Ant Design, em sẽ thêm vào file ***.babelrc*** và ***webpack.config.js*** các khai báo như sau:
* ***.babelrc:***

{

  "presets": ["@babel/preset-env", "@babel/preset-react"],

  "plugins": [["import", { "libraryName": "antd", "style": true }]],

  "env": {

    "development" : {

      "compact": false

    }

  },

  "compact": true,

}

* ***webpack.config.js:*** em sẽ thêm plugin less loader để sử dụng được các file css trong Ant Design:

{

loader: "less-loader", // compiles Less to CSS

  options: {

   lessOptions: {

   javascriptEnabled: true,

   },

  },

},

* Sau khi chạy lệnh webpack em được folder như sau :

A screenshot of a video game

Description automatically generated with medium confidence

* + 1. **Source code**
* Tất cả source code nằm trong link github sau:

[internSCC/React/demo-webpack at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/demo-webpack)

* + 1. **Demo**
* Khi em mở file index.html, em đã có thể sử dụng web giống như chạy server

A screenshot of a computer

Description automatically generated

* 1. **Jest**
     1. **Giới thiệu**
* Jest là công cụ dùng để unit testing trong React. Em sẽ sử dụng công cụ này để test các component trong React.
* Em sẽ viết các test case với yêu cầu như sau:
  + Kiểm tra xem các component đã render đúng chưa
  + Trong form đăng nhập, mật khẩu có ít nhất 6 kí tự và tối đa 20 kí tự
* Em sẽ sử dụng Jest với Enzyme để test component tiện hơn
  + 1. **Cài đặt và sử dụng Jest với Enzyme**
* Em sẽ cài đặt các thư viện sau:
  + @testing-library/jest-dom
  + Enzyme
  + enzyme-adapter-react-16
  + @babel/preset-env
  + @babel/preset-react
* Em sẽ tạo thư mục *\_\_test\_\_* để chứa file test ***FormLogin.test.js***, đồng thời em sẽ tạo file ***babel.config.js*** và ***jest.config.js*** để khai báo các cài đặt
* Trong file ***FormLogin.test.js*** em sẽ tạo 3 test cho 3 yêu cầu ở trên:
  + Kiểm tra xem component đã render đúng chưa:

describe("Form login component", () => {

  it("should render form", () => {

    const wrapper = shallow(<FormLogin />);

    const form = wrapper.find(".form-login");

    expect(form.hasClass("login-form-button"));

  });

});

* + Mật khẩu có ít nhất 6 kí tự :

describe("Password min 6 chars", () => {

  it("should have min 6 chars", () => {

    const wrapper = mount(

      <StaticRouter>

        <FormLogin />

      </StaticRouter>

    );

    let password = wrapper.find("input#normal\_login\_password");

    password.simulate("change", {

      target: { value: "12345" },

    });

    password = wrapper.find("input#normal\_login\_password")

    expect(password.props().value).toEqual("12345");

  });

});

* + Mật khẩu có tối đa 20 kí tự :

describe("Password max 20 chars", () => {

  it("should have max 20 chars", () => {

    const wrapper = mount(

      <StaticRouter>

        <FormLogin></FormLogin>

      </StaticRouter>

    );

    let password = wrapper.find("input#normal\_login\_password");

    password.simulate("change", {

      target: { value: "1234512312312123123131231231231231" },

    });

    password = wrapper.find("input#normal\_login\_password");

    expect(password.props().value).toEqual(

      "1234512312312123123131231231231231"

    );

  });

});

* + 1. **Kết quả**

**Text

Description automatically generated**

* + 1. **Source code**
* Tất cả source code nằm trong link github sau:

[internSCC/React/demo-jest at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/demo-jest)

**CHƯƠNG 11: XÂY DỰNG APP THEO UMIJS VÀ DVAJS**

* 1. **Ý tưởng**
* Em sẽ chuyển app redux lần trước theo UmiJS và quản lý redux bằng DvaJS
  1. **Cách thực hiện**
* Đầu tiên em cài đặt DvaJS và UmiJS qua lệnh terminal sau:

yarn create @umijs/umi-app

npm install dva-cli -g

* Sau đó em cấu trúc lại thư mục như sau

**A screenshot of a computer

Description automatically generated with low confidence**

* Trong đó folder *models* chứa các file quản lý state trong redux và folder *pages* chứa các file theo feature.
* Em sẽ config route ở trong file cài đặt ***.umirc.ts*** như sau:

export default defineConfig({

  dva: {},

  nodeModulesTransform: {

    type: 'none',

  },

  routes: [

    { exact: true, path: '/', component: '@/pages/common/index' },

    { exact: true, path: '/login', component: '@/pages/session/FormLogin' },

    {

      exact: true,

      path: '/register',

      component: '@/pages/session/FormRegister',

    },

  ],

  fastRefresh: {},

});

