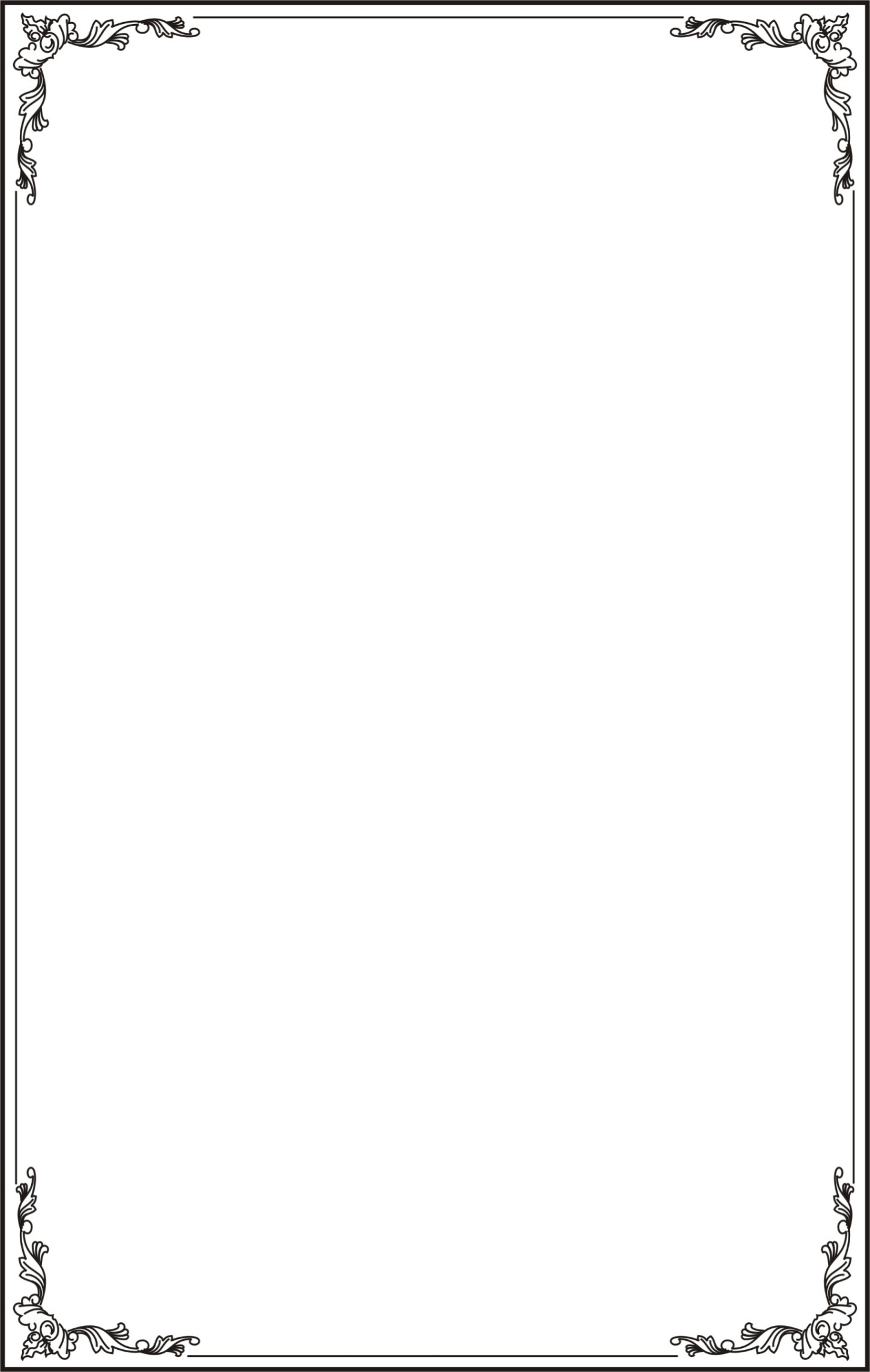
**CÔNG TY CỔ PHẦN**

**VIỄN THÔNG FPT**

****

**BÀI THU HOẠCH**

**ĐÀO TẠO TÂN BINH R&D TUẦN 1 & 2**

|  |  |
| --- | --- |
| Người hướng dẫn: | **MR. 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 05 năm 2021***

