# Mục lục

[Mục lục i](#_Toc55233105)

[Danh mục hình vẽ iii](#_Toc55233106)

[Danh mục bảng iv](#_Toc55233107)

[Chương 1 Giới thiệu 1](#_Toc55233108)

[1.1 Công nghệ sử dụng trong dự án quản lý công việc 1](#_Toc55233109)

[1.2 Một số thuật ngữ cần chú ý 1](#_Toc55233110)

[Chương 2 Mô hình phân quyền RBAC và cấu trúc CSDL trong dự án quản lý công việc 3](#_Toc55233111)

[2.1 Mô hình phân quyền RBAC 3](#_Toc55233112)

[2.2 Cơ sở dữ liệu của dự án Quản Lý Công Việc 3](#_Toc55233113)

[2.2.1 Mô hình 3](#_Toc55233114)

[2.2.2 Company 6](#_Toc55233115)

[2.2.3 User 6](#_Toc55233116)

[2.2.4 Role 7](#_Toc55233117)

[2.2.5 UserRole 7](#_Toc55233118)

[2.2.6 RoleType 8](#_Toc55233119)

[2.2.7 Department 10](#_Toc55233120)

[2.2.8 Privilege 10](#_Toc55233121)

[2.2.9 Link 11](#_Toc55233122)

[2.2.10 Component 11](#_Toc55233123)

[2.2.11 Action 12](#_Toc55233124)

[Chương 3 Project quản lý công việc 13](#_Toc55233125)

[3.1 Coding convention 13](#_Toc55233126)

[3.1.1 Đặt tên Rest Services 13](#_Toc55233127)

[3.1.2 MongoDB 15](#_Toc55233128)

[3.1.3 NodeJS 17](#_Toc55233129)

[3.1.4 ReactJS 17](#_Toc55233130)

[3.2 Bảo mật trên website 19](#_Toc55233131)

[3.2.1 Các kỹ thuật được sử dụng trong bảo mật 19](#_Toc55233132)

[3.2.2 Xác thực tài khoản người dùng 21](#_Toc55233133)

[3.2.3 Xác thực quyền của người dùng 22](#_Toc55233134)

[3.3 Cấu trúc thư mục 22](#_Toc55233135)

[3.3.1 Cấu trúc của project 22](#_Toc55233136)

[3.3.2 Cấu trúc thư mục Server 22](#_Toc55233137)

[3.3.3 Cấu trúc thư mục Client 27](#_Toc55233138)

[Chương 4 Cài đặt và cấu hình 42](#_Toc55233139)

[4.1 Cấu hình xác thực truy cập trong MongoDB 42](#_Toc55233140)

[4.2 Cài đặt và cấu hình project 44](#_Toc55233141)

[Chương 5 Hướng dẫn về React 47](#_Toc55233142)

[5.1 Class component 47](#_Toc55233143)

[5.1.1 Hướng dẫn phân biệt và sử dụng getDerivedStateFromProps và shouldComponentUpdate 47](#_Toc55233144)

[5.1.2 Trình tự vòng đời ReactJS 48](#_Toc55233145)

[5.2 Function component 49](#_Toc55233146)

[5.2.1 React.memo 49](#_Toc55233147)

[5.2.2 React Hooks 50](#_Toc55233148)

[Tài liệu tham khảo 63](#_Toc55233149)

# Danh mục hình vẽ

[**Hình 1** Ví dụ về JSON 1](#_Toc44686924)

[**Hình 2** Mô hình CSDL 4](#_Toc44686925)

[**Hình 3** Ví dụ về coding convention trong MongDB 16](#_Toc44686926)

[**Hình 4** JSX đa dòng 18](#_Toc44686927)

# Danh mục bảng

**No table of figures entries found.**

# Giới thiệu

## Công nghệ sử dụng trong dự án quản lý công việc

NodeJS, MongoDB – lập trình backend cho bên server

ReactJS, Redux – lập trình frontend cho bên client.

## Một số thuật ngữ cần chú ý

Database (DB): Nơi chứa các Collection – giống với cơ sở dữ liệu quan hệ thì chứa các bảng.

Collection: Là nhóm của nhiều Document trong MongoDB. Collection có thể được hiểu tương ứng như là 1 bảng trong cơ sở dữ liệu quan hệ (SQL). Các collection không nhất thiết phải định nghĩa các cột, các hàng hay kiểu dữ liệu trước như trong Cơ sở dữ liệu quan hệ vì tất cả thông tin được lưu dưới dạng document.

Document: Là 1 bản ghi thuộc 1 Collection. Các Document lần lượt bao gồm các trường tên và giá trị tương ứng.

JSON: Viết tắt của JavaScript Object Notation. Con người có thể đọc được ở định dạng văn bản đơn giản thể hiện cho các dữ liệu có cấu trúc.

A screenshot of a cell phone

Description automatically generated

**Hình 1** Ví dụ về JSON

User: người dùng

Role: vai trò, chức danh, … của người dùng.

Resource: tài nguyên của hệ thống, có thể là các page của website, các button trên giao diện, các tài liệu biểu mẫu, …

Privilege: đặc quyền của từng Role ứng với từng loại tài nguyên.

Action: các loại hành động tương ứng với tài nguyên.

# Mô hình phân quyền RBAC và cấu trúc CSDL trong dự án quản lý công việc

## Mô hình phân quyền RBAC

RBAC (Role-Based Access Control): là một mô hình phân quyền dựa theo vai trò. Trong đó:

* Mỗi User sẽ có 1 hoặc nhiều Role.
* Mỗi Role sẽ có 1 hoặc nhiều Privilege thực thi các Action tương ứng với một tài nguyên (Resource) nào đó trong hệ thống.
* Tài nguyên có thể là một trang web nào đó, một thành phần giao diện, hoặc một tài nguyên do người dùng tạo ra khi hệ thống hoạt động.

Ví dụ: Ông Nguyễn Văn A có vai trò trong 1 công ty là Trưởng Phòng Kế Hoạch. Vai trò này cho phép ông có quyền truy cập vào **trang** **quản lý mẫu công việc**. Trong khi đó, chị Phạm Thị C, với vai trò là Nhân viên phòng kế hoạch, sẽ không vào được trang quản lý mẫu công việc đó. Ở đây, tài nguyên được xét đến là trang quản lý mẫu công việc.

Tương tự, dù cả hai cùng được vào một trang web nào đó (VD trang xem mẫu công việc), ông A sẽ thấy nút Edit cho từng mẫu công việc, nhưng chị C không thấy nút này. Ở đây, tài nguyên được xét đến là nút Edit.