* File ***session.ts*** ở trong folder *models* để quản lý state khi đăng kí, đăng nhập.

export default {

  namespace: 'session',

  state: {

    register: { username: '', password: '' },

    login: {

      username: '',

      password: '',

    },

    auth: window.localStorage.getItem('jwtToken'),

  },

  reducers: {

    login(state: Session, action: any) {

      return { ...state, login: action.user };

    },

    auth(state: Session, action: any) {

      return { ...state, auth: action.token };

    },

    register(state: Session, action: any) {

      return { ...state, register: action.user };

    },

  },

  effects: {

    \*handleLogin(action: any, { call, put }: { call: any; put: any }) {

      const user = action.user;

      yield axios.get(userData).then((response) => {

        for (const userInt of response.data) {

          if (user.username === userInt.username) {

            bcrypt

              .compare(user.password, userInt.password)

              .then((valid: any) => {

                if (valid) {

                  const token = userInt.token;

                  window.localStorage.setItem('jwtToken', token);

                }

              });

            break;

          }

        }

      });

      yield put({

        type: 'login',

        user: user,

      });

      yield call(delay, 500);

      yield put({

        type: 'auth',

        token: window.localStorage.getItem('jwtToken'),

      });

    },

    \*handleRegister(action: any, { put }: { put: any }) {

      const user = action.user;

      const salt = bcrypt.genSaltSync(10);

      const hassPassword = bcrypt.hashSync(user.password, salt);

      const token = jwt.sign(

        {

          username: user.username,

          password: hassPassword,

        },

        'SECRET\_TOKEN',

      );

      const data = {

        username: user.username,

        password: hassPassword,

        token: token,

      };

      yield axios

        .post(userData, data)

        .then((response) => {

          console.log(response);

        })

        .catch((err) => {

          console.log(err);

        });

      yield put({

        type: 'register',

        user: user,

      });

    },

  },

};

* Trong đó *state* để khởi tạo state ban đầu, *reducers* để lắng nghe các action mỗi khi được gọi và trả về state tương ứng, *effects* để xử lý các logic liên quan.
* Khi đó, trong file ***FormLogin.tsx*** em sẽ cài đặt để gọi action như sau :

const dispatch = useDispatch();

  const onFinish = (values: User) => {

    dispatch({ type: 'session/handleLogin', user: values });

    setTimeout(() => {

      let path = '/';

      history.push(path);

    }, 2000);

  };

* Tương tự với các feature còn lại
  1. **Source code**
* Tất cả source code nằm trong link github sau:

[internSCC/React/umijs-dva at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/umijs-dva)

* 1. **Demo**

**Graphical user interface, text, application, email

Description automatically generatedGraphical user interface, text, application, Teams

Description automatically generated**

**CHƯƠNG 12: XÂY DỰNG APP VỚI AUTH0 VÀ REFRESH TOKEN**

* 1. **Ý tưởng**
  2. **Cách thực hiện**
* Em sẽ clone lại project sử dụng UmiJS và DvaJS và thêm Auth0 và refresh token trong project này
  + 1. **Refresh token**
* Trong file ***sesstion.ts***

\*handleLogin(action: any, { call, put }: { call: any; put: any }) {

const user = action.user;

  yield axios.get(userData).then((response) => {

   for (const userInt of response.data) {

     if (user.username === userInt.username) {

       bcrypt

        .compare(user.password, userInt.password)

        .then((valid: any) => {

         if (valid) {

           const token = jwt.sign(

             {

             username: user.username,

             password: userInt.password,

             },

             'SECRET\_TOKEN',

             { expiresIn: '1h' },

         );

            const refreshToken = jwt.sign(

             {

               username: user.username,

                password: userInt.password,

              },

              'SECRET\_TOKEN',

              { expiresIn: '100d' },

            );

            window.localStorage.setItem('jwtToken', token);

            window.localStorage.setItem('refreshJwtToken', refreshToken);

          }      }

        });

        break;