# MỤC LỤC

[MỤC LỤC 2](#_Toc73301791)

[CHƯƠNG 1: TỔNG QUAN VỀ SCC 3](#_Toc73301792)

[1.1 GIỚI THIỆU CHUNG 3](#_Toc73301793)

[1.2 Cơ cấu tổ chức 4](#_Toc73301794)

[1.3 Các đơn vị Ftel khác 5](#_Toc73301795)

[CHƯƠNG 2: TÌM HIỂU VỀ NHỮNG THAY ĐỔI TỪ ES6 – ES11 6](#_Toc73301796)

[2.1 ES6 6](#_Toc73301797)

[2.2 ES7 7](#_Toc73301798)

[2.3 ES8 7](#_Toc73301799)

[2.4 ES9 8](#_Toc73301800)

[2.5 ES10 9](#_Toc73301801)

[2.6 ES11 10](#_Toc73301802)

[CHƯƠNG 3: TÌM HIỂU VỀ SPA, VIRTUAL DOM VÀ REALDOM 11](#_Toc73301803)

[3.1 Single Page Application (SPA) 11](#_Toc73301804)

[3.2 Virtual DOM và Real DOM 12](#_Toc73301805)

[CHƯƠNG 4: TÌM HIỂU VỀ SEVER SIDE RENDERING & CLIENT SIDE RENDERING, NEXTJS, GASBY VÀ GENERATOR FUNCTION 14](#_Toc73301806)

[4.1 Server side rendering & client side rendering 14](#_Toc73301807)

[4.2 Nextjs 16](#_Toc73301808)

[4.3 Gasbyjs 16](#_Toc73301809)

[4.4 Generator function 17](#_Toc73301810)

[CHƯƠNG 5: TÌM HIỂU VỀ LIFECYCLE, HOOK 19](#_Toc73301811)

[5.1 Vòng đời của component 19](#_Toc73301812)

[5.1.1 Mounting 19](#_Toc73301813)

[5.1.2 Updating 20](#_Toc73301814)

[5.1.3 Unmounting 22](#_Toc73301815)

[5.1.4 Error handling 22](#_Toc73301816)

[5.1.5 Tổng kết về lifecycle component 22](#_Toc73301817)

[5.2 Hook 23](#_Toc73301818)

[CHƯƠNG 6 : DEMO REACT APPLICATION 26](#_Toc73301819)

[6.1 Ý tưởng 26](#_Toc73301820)

[6.2 Cách thực hiện 27](#_Toc73301821)

[6.2.1 App.js 27](#_Toc73301822)

[6.2.2 TodoInput.js 28](#_Toc73301823)

[6.2.3 TodoFilter.js 28](#_Toc73301824)

[6.2.4 TodoItems.js 28](#_Toc73301825)

[6.3 Source code 28](#_Toc73301826)

[6.4 Demo 28](#_Toc73301827)

[6.5 Mở rộng 29](#_Toc73301828)

[6.5.1 Những thay đổi 29](#_Toc73301829)

[6.5.2 Source code 29](#_Toc73301830)

[6.5.3 Demo 29](#_Toc73301831)

[6.4 Những hướng đi tiếp theo 29](#_Toc73301832)

# CHƯƠNG 1: TỔNG QUAN VỀ SCC

## GIỚI THIỆU CHUNG

Trung tâm Giám sát và đảm bảo dịch vụ (SCC – Service Command Center) – Trực thuộc Công ty cổ phần Viễn thông FPT - tiền thân là NOC-MON được thành lập từ 10/2012 sau này chuyển thành SCC từ 01/2017 với nhiệm vụ cung cấp 3 loại dịch vụ như sau:

* Giám sát hệ thống
* Hỗ trợ kỹ thuật và xử lý sự cố
* Theo dõi toàn bộ quá trình xử lý và escalate lên cấp L2+
* Kiểm soát và theo dõi toàn bộ Kế hoạch bảo trì của Ftel
* Tối ưu hóa công cụ hỗ trợ cho hoạt động giám sát/ vận hành hệ thống
* Phân tích dữ liệu, báo cáo tình hình hệ thống mạng/ dịch vụ gia tăng

### 1.2 Cơ cấu tổ chức

Có 4 phòng ban trong SCC, bao gồm:

* Phòng đảm bảo chất lượng dịch vụ
* Phòng vận hành
* Phòng giải pháp công nghệ (RnD)
* Phòng hệ thống

**1.2.1 Phòng đảm bảo chất lượng dịch vụ**

Phòng đảm bảo chất lượng dịch vụ bao gồm 3 dịch vụ chính:

* Dịch vụ Giám sát
* Dịch vụ Hỗ trợ kỹ thuật
* Dịch vụ VoIP
* Tư vấn thiết kế và triển khai mô hình dịch vụ
* Kiểm soát, đánh giá chất lượng và hỗ trợ vận hành dịch vụ
* Cải tiến dịch vụ

**1.2.2 Phòng vận hành**

Bao gồm 2 mảng:

* Mảng vận hành có vai trò:
  + Giám sát hạ tầng mạng FPT Telecom
  + Hỗ trợ kỹ thuật xử lý sự cố các dịch vụ do FPT Telecom cung cấp
* Mảng dự án có vai trò:
  + Thực hiện dự án với mục tiêu nâng cao, cải tiến và tối ưu chất lượng dịch vụ

**1.2.3 Phòng giải pháp công nghệ (RnD)**

* Phát triển phần mềm cho SCC và FTEL
* Xây dựng và quản trị hệ thống máy chủ dịch vụ của SCC
* Ứng dụng phân tích dữ liệu và học máy trong dự án của SCC và FTEL

**1.2.4 Phòng hệ thống**

Bao gồm 2 mảng:

* Mảng vận hành có vai trò:
  + Quản lý, vận hành và phát triển tính năng mới hệ thống Voice của FTEL
  + Triển khai mở rộng, nâng cấp hệ thống Voice và thực hiện các dự án của Bộ/cục liên quan đến VoIP
  + Phân tích dữ liệu/ báo cáo tình trạng năng lực hệ thống Voice
* Mảng dự án có vai trò:
  + Nghiên cứu, phát triển công cụ tối ưu hệ thống và các sản phẩm mới tăng trưởng chất lượng dịch vụ VoIP của Ftel
  + Triển khai các dự án cấp công ty liên quan đến hệ thống VoIP và các kênh đa phương tiện

### 1.3 Các đơn vị Ftel khác

**1.3.1 Các đơn vị thuộc khối dịch vụ khách hàng**

* CS: trung tâm chăm sóc khách hàng, có nhiệm vụ hỗ trợ khách hàng tại các phòng giao dịch và qua điệnt hoại
* FPLAY: có nhiệm vụ cung cấp dịch vụ Box FPT
* FSS: có nhiệm vụ cung cấp dịch vụ camera
* FOXPAY: cung cấp dịch vụ ví điện tử

**1.3.2 Các đơn vị thuộc khối kĩ thuật**

* ***INF***: trung tâm phát triển quản lý hạ tầng, có nhiệm vụ chịu trách nhiệm xây dựng và quản lý đài trạm hàng tầng access, …
* ***NOC***: trung tâm điều hành mạng, có nhiệm vụ phụ trách nghiên cứu phát triển quy hoạch toàn bộ mạng core trong nước quốc tế, truyền dẫn và phụ trách toàn bộ Data Center FPT
* ***PMB***: ban quản lý dự án tuyến cáp trục của Ftel, có nhiệm vụ chịu trách nhiệm thiết kế triển khai vận hành đường cáp trục Bắc-Nam, các tuyến trục liên tỉnh các vòng ring trục quốc tế.
* ***PAYTV***: trung tâm dịch vụ truyền hình, có nhiệm vụ triển khai dịch vụ truyền hình FPT
* ***CSOC***: trung tâm bảo mật, có nhiệm vụ phục trách toàn bộ hệ thống bảo mật của Ftel
* ***ISC, CADS***: trung tâm hệ thống thông tin, trung tâm dữ liệu, có nhiệm vụ phụ trách phát triển, vận hành các Tool công cụ và hệ thống dữ liệu