Một ví dụ khác, với một tài liệu X nào đó, ông A có thể nhìn thấy, trong khi chị C không thể. Trong cùng trang xem danh sách tài liệu, tài liệu X hiển thị cho ông A thấy, và không xuất hiện với chị C. Sau này, khi ông A thôi chức trưởng phòng kế hoạch và do vậy không còn vai trò này nữa, tài liệu X sẽ không hiển thị cho ông A thấy nữa.

## Cơ sở dữ liệu của dự án Quản Lý Công Việc

### Mô hình

Database của hệ thống sẽ bao gồm các Collection chính sau đây:

* Company
* User
* Role
* UserRole
* RoleType
* Department
* Privilege
* Link
* Component
* Action

A close up of a map

Description automatically generated

**Hình 2** Mô hình CSDL

Tham khảo cách mô hình hóa csdl trong MongoDB:

<https://docs.mongodb.com/manual/core/data-model-design/>

Hệ thống quản lý công việc sẽ quản lý cho nhiều công ty cùng một lúc. Cần phải biết được các đối tượng thuộc về công ty nào. Vì vậy, mỗi một user, role, department, resource (link, component, …) sẽ có một thuộc tính là company – để xác định user, role, department, resource đó thuộc về công ty nào.

Giữa Link (1 page) và Component (Thành phần của giao diện) có mối liên hệ One to Many – 1 trang sẽ bao gồm nhiều thành phần (button, form, …) thì mối liên hệ này sẽ được biểu diễn : trong collection Link sẽ có 1 trường là components – định danh là một mảng các Id của component tương ứng với page này.

Giữa User và Role có mối liên hệ là một user có thể có nhiều role và một role có thể là role của 1 hoặc nhiều user. Điều này khá giống với quan hệ n-n (trong cơ sở dữ liệu quan hệ). Tuy nhiên trong MongoDB hiện tại chưa có khái niệm (Many to Many) nên để thể hiện được quan hệ Many to Many như của cơ sở dữ liệu quan hệ thì chúng ta sẽ xây dựng collection trung gian UserRole để lưu giữ mối liên hệ này. Cụ thể, mỗi một bản ghi trong UserRole sẽ là một cặp giá trị (user, role). Trong model User (code phần server) có một trường ảo là roles – sẽ truy xuất tất cả những role mà user này có. Ngược lại với model Role sẽ có một trường ảo là users – sẽ truy xuất tất cả những user có role này.

A screenshot of a cell phone

Description automatically generated

A screenshot of a cell phone

Description automatically generated

Tương tự với mối liên hệ giữa role và các loại tài nguyên cũng là Many to Many (collection trung gian là Privilege) thì sẽ có các trường ảo cho thuộc tính tương ứng để truy xuất dữ liệu giữa hai bên. Ví dụ như 1 role có quyền với nhiều trang (có trường ảo là links), có quyền với nhiều component (có trường ảo là components) như hình mô tả cấu trúc của Role như ở trên. Tương ứng với mỗi một loài nguyên sẽ đi kèm với một action cấp phép cho role có quyền với tài nguyên được phép làm những gì với tài nguyên đó (xem, sửa, xóa, …).

### Company

Lưu trữ thông tin về các công ty sử dụng dịch vụ quản lý công việc của hệ thống

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID của công ty |
| **name** | *String* | Tên đầy đủ của công ty |
| **short\_name** | *String* | Tên viết tắt của công ty |
| **description** | *String* | Mô tả về công ty |

### User

Thông tin về tài khoản của người dùng

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID của người dùng |
| **name** | *String* | tên của người dùng |
| **email** | *String* | địa chỉ email |
| **password** | *String* | mật khẩu |
| **company** | *ObjectId* | công ty mà người dùng làm việc |
| **active** | *Boolean* | trạng thái hoạt động của tài khoản người dùng |
| **status** | *Number* | đếm số lần tối đa nhập sai thông tin tài khoản khi đăng nhập |
| **reset**\_**password**\_**token** | *String* | token xác thực đổi mật khẩu khi quên |
| **token** | *Array String* | Mảng các token lưu trữ phiên đăng nhập của người dùng |

### Role

Các role trong hệ thống

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID của role |
| **name** | *String* | tên của role |
| **company** | *ObjectId* | Công ty có role này |
| **type** | *ObjectId* | Loại role (lưu trong collection RoleType) |
| **parents** | *Array ObjectId* | Là mảng các role là cha của role hiện tại  Role hiện tại, ngoài các quyền của chính nó với một loại tài nguyên nào đó, sẽ kế thừa thêm quyền của những role cha với các tài nguyên khác.  Chỉ cho phép kế thừa trực tiếp, không cho phép kế thừa gián tiếp.  VD: Có 3 Role, Role 2 kế thừa Role 1, và Role 3 kế thừa Role 2 như sau:  Role 1🡨 Role 2🡨 Role 3.  Nếu Role 1 có quyền xem tài liệu X. Role 2 dù không được liên kết với tài liệu X (không có quyền gì), nhưng vì kế thừa trực tiếp Role 1, Role 2 vẫn có quyền xem tài liệu X. Giả sử Role 3 không có liên kết gì tới tài liệu X, do Role 3 không kế thừa trực tiếp Role 1, nên nó sẽ không có quyền xem tài liệu X |

### UserRole

Chứa thông tin về mối liên hệ Many to Many( nhiều-nhiều) giữa User và Role. Một mối liên hệ nhiều – nhiều giữa nhiều User với nhiều Role sẽ bao gồm nhiều cặp thông tin bao gồm Id của user sẽ tương ứng với Id của role nào.

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **userId** | *ObjectId* | ID của user |
| **roleId** | *ObjectId* | ID của role |

### RoleType

Các loại role có trong hệ thống

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID loại role |
| **name** | *String* | Tên loại ( kiểu ) role |

Trong hệ thống được định nghĩa sẵn 3 loại role đó là: RoleAbstract, RoleChucDanh và RoleTuTao.