      }

    }

  });

  yield put({

   type: 'login',

    user: user,

  });

  yield call(delay, 500);

  yield put({

   type: 'auth',

    token: window.localStorage.getItem('jwtToken'),

  });

},

* Như vậy em đã cấp phát token khi đăng nhập và nó có hiệu lực trong 1 giờ. Ngoài ra, em cấp phát thêm refresh token để cấp phát lại token khi nó hết hiệu lực.
* Trong file chứa component App:

function App(props: any) {

  const token: any = window.localStorage.getItem('jwtToken');

  let dec: any;

  if (token !== null) {

    jwt.verify(token, 'SECRET\_TOKEN', (err: any, decoded: any) => {

      console.log(err);

      dec = decoded;

      console.log(decoded);

    });

    if (dec === undefined) {

      const refreshToken: any = window.localStorage.getItem('refreshJwtToken');

      jwt.verify(refreshToken, 'SECRET\_TOKEN', (err: any, decoded: any) => {

        console.log(err);

        dec = decoded;

      });

      const newToken = jwt.sign(

        {

          username: dec.username,

          password: dec.password,

        },

        'SECRET\_TOKEN',

        { expiresIn: '1h' },

      );

      window.localStorage.setItem('jwtToken', newToken);

    }

  }

  const { isAuthenticated, isLoading, getAccessTokenSilently, user } =

    useAuth0();

  if (isLoading) {

    return <Spin></Spin>;

  }

  getAccessTokenSilently().then((token) => {

    window.localStorage.setItem('token', token);

  });

  console.log('is authenticated');

  console.log(isAuthenticated);

  return (

    <div className="App">

      {(token !== null ) || isAuthenticated === true ? (

        <Contact />

      ) : (

        <Redirect to="/login" />

      )}

    </div>

  );

}

* Em sẽ decode jwt và nếu như nó bị hết hạn thì sẽ sử dụng refresh token để tạo token mới.
  + 1. **Auth0**
* Đầu tiên em tạo app trong Auth0 và thêm địa chỉ <http://localhost:8000> như sau:

Graphical user interface, text, application

Description automatically generated

* Để sử dụng được Auth0, em sẽ thêm folder wrapper như sau:

Graphical user interface, text, website

Description automatically generated

* Trong file ***Auth0.tsx***

import { Auth0Provider } from '@auth0/auth0-react';

export default (props: any) => {

  return (

    <Auth0Provider

      domain="uytran.us.auth0.com"

      clientId="n8ZreS9jANtx5zEnhyYxFjEf4S1RyKt7"

      redirectUri="http://localhost:8000"

      useRefreshTokens={true}

    >

      {props.children}

    </Auth0Provider>

  );

};

* Domain và clientId được lấy từ cài đặt app trong Auth0
* Sau đó em config wrapper như sau:

export default defineConfig({

  dva: {},

  nodeModulesTransform: {

    type: 'none',

  },

  routes: [

    {

      exact: true,

      path: '/',

      component: '@/pages/common/index',

      wrappers: ['@/wrappers/Auth0'],

    },

    {

      exact: true,

      path: '/login',

      component: '@/pages/session/FormLogin',

      wrappers: ['@/wrappers/Auth0'],

    },

    {

      exact: true,

      path: '/register',

      component: '@/pages/session/FormRegister',

      wrappers: ['@/wrappers/Auth0'],

    },

  ],

  fastRefresh: {},

});

* Em thêm button đăng nhập vào Auth0 trong app của mình:

const { loginWithRedirect} = useAuth0();

<button onClick={() => loginWithRedirect()}>Login with Auth0</button>

* Tương tự với button logout:

const { logout } = useAuth0();

  //when click logout button

  const onClickLogout = () => {

    window.localStorage.removeItem('jwtToken');

    window.localStorage.removeItem('refreshJwtToken');

    window.localStorage.removeItem('token');

    logout();

  };

<Button

type="primary"

  size="large"

  onClick={props.onClickLogout}

>

Logout

</Button>

* Khi login thành công bằng Auth0, em sẽ lưu token của Auth0 vào local storage

getAccessTokenSilently().then((token) => {

window.localStorage.setItem('token', token);

});

* Như vậy là em có thể sử dụng được Auth0 trong app của mình.
  1. **Source code**
* Tất cả source code nằm trong link github sau:

[internSCC/React/auth0-umijs at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/auth0-umijs)

* 1. **A screenshot of a computer

     Description automatically generatedDemo**