**1.3.3 Các đơn vị thuộc khối khách hàng bên ngoài**

* ***PNC và TIN***: là 2 công ty đối tác của FTS và FTN (Miền Nam và miền Bắc) nhưng vẫn thuộc FTEL quản lý nên gọi là 2 trung tâm quản lý đối tác. Chịu trách nhiệm chính là triển khai, bảo trì cho khách hàng Ftel.
* ***FTI***: công ty TNHH Viễn Thông Quốc tế, có vai trò cung cấp các kênh thuê riêng, truyền số liệu, voice , dịch vụ DC…

# CHƯƠNG 2: 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**

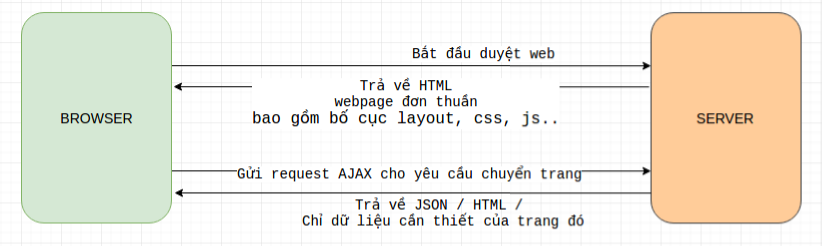
* Cho phép sử dụng từ khóa await bên ngoài async function. Nó hoạt động giống như một hàm async lớn khiến các module khác phải đợi trước khi thực hiện các lệnh bên trong

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

// → temp

# CHƯƠNG 3: TÌM HIỂU VỀ SPA, VIRTUAL DOM VÀ REALDOM

## 3.1 Single Page Application (SPA)

**3.1.1 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:



* 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

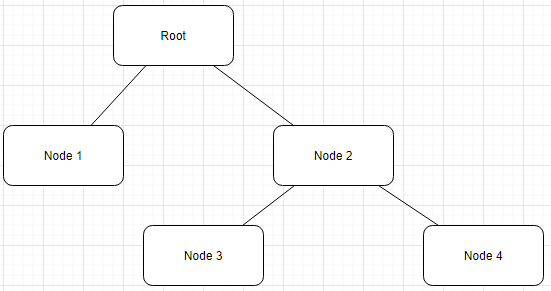
**3.1.4 Nhược điểm của SPA**

* 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 :

****

* 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**

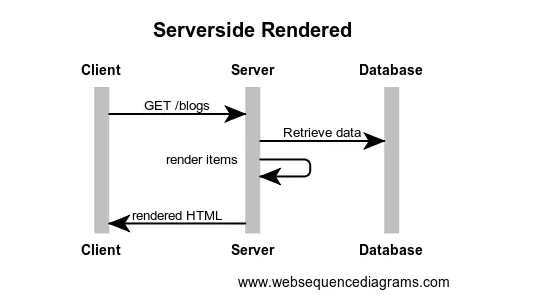
* Virtual DOM chỉ render lại những node bị thay đổi chứ không phải toàn bộ DOM như real DOM.

# CHƯƠNG 4: 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**

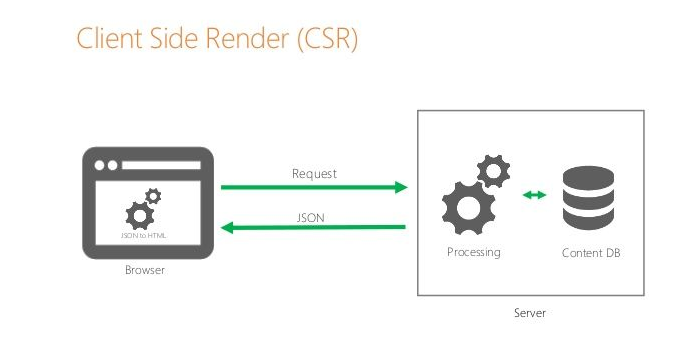
* Đây là cơ chế được sử dụng từ lâu. Các xử lý về logic sẽ được thực hiện ở phía server và client chỉ nhận các kết quả trả về từ phía server này.
* Cụ thể hơn như sau:
  + Khi người dùng truy cập vào một trang web. Ngay lập tức trình duyệt sẽ gửi GET request đến server.
  + Server sau khi nhận request 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.