#### RoleAbstract

Là loại role được thiết kế riêng để phân quyền cho giao diện frontend của hệ thống. *Tất cả các role thuộc loại RoleAbstract là không thể xóa*. Mặc định sẽ có 5 role thuộc loại RoleAbstract đó là:

* **System Admin** – là role quản trị viên của toàn bộ hệ thống. Hệ thống có thể phục vụ đồng thời cho nhiều công ty khác nhau. **System Admin** không không tham gia vào hoạt động nghiệp vụ bất kì công ty nào. **System Admin** chỉ có nhiệm vụ khởi tạo các dữ liệu ban đầu cho một công ty mới (khởi tạo dữ liệu về công ty, và tạo tài khoản SuperAdmin cho công ty đó, …)
* **Super Admin**: là role quản trị cấp cao nhất của 1 công ty. Trong công ty đó, chỉ có duy nhất 1 tài khoản user X được gắn role là **super admin**. Tài khoản X này được tạo khi **System Admin** tạo mới 1 công ty. Không thể gán role **Super Admin** cho bất kỳ tài khoản nào khác ngoài tài khoản X này, cũng không thể bỏ role này khỏi tài khoản X. Tài khoản X này cũng không thể bị xóa đi. Role **Super Admin** kế thừa **Admin**, nên sẽ có mọi quyền của role **Admin**.
* **Admin**: role quản trị của công ty. **Super Admin** hoặc một **Admin** bất kỳ có thể gán role này cho tài khoản Y bất kỳ trong công ty đó. Không thể xóa role **Admin**, nhưng có thể xóa (deactivate) tài khoản bất kỳ có role này, cũng như có thể thu hồi role **Admin** cho một tài khoản bất kỳ.
* **Dean**: Trưởng một đơn vị bất kỳ trong công ty, sẽ vào được nhiều trang thông tin hơn, cũng có nhiều menu chức năng hiển thị hơn so với **Vice Dean**, **Employee**. Việc phân quyền này có thể linh động cho từng công ty, được quản lý bởi **Admin**.
* **Vice** **Dean**: phó trưởng một đơn vị trong công ty
* **Employee**: nhân viên của đơn vị bất kỳ trong công ty

#### RoleChucDanh

Là loại role đại diện cho một chức vụ, chức danh của người dùng trong các phòng, ban, bộ phận, khối (được gọi tổng quát là một **đơn vị**) trong một công ty. Mỗi một **RoleChucDanh** phải luôn đi kèm với một **Department** (*đơn vị, phòng ban, …*) nào đó. Với một **RoleChucDanh,** người dùng sẽ có quyền tương ứng với một tài nguyên mà trong phạm vi **Department** đó có thể thực hiện*.* Ví dụ, một tài liệu nào đó sẽ chỉ được xem bởi nhân viên phòng Hành chính và chỉ được edit bởi trưởng phòng phòng Hành chính. Nhân viên phòng khác hoàn toàn không biết đến sự tồn tại của tài liệu này.

Khi tạo một đơn vị mới, ví dụ phòng kinh doanh, hệ thống tự động tạo 3 role chức danh với quan hệ kế thừa như sau: *Chức danh trưởng phòng kinh doanh* 🡨 *Chức danh phó phòng kinh doanh* 🡨 *Chức danh nhân viên phòng kinh doanh*. Người dùng được nhập vào tên cụ thể của 3 loại chức danh này. Trưởng phòng kinh doanh sẽ kế thừa 3 role là: **Dean**, phó phòng kinh doanh, nhân viên phòng kinh doanh. Tương tự, phó phòng kinh doanh kế thừa **Vice Dean**, nhân viên phòng kinh doanh. Cuối cùng, nhân viên phòng kinh doanh chỉ kế thừa role **Employee**. Điều này cho phép Trưởng phòng kinh doanh xem được mọi tài nguyên mà phó phòng hoặc nhân viên phòng kinh doanh được xem. Role trưởng phòng kinh doanh không kế thừa role Employee, vì có những chức năng (như khai báo KPI cá nhân) chỉ nhân viên mới được thực hiện.

Ngoài ra, cần lưu ý trường hợp sau. Giả sử trong cơ cấu tổ chức của 1 công ty, Phòng kinh doanh trực thuộc trong Khối Văn phòng thì người đứng đầu phòng kinh doanh sẽ có 2 role, một role là **Trưởng phòng kinh doanh,** và role kia là **nhân viên Khối Văn phòng**.

#### RoleTuTao

Là role được tạo thêm nhằm mục đích phân quyền một cách cụ thể với một tài nguyên nào đó hoặc áp dụng cho một nhóm người cụ thể nào đó. Những Role thuộc vào nhóm **RoleTuTao** không nhiết thiết phải đi kèm hay có ràng buộc với một đơn vị, phòng ban nào mà có thể hoàn toàn độc lập trong hệ thống.

### Department

Lưu thông tin về các đơn vị/phòng ban của 1 công ty. Mỗi một đơn vị sẽ có 3 RoleChucDanh được tạo ra, như mô tả ở phần các loại Role

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID của đơn vị |
| **name** | *String* | Tên đơn vị |
| **company** | *String* | Công ty chứa đơn vị |
| **description** | *String* | mô tả về đơn vị |
| **dean** | *ObjectId* | Trưởng đơn vị |
| **vice\_dean** | *ObjectId* | Phó đơn vị |
| **employee** | *ObjectId* | Nhân viên trong đơn vị |
| **parent** | *ObjectId* | Đơn vị cha của đơn vị hiện tại |

### Privilege

Chứa phân quyền của từng role ứng với một tài nguyên cụ thể

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **resourceId** | *ObjectId* | ID của tài nguyên |
| **resourceType** | *String* | Loại tài nguyên. Hiện tại có nhưng loại tài nguyên là Link, Component. |
| **roleId** | *ObjectId* | Role tương ứng có quyền với tài nguyên |
| **action** | *ObjectId* | Đây là thuộc tính được thêm vào để xác định rõ hơn mối quan hệ giữa role (roleChucDanh, roleTuTao) với cái tài nguyên không phải giao diện (link, component) như tài liệu, biểu mẫu, công văn, ...  Khác với tài nguyên link, component, các loại tài nguyên khác (tài liệu, biểu mẫu, công việc, ...) cần rõ ràng hơn về các hành động role được phép thực hiện như là: xem, thêm, sửa, xóa.  Đối với RoleAbstract, khi liên kết với tài nguyên giao diện (link, component) giá trị mặc định cho trường này sẽ là NULL. Điều này không phải có nghĩa là RoleAbstract không có quyền với tài nguyên giao diện mà là không cần chia rõ ràng xem, thêm, sửa, xóa với tài nguyên giao diện. Giao diện là do lập trình viên lập trình, người dùng cuối không thể xóa/sửa/thêm được. VD với một link, Admin sẽ được truy cập để mở ra trang tương ứng. Như vậy là đủ (gần như chỉ cần quyền xem) |

Bản chất giữa Role và các loại tài nguyên (Resource) có mối quan hệ nhiều – nhiều. Collection Privilege này có tính chất như một bảng trung gian có chức năng giống với collection UserRole ở trên. Privilege sẽ đảm nhận nhiệm vụ lưu các cặp dữ liệu Role – Resource (tài nguyên ở đây có thể là các trang - Link, các button trên giao diện của người dùng - Component, các biểu mẫu công việc – TaskTemplate, …)

### Link

Lưu trữ thông tin của các page của website hệ thống quản lý công việc.

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID trang page |
| **url** | *String* | URL của page |
| **company** | *ObjectId* | Công ty chứa page này |
| **description** | *String* | Mô tả |
| **components** | *Array ObjectId* | Các thành phần UI, UX được phân quyền theo từng role trên page này. |

### Component

Những thành phần UI, UX trên từng page.

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID trang page |
| **name** | *String* | Mô tả |
| **company** | *ObjectId* | Công ty chứa page này |
| **description** | *String* | Mô tả |

### Action

Các hành động tương ứng của từng role với mỗi loại tài nguyên khác nhau. Ví dụ Trưởng Phòng Kế Hoạch sẽ có thể action (*xem, sửa, xóa, kích hoạt*,…) 1 loại tài liệu biểu mẫu công việc nào đó.

|  |  |  |
| --- | --- | --- |
| Thuộc tính | Kiểu giá trị | Mô tả |
| **\_id** | *ObjectId* | ID trang page |
| **name** | *String* | Tên của hành động |
| **see** | *Boolean* | Cho phép nhìn thấy tài nguyên |
| **open** | *Boolean* | Cho phép mở tài nguyên |
| **edit** | *Boolean* | Cho phép chỉnh sửa tài nguyên |
| **delete** | *Boolean* | Cho phép xóa tài nguyên |
| **enable** | *Boolean* | Cho phép kích hoạt hoạt động cho tài nguyên |
| **disable** | *Boolean* | Cho phép khóa tài nguyên |

# Project quản lý công việc

## Coding convention

### Đặt tên Rest Services

Xem kỹ các ví dụ sau:

|  |  |
| --- | --- |
| Chức năng service | Ví dụ |
| Lấy ra tất cả sản phẩm | GET http://www.example.com/products |
| Tạo mới một sản phẩm | POST http://www.example.com/products |
| Đọc, cập nhật, xóa sản phẩm 66432 | GET|PATCH|DELETE http://www.example.com/products/66432 |
| Cập nhật dùng PUT hay PATCH | Cập nhật có thể dùng PUT/PATCH. PUT chứa toàn bộ thông tin về entity, thay thế hoàn toàn 1 entity. PATCH cung cấp 1 phần thông tin về entity và chỉ sửa phần thông tin được cung cấp |
| Dùng / để tạo cấu trúc phân cấp | Ví dụ tạo order cho 1 customer:  POST http://www.example.com/customers/33245/orders |
| Luôn dùng dạng số nhiều | GET http://www.example.com/customers/33245/orders/8769/lineitems/1 |
| Dùng dạng chữ thường và dấu gạch dưới | /users/{id}/pending-orders thay vì /users/{id}/Pending\_Orders |
| Không kết thúc bằng dấu gạch / | /users/{id}/pending-orders thay vì /users/{id}/pending-orders/ |
| Sử dụng query string param để lấy dữ liệu theo điều kiện lọc. | Không tạo API mới mà dùng query parameters  http://api.example.com/device-management/managed-devices  http://api.example.com/device-management/managed-devices?region=USA  http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ  http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ&sort=installation-date |
| Các API xử lý khác (controller) | Dùng POST  Controller API hoạt động tương tự các hàm, có tham số. Dùng động từ với các controller API này  http://api.example.com/cart-management/users/{id}/cart/checkout  http://api.example.com/song-management/users/{id}/playlist/play |
| Idempotent và safe request | GET, PUT và DELETE bắt buộc phải là idempotent request  GET là safe request  Một request được xem là idempotent nếu dù gọi bao nhiêu lần, nó vẫn trả về kết quả như nhau. Ví dụ idempotent request: GET http://www.example.com/products  Một request được xem là safe nếu sau vô số lần gọi, nó vẫn không làm thay đổi resource mà nó đang truy cập đến. Ví dụ GET http://www.example.com/products/66432 là safe request và PATCH /http://www.example.com/products/66432 không là safe request. |

**Dùng tiền tố module + tên bảng/collection tác động + các quy tắc ở trên**

VD:

PATCH /auth/profile/:id/change-information 🡪 PATCH auth/users/:userId

POST /auth/login 🡪 Okie

GET /auth/logout 🡪 POST /auth/logout (do khi logout, xóa token, làm thay đổi dữ liệu, không thể dùng GET, vì GET là Idempotent và safe request)

GET /links-default-management 🡪 GET /system-admin/system-links

GET /department/departments-that-user-is-dean/:id 🡪 GET /super-admin/organizational-units?deanOfUnit=id

### MongoDB

#### Quy tắc chung

Sử dụng 4 space để lùi đầu dòng, điều này áp dụng cho tất cả những mã code của MongoDB và cho tất cả các đối tượng sử dụng mongoDB (Queries, documents)

Luôn có một khoảng trắng sau dấu hai chấm

Dấu phẩy ở cuối

Nếu chia thành phần của mỗi đối tượng hoặc mảng thành nhiều dòng khác nhau thì hãy chia mỗi thành phần thành một dòng, dấu ngoặc nhọn đóng phải đi theo sau thành phần cuối cùng (ngoại trừ tập hợp)

Tất cả các tên đều không được chứa khoảng trắng

Không có dấu gạch dưới ở giữa tên (database, collection, fields).

Tên của các collection, variables, properties và function sử dụng quy tắc lowerCamelCase. Các biến kí tự đơn hoặc viết tắt không phổ biến thì không được sử dụng.

A screen shot of a monitor

Description automatically generated

**Hình 3** Ví dụ về coding convention trong MongDB

#### Đặt tên Database

Tên cơ sở dữ liệu đặt theo tên của dự án

Sử dụng UpperCamelCase

Tên cơ sở dữ liệu không được dài quá (ít hơn 64 bytes)

Tên cơ sở dữ liệu phải luôn phân biệt chữ hoa chữ thường

Tên cơ sở dữ liệu không được chứa các ký tự đặc biệt “/, \, ., “, \*, <, >, :, |, ?, $”. Đồng thời cũng không được chứa các khoảng trắng hoặc chuỗi rỗng

#### Đặt tên Collection

Sử dụng UpperCamelCase

Tên collection không được chứa khoảng trắng hoặc là chuỗi rỗng

#### Đặt tên Field

Sử dụng LowerCase

Không sử dụng dấu gạch dưới trong tên trường ngoại trừ id

Không được sử dụng dấu chấm hoặc chuỗi rỗng và không bắt đầu bằng $

#### Functions

Nếu chia các thành phần thành nhiều dòng thì mỗi thành phần sẽ để một dòng

Sử dụng dấu tab để lùi đầu dòng với mỗi thành phần

### NodeJS

#### Quy tắc chung

Sử dụng 4 space để lùi đầu dòng và không trộn lẫn tab với khoảng trống khi căn lề

Không để khoảng trắng cuối mỗi tệp .js

Tối đa 80 kí tự trên một dòng

Sử dụng dấu ngoặc đơn, ngoại trừ khi viết JSON

Ví dụ đúng: var foo = ‘bar’;

Dấu ngoặc mở trên cùng một dòng với statement

Kết thúc câu lệnh khai báo với dấu chấm phẩy trên cùng một dòng.

#### Đặt tên

Tên biến và hàm bao gồm nhiều từ thì kí tự đầu mỗi từ viết hoa trừ từ đầu tiên.

( VD: departmentName )

Tên lớp viết hoa chữ cái đầu tiên của các từ. Các chữ còn lại viết thường.

( VD: ViceDean,…)

Tên của hằng thì tất cả chữ cái viết hoa (VD:LOGGED,AUTHENTICATION,…)

### ReactJS

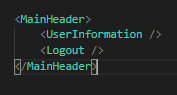
#### Quy tắc chung

Chia Component nhỏ nhất có thể

Mỗi Component lưu riêng vào 1 file (ưu tiên dạng .jsx với các file giao diện).

Sử dụng Composition để mở rộng chức năng (thông qua props thì một component cha có thể render ra một hoặc nhiều componet con)

JSX đa dòng: Mỗi một component nên được để riêng trên một dòng. Trước khi viết thụt lùi một dấu Tab.



**Hình 4** JSX đa dòng

#### Đặt tên

**Tên tệp**: Sử dụng lowerCamelCase cho tên tệp

**Tên tham chiếu**: Sử dụng UpperCamelCase cho các thành phần React và lowerCamelCase cho các thể hiện của chúng

Ví dụ:

import reservationCard from './ReservationCard'; // sai

import ReservationCard from './ReservationCard'; // đúng

const ReservationItem = <ReservationCard />; // sai

const reservationItem = <ReservationCard />; // đúng

**Tên component**:

Tên component phải duy nhất và thể hiện được vai trò của nó

Ví dụ: SideBar, ChatConversationName, …

Tên của component được đặt tên theo đường dẫn

Ví dụ: component/User/List -> UserList

**Tên props**: Tránh dùng tên prop components Dom cho các mục đích khác

Ví dụ:

Sai: <MyComponent style=”fancy”/>

Đúng: <MyComponent variant=”fancy”/>

## Bảo mật trên website

### Các kỹ thuật được sử dụng trong bảo mật

#### JSON Web Token

JSON Web Token (JWT) là 1 tiêu chuẩn mở (RFC 7519) định nghĩa cách thức truyền tin an toàn giữa các thành viên bằng 1 đối tượng JSON. Thông tin này có thể được xác thực và đánh dấu tin cậy nhờ vào "chữ ký" của nó. Phần chữ ký của JWT sẽ được mã hóa lại bằng HMAC hoặc RSA.

Cấu trúc của 1 JWT gồm 3 phần và được ngăn cách với nhau bởi dấu chấm (.), ví dụ: **xxxxx.yyyyy.zzzzzz**

Cụ thể là :

* Header: dùng để khai báo kiểu chữ ký và thuật toán mã hóa.
* Payload: nơi chứa các nội dung của thông tin (ví dụ: email, tên, thông công ty người dùng, browser finger,…)
* Signature (chữ ký)

A screenshot of a computer screen

Description automatically generated

Ứng dụng vào trong project, sau kiểm tra thông tin đăng nhập của người dùng là chính xác, chúng ta sẽ tạo ra JWT với cách thức tạo là sử dụng package jwt trong NodeJS với hai tham số truyền vào là :

* Tham số thứ nhất: các thông tin liên quan đến người dùng (tên, email, công ty, fingerprint)
* Tham số thứ hai là TOKEN\_SECRET – đây là một chuỗi bí mật do người lập trình tự tạo ra để thực hiện quá trình mã hóa và giải mã JWT để lấy thông tin.

Sau khi tạo xong JWT chúng ta sẽ gửi đến cho người dùng.

#### FingerPrint

Đây là thông tin cá nhân của trình duyệt đang sử dụng được mã hoá thành 1 chuỗi. Các thông tin này cơ bản bao gồm:

* UserAgent
* Language
* Color Depth
* Screen Resolution
* Timezone
* Has session storage or not
* Has local storage or not
* Has indexed DB
* Has IE specific ‘AddBehavior’
* Has open DB
* CPU class
* Platform
* DoNotTrack or not
* Full list of installed fonts (maintaining their order, which increases the entropy), implemented with Flash.
* A list of installed fonts, detected with JS/CSS (side-channel technique) - can detect up to 500 installed fonts without flash
* Canvas fingerprinting
* WebGL fingerprinting
* Plugins (IE included)
* Is AdBlock installed or not
* Has the user tampered with its languages 1
* Has the user tampered with its screen resolution 1
* Has the user tampered with its OS 1
* Has the user tampered with its browser
* Touch screen detection and capabilities
* Pixel Ratio

Và có thể chứa thêm một vài giá trị khác.

Mỗi một trình duyệt trên từng máy tính khác nhau sẽ có một fingerprint khác nhau -> chính vì thế đây sẽ là yếu tố để xác minh được rằng JWT có được gửi từ cùng một người hay không.

Các request gửi tới server ngoài việc đính kèm token thì sẽ phải đính kèm luôn fingerprint của trình duyệt hiện tại. Khi server nhận được request khi login thì server sẽ cập nhật thông tin fingerprint trong tài khoản user và sau đó dùng 2 thông số là token + fingerprint để định danh user thay vì chỉ token thôi. Mặc dù các thông số trên rất có thể trùng nhau cho 1 số trường hợp nhưng ta đã hạn chế được rất nhiều rủi ro bị giả mạo request.

### Xác thực tài khoản người dùng

Trước khi làm việc trên website người dùng sẽ phải đăng nhập vào hệ thống bằng email và password đã được admin của hệ thống cấp phát được gửi đến cho người dùng qua email. Khi người yêu cầu đăng nhập vào hệ thống website sẽ tự động đính kèm với dữ liệu nhập vào của người dùng một browser-finger sau đó gửi đến server.

Server sau nhận được request yêu cầu đăng nhập từ người dùng sẽ lấy các thông tin email và password để kiểm tra trong cơ sở dữ liệu. Nếu như thông tin email và password là hợp lệ server sẽ mã hóa kèm với thông tin của người dùng cộng với browser-finger nhận được từ client thành một JWT (JSON WEB TOKEN) sau đó lưu vào trong cơ sở dữ liệu (trường thông tin những token JWT đăng nhập thành công) của người dùng và trả về JWT đó cho người dùng. Từ đó mỗi khi client gửi request của mình có đính kèm JWT trong header để xác thực thì server sẽ kiểm tra JWT này có đúng là của người dùng ngày hay không. Nếu đúng thì cho yêu cầu được hoạt động. Còn về phía bên người dùng (client, website) nhận được dữ liệu trả về là JWT và lưu nó vào trong localstorage của trình duyệt của mình. Lúc này người dùng đã đăng nhập thành công và có thể làm việc bình thường trên hệ thống. Thông tin xác thực người dùng được lưu trữ trong JWT.

Một vấn đề đặt ra là nếu có một người nào đó lấy cắp được JWT này?

Giải pháp hiện tại là mỗi một request của người dùng gửi lên server thì trước khi gửi website sẽ đính kèm với đó là 1 fingerprint vào header của request. Server sẽ giải mã JWT được gửi lên và lấy được giá trị fingerprint được mã hóa trong đó để đối chiếu với fingerprint được gửi kèm theo vói request. Nếu như hai giá trị fingerprint này giống nhau tức là request này được gửi từ đúng chủ nhân của nó (đúng với trình duyệt trên máy tính của người dùng). Trường hợp nếu hai giá trị fingerprint này khác nhau có nghĩa là JWT này đã được gửi lên từ một trình duyệt của máy tính khác (tức là không phải người đã tạo ra JWT này) bởi vì mỗi một trình duyệt trên các máy tính khác nhau sẽ có fingerprint khác nhau. Vì thế sau khi server phát hiện JWT bị người khác giả mạo thì server sẽ không cho phép request giả mạo đó được thực hiện và gửi về một mã lỗi ACCESS\_DENIED – truy câp trái phép và sẽ chuyển hướng đến trang đăng nhập để yêu cầu xác thực lại

### Xác thực quyền của người dùng

Trong quá trình người dùng làm việc trên hệ thống, người dùng sẽ có một vai trò cụ thể nào đó tại thời điểm làm việc. Giá trị này được gọi là currentRole của người dùng – có nghĩa là tại thời điểm này người dùng đang làm việc với vai trò là gì (ví dụ: Admin, Trưởng Phòng Kinh Doanh, …). Thông tin về currentRole của người dùng được lưu trữ trong localstorage và ứng với mỗi một request website cũng sẽ tự động đính kèm currentRole vào trong header của request mà người dùng gửi lên server. Server sẽ sau khi thực hiện việc kiểm tra **Xác thực tài khoản người dùng** thì sẽ tiếp tục kiểm tra xem là currentRole này có đúng là thuộc về người dùng này hay không. Nếu như thông tin là chính xác thì request của người dùng sẽ được thực hiện còn nếu thông tin là sai thì request sẽ không được thực hiện. Điều này sẽ giải quyết được vấn đề là một tài khoản chỉ có vai trò là một Nhân viên trong công ty nhưng bằng một cách nào đó anh ta lấy được currentRole là Admin hệ thống và thực hiện hành vi thay thế currentRole trên máy của anh ta thành currentRole của admin. Tuy nhiên do server có kiểm tra lại và phát hiện là anh ta không có vai trò là Admin nên yêu cầu của anh ta với vai trò là Admin sẽ không được thực hiện.

## Cấu trúc thư mục

### Cấu trúc của project

Cấu trúc của project Quản lý công việc bao gồm hai phần chính:

* Server: backend
* Client: frontend

### Cấu trúc thư mục Server

#### Tổng quan

Trong thư mục server có chứa mã nguồn cho phần backend của hệ thống, các thành phần trong thư mục server bao gồm:

**logs**: chứa file ghi lại lịch sử tương tác giữa client và server.

**middleware**: xác thực người dùng từ request và kiểm tra, nếu thông tin là chính xác thì sẽ chuyển tiếp cho server thực hiện yêu cầu mà client gửi đến. Ngược lại nếu xác thực người dùng không hợp lệ thông báo lỗi yêu cầu của client không được server xử lý và trả về thông báo lỗi cho client.

**models**: định nghĩa các collection cho cấu trúc dữ liệu của hệ thống.

**seed**: chứa các file tạo dữ liệu gốc khi build hệ thống, hoặc lập trình viên có thể khởi tạo các dữ liệu test tại đây.

**modules**: chứa các file module – tương ứng với từng chức năng, dịch vụ của hệ thống. Mỗi một module khi được xây dựng sẽ nằm trong thư mục này. Trong thư mục modules có chứa sẵn 1 thư mục con là “ *\_sample-module “* – đây là một thư mục module mẫu. Trong “\_sample-module “ bao gồm 4 file chính đó là:

* README.txt: file mô tả về chức năng của module, cấu trúc của module và sự liên quan của module với các module khác.
* route.js: định nghĩa các đường định tuyến, các API cho từng dịch vụ của hệ thống mà người dùng (client sẽ gọi để yêu cầu sử dụng dịch vụ. Có thể lồng thêm middleware nhằm thực hiện nhiệm vụ xác thực người dùng trước khi request được chuyển tiếp sang cho controller.
* controller.js: nhận các request của người dùng và gọi đến các dịch vụ (service) để xử lý dữ liệu và trả về (response) cho người dùng (client).
* service.js: chứa các service (dịch vụ) – nhận dữ liệu đầu vào, xử lý yêu cầu và đưa ra kết quả đầu ra. Mỗi một hàm trong trong service chỉ nên thực hiện một chức năng duy nhất.
* Ví dụ: trong service của module user có hàm getById – lấy thông tin user theo Id : thì hàm này chỉ nên thực hiện một nhiệm vụ duy nhất là truy xuất thông tin về user theo Id.

**index.js**: file chạy của server.

**package**.**json**: chứa tên các thư viện cần được cài đặt trong project.

#### Các bước tạo một module mới trong server.

**Bước 1**: Tạo một thư mục cho module cần xây dựng trong thư mục server/modules. Cấu trúc của module sẽ giống với module mẫu \_sample-module. Ngoài ra trong quá trình lập trình có thể bổ sung thêm các file khác (ngoài 4 file mẫu trong thư mục \_sample-module) . Ví dụ: validation – nhằm mục đích kiểm tra dữ liệu đầu vào do client người đến trước khi cho xử lý. Lưu ý rằng với những module có khối lượng lớn thì ta sẽ chia nhỏ thành những module con và những module con này sẽ có cấu trúc giống với thư mục \_sample-module.

**Bước 2**: Viết mã nguồn cho file service.js để xử lý yêu cầu dịch vụ trong module. Định nghĩa rõ ràng các tham số đầu vào cho mỗi một chức năng trong file service.

**Bước 3**: Viết mã nguồn cho file controller.js . Tại đây controller nhận dữ liệu đầu vào qua request (req) được gửi đến từ bên client và truyền đến cho service để thực hiện xử lý dữ liệu. Controller có thể gọi đến các dịch vụ của chính module hiện tại hoặc có thể gọi thêm các service từ các module khác. Cuối cùng là trả về dữ liệu cho người dùng qua response (res).

**Bước 4**: Viết mã nguồn cho file route.js. Tạo ra các API tương ứng với từng chức năng của module.

**Bước 5**: Import route của module vừa xây dựng vào file chạy (index.js) của bên server.

#### Ví dụ

Cần xây dựng chức năng lấy tất cả thông tin về các user trong hệ thống trong module quản lý user .

Vì module quản lý user là một module con trong module cha là quản lý hệ thống của SuperAdmin của 1 công ty 🡪 tạo cấu trúc thư mục như hình:

A screenshot of a video game

Description automatically generated

Tiếp theo, trong file user.service.js tạo hàm get() – lấy thông tin của tất cả user trong công ty được chọn. Hàm get() nhận đầu vào là Id của công ty được chọn và search trong collection User tất cả những user có company bằng Id của công ty được truyền vào. Cụ thể: find() sẽ tìm kiếm tất cả nhưng document trong collection User với điều kiện là company là tham số nhận được, select() sẽ tùy chọn những thuộc tính của user sẽ được lấy ra trong quá trình truy vấn, populate – truy vấn tất cả các roles mà user có thông qua trường ảo roles được thiết lập mối quan hệ Many to Many giữa collection User và Role, cộng với đó là thông về thông tin mà user đó đang làm việc. Kết quả trả về là một mảng những user.

A screenshot of a computer screen

Description automatically generated

Tiếp theo là tạo controller trong file user.controller.js gọi đến service get – lấy thông tin tất cả user trong công ty. Trong khối try-catch thì try sẽ làm nhiệm vụ gọi service để xử lý dữ liệu và trả về cho client với mã là *200*. Nếu phát hiện có lỗi khối catch sẽ bắt lỗi đó và trả về cho client với mã lỗi là 400.

A screenshot of a computer screen

Description automatically generated

Tạo API cho chức năng lấy tất cả thông tin user trong file user.route.js. Bởi vì nếu như muốn gọi đến API truy vấn thông tin của tất cả user trong công ty thì bắt buộc phải có một middleware để check xem yêu cầu này có được gửi bởi một tài khoản đã được xác thực – thông qua middleware auth.

A screenshot of a computer screen

Description automatically generated

Cuối cùng là import route của module quản lý user vào file index.js

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer screen

Description automatically generated

Để test dữ liệu nhận được khi gọi API , ta sử dùng phần mềm POSTMAN để gọi đến API đó và kết quả nhận được sẽ là một mảng các users lưu theo cấu trúc JSON hoặc trả về thông báo lỗi nếu có lỗi xảy ra.

Như vậy là chúng ta đã viết xong 1 API – lấy thông tin tất cả user của một công ty.

### Cấu trúc thư mục Client

#### Tổng quan

Thư mục client chứa mã nguồn frontend của hệ thống, cấu trúc sẽ giống như hình bên dưới đây:

A picture containing screenshot

Description automatically generated

Cấu trúc của project về cơ bản giống với frame của 1 project ReactJS. Tuy nhiên có sự khác biệt đó là ở trong thư mục src- cấu trúc mã nguồn phân chia theo từng module tương ứng với các module của bên server. Tổ chức và xử lý dữ liệu nhận về từ server được xử lý bằng Redux – trong thư mục src.Hai phần quan trọng nhất là : public và src.

**public**: chứa các file css,js, image,vv… chung cho cả project.

**src**: chứa mã nguồn cho phần giao diện. Trong đó cấu trúc của thư mục src bao gồm:

* **common-components**: chứa các component được tái sử dụng trên nhiều modules ( ví dụ như thanh tìm kiếm, form, thẻ input…)
* **lang**: phần chuyển đổi ngôn ngữ cho project gồm các file định nghĩa sẵn các loại ngôn ngữ ( ví dụ : tiếng việt và tiếng anh).
* **layouts**: chứa các component phần layout – bố cục trên 1 trang web, gồm : Header, SideBar, Content, Footer.
* **modules**: chứa các module tương ứng với bên server.Mỗi module sẽ gồm hai phần chính :
  + **components**: giao diện cho module chứa các file .jsx và có thể các file css,js riêng của module.
  + **redux**: xử lý lời gọi của client và dữ liệu nhận về từ server.
    - action.js – thực hiện lời gọi đến các hành động mà người dùng yêu cầu
    - constants.js – định nghĩa tên cho các hành động
    - reducer.js – xử lý dữ liệu sau khi xử lý và cập nhật vào trong store
    - service.js – xử lý dữ liệu tương ứng với hành động mà người dùng yêu cầu (gọi API , gửi request, nhận dữ liệu trả về từ server và xử lý).
* **react-routes:** định nghĩa các route ứng với từng trang của website
* **redux**: là nơi tập hợp state chung , kho store của redux.
* **env.example.js:** file mẫu định nghĩa cấu hình cho project. Sau khi clone source từ trên github về ta sẽ tạo ra file **env.js** theo cấu trúc mẫu như file env.example.js để thiết lập các cấu hình cho client.

#### Ví dụ

Ở phần server chúng ta đã xây dựng một API cho chức năng liệt kê tất cả các user của một công ty. Vậy nên ở phần này chúng ta sẽ làm phần giao diện và gọi API cho chức năng đó ở bên client.

Trước khi bắt tay vào code chức năng hiển thị tất cả user trong hệ thống thì chúng ta sẽ xem qua cách làm thế nào để thêm module của chức năng quản lý người dùng vào menu của trang web. Ta sẽ có 2 bước để thêm 1 module đó là add menu item điều hướng đến module (menu item nằm trên menu của website) và thêm route cho module.

##### Bước 1

Bước 1 chúng ta thêm menu item điều hướng đến module. Trong ví dụ này, ta tạo thêm một menu item Quản lý người dùng vào trong menu để gọi tới module quản lý người dùng.

Một menu item trên menu của website sẽ có icon và phần text hiển thị tương ứng (còn gọi là tên menu item). Đối với tên của menu item, ta định nghĩa tên tiếng anh và tiếng việt trong thư mục *lang* để sau đó có thể gọi ra giá trị tương ứng và add vào tên cho menu item.

A screenshot of a computer screen

Description automatically generated

Sau khi tạo xong tên menu bằng các ngôn ngữ khác nhau, ta sẽ định nghĩa đường link cho module đó và viết vào trong file. Để làm điều này, chúng ta thao tác trên file **src/layouts/sidebar/components/SideBar.jsx**

A screenshot of a computer screen

Description automatically generated

Với các menu item 1 cấp (không có menu con trong nó), ta sử dụng component **Item** như minh họa trong hình dưới đây, trong đó có 3 tham số quan trọng là

* **name**: tên hiển thị của button-tương ứng trong file ngôn ngữ.
* **path**: đường link đến module
* **icon**: icon của module

A screenshot of a computer screen

Description automatically generated

Sau khi làm xong bước này thì trên menu của website sẽ có một menu item điều hướng tương ứng tới module quản lý người dùng.

A screenshot of a computer

Description automatically generated

Đối với các menu item nhiều cấp như ví dụ ở hình dưới :

A screenshot of a computer

Description automatically generated

Trong trường hợp menu item nhiều cấp này, chúng ta có thể tự định nghĩa cách viết của mình. Tuy nhiên lưu ý, tên của menu item được gọi thông qua biến **translate** với cú pháp là:

**translate(‘<giá trị tương ứng cho tên menu trong file ngôn ngữ>’)**

Ví dụ:

**A screenshot of a computer screen

Description automatically generated**

##### Bước 2

Sau khi thêm menu item điều hướng đến module ở bước 1 trong file SideBar.js, trong bước số 2, ta sẽ tạo route cho module quản lý người dùng. Ta sẽ gọi và thêm vào trong file **src/react-routes/Routes.js** như hình dưới đây:

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

Chúng ta sử dụng component PrivateRoute để tạo route cho module. Trong đó mỗi module sẽ khác nhau ở các tham số là :

**arrPage**: mảng giá trị cho các đường link muốn hiển thị trên module này. Ví dụ tương ứng với mảng trong hình ta sẽ có giao diện tương ứng như hình dưới đây:

A screenshot of a computer

Description automatically generated

**link**, **path:** điền vào đường link dẫn đến trang của module.

**pageName:** tên tiêu đề của trang tương ứng với module

**component:** truyền vào component của module đó. Ví dụ với module là quản lý người dùng ta sẽ truyền vào component **ManageUser**.

Như vậy là chúng ta đã hoàn thành xong việc thêm module mới.

##### Cài đặt chức năng trong module

Tiếp theo chúng ta sẽ đi vào code cho phần chức năng hiển thị danh sách người dùng trong hệ thống. Mã nguồn cho chức năng này nằm trong module quản lý người dùng:

A screenshot of a computer screen

Description automatically generated

Đầu tiên,vào thư mục redux, trong file service.js ta sẽ viết chức năng lấy thông tin của tất cả người dùng trong một công ty:

A screenshot of a computer

Description automatically generated

Tương ứng với API ở bên server là */user* với ta truyền vào option có định dạng là :

* **url** – đường dẫn tương ứng với API,
* **method** – phương thức gọi ( cụ thể trường hợp ở đây là GET ), headers – phần headers của http Request ta gửi thêm các dữ liệu như jwt token, browser-finger để có yêu cầu được middleware của server xác thực và cho tiến hành thực hiện gọi service bên server. Trong token – nhận được sau khi login và lưu trong localStorage có chứa các thông tin về user đã được mã hóa,bao gồm như tên, email, roles, ... và đi kèm là thông tin về công ty mà user đó đang làm việc – Id của công ty dùng để thực hiện lệnh truy vấn sẽ được lấy ra từ trong token này do client gửi lên.
* **data** – với những method của yêu cầu thêm, sửa dữ liệu như POST, PATCH thì đây là phần chứa dữ liệu mà client muốn gửi đến server để chỉnh sửa data trong DB. Ở đây GET không yêu cầu chỉnh sửa mà chỉ trích xuất dữ liệu nên phần này là không cần thiết.

Có hai cách để client có thể thực hiện được lời gọi đến API của server là sử dụng axios hoặc fetch đều được. Ở đây chúng ta sử dụng axios.

Sau khi viết xong service ta chuyển đến file constants.js để định nghĩa hành động cho yêu cầu lấy tất cả user trong công ty.

A screenshot of a computer

Description automatically generated

Tiếp đến ta chuyển qua file action.js để viết lời gọi cho hành động này:

A screenshot of a computer

Description automatically generated

Tiếp theo,chúng ta chuyển qua file reducer.js và viết mã nguồn xử lý tương ứng với hành động được yêu cầu trong fle action.js

A screenshot of a computer

Description automatically generated

Sau khi viết xong reducer, ta vào thư mục src/redux để import state của user – lưu trữ data nhận được từ server vào trong store trong file combine-reducers.js

A screenshot of a computer screen

Description automatically generated

Cuối cùng là chúng ta sẽ đi vào code giao diện, lấy dữ liệu từ api và đổ ra màn hình. Hiển thị danh sách các user ta sẽ làm ở component <ManageUserTable />

Sau khi đã viết xong code cho phần gọi API, việc còn lại là ta gọi hàm chạy đó bên trong component này và hiển thị ra.

Đầu tiên ta khai báo UserActions để gọi action get – lấy tất cả user.

A screenshot of a computer screen

Description automatically generated

Tiếp theo đó ta sẽ kết nối đến **store** của redux, gọi đến hàm get của UserActions.

A screenshot of a computer

Description automatically generated

Tiếp theo ta sẽ gọi hàm get trong componentDidMount để load dữ liệu.

A screenshot of a computer screen

Description automatically generated

Dữ liệu trả về sẽ được lưu vào trong user.list – là 1 mảng danh sách các user trong công ty được trả về từ server. Ta sẽ lấy được dữ liệu này thông của qua props.

A screenshot of a computer screen

Description automatically generated

Cuối cùng ta chỉ việc liệt kê tất các các phần tử trong mảng list ra giao diện màn hình.

A screenshot of a computer screen

Description automatically generated

Ta có thể kiểm tra kết quả gọi API trên giao diện thông qua công cụ Redux Dev Tool. Khi check giá trị của user.list nếu như là 1 mảng dữ liệu ví dụ như trong hình thì có nghĩa API đã được gọi và client đã nhận được dữ liệu.

A screenshot of a computer

Description automatically generated

# Cài đặt và cấu hình