* 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



* 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());

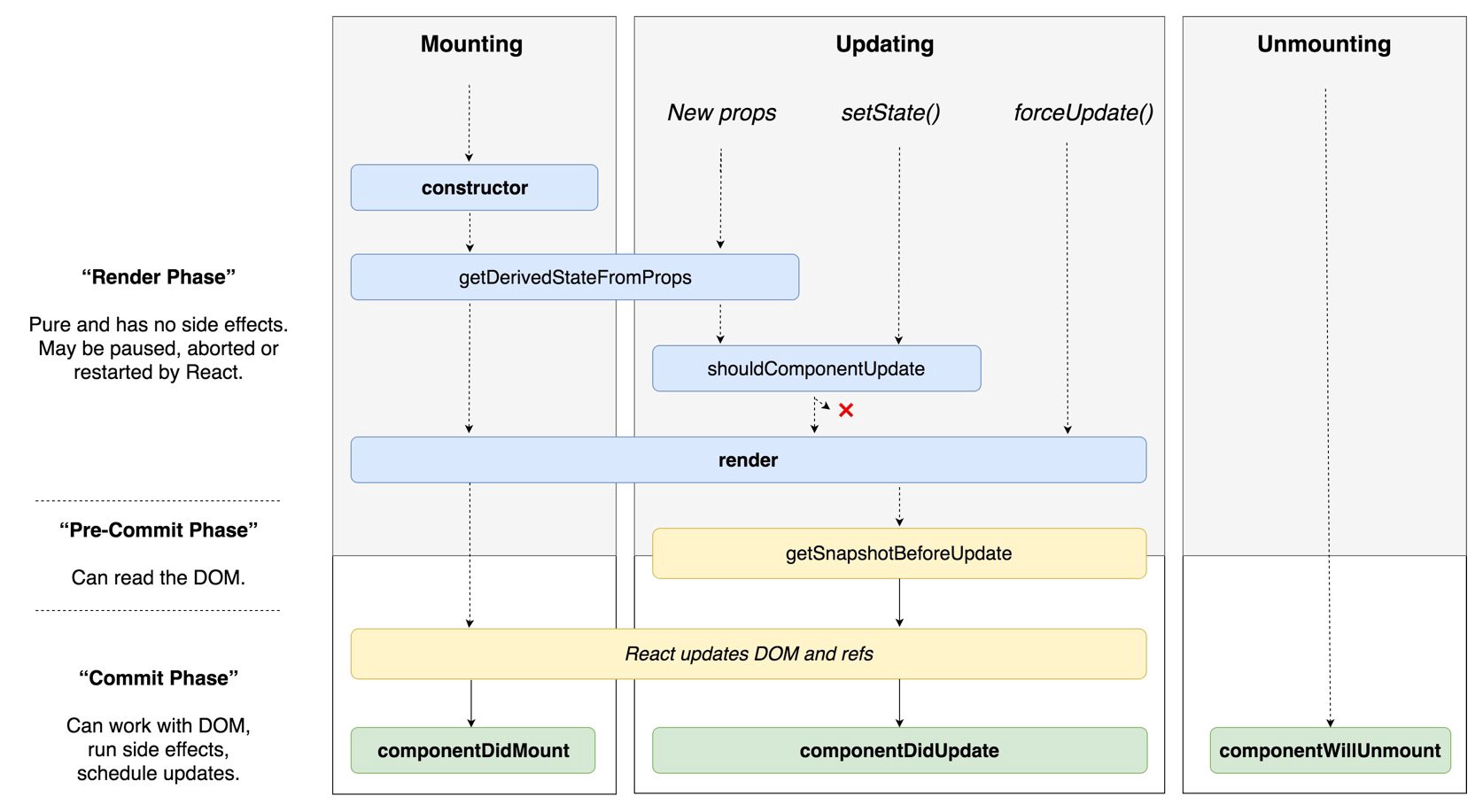
// {value: undefined, done: true},

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

**CHƯƠNG 5: 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*



### 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) {

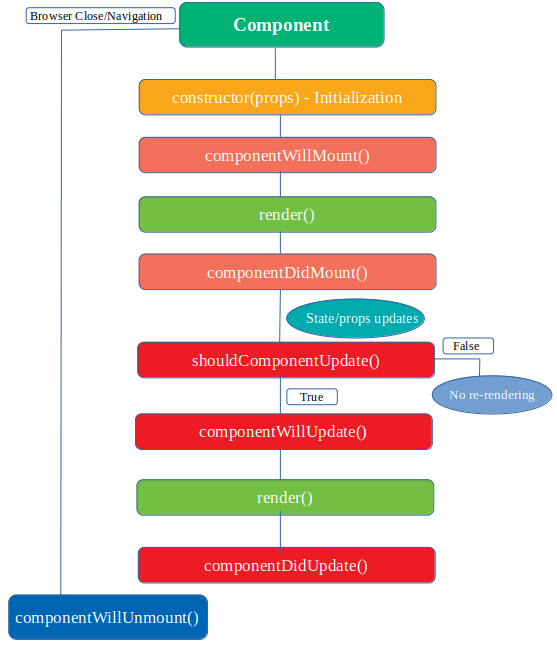
  // You can also log the error to an error reporting service

  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 6 : 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:

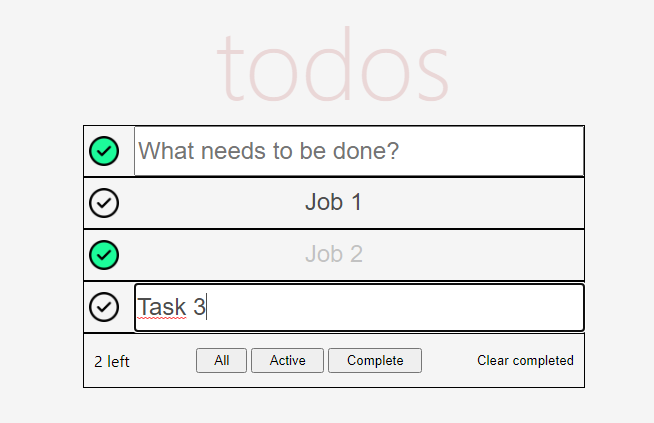
[internSCC/React/todo-list at main · uytran36/internSCC (github.com)](https://github.com/uytran36/internSCC/tree/main/React/todo-list)

## 6.4 Demo

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

Một số hình ảnh:





## 6.5 Mở rộng

Tất cả những component và các logic trong app đều được viết theo class component. Nhưng em muốn mở rộng app này qua sử functional component và sử dụng hook nên em đã sửa lại app như sau:

### 6.5.1 Những thay đổi

Em sẽ thay đổi tất cả class component thành functional component, sau đó sử dụng useState để tạo các state, sử dụng các Arrow function để xử lý các logic cũng như bắt sự kiện trong app. Khi em muốn sử dụng các lifecycle giống như bên class component, em sử dụng useEffect để giải quyết vấn đề đó.

### 6.5.2 Source code

Tất cả source code nằm ở trong link github sau:

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

### 6.5.3 Demo

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

## 6.4 Những hướng đi tiếp theo

* Sử dụng Mock API để tạo data xong dùng app của mình để kết nối và xử lý đến data đó
* Sử dụng các design (Bootstrap, Material Design, Ant Desgin) để làm table và xử lý các sự kiện thêm, xóa, sửa trên table đó
* Tìm hiểu thêm Redux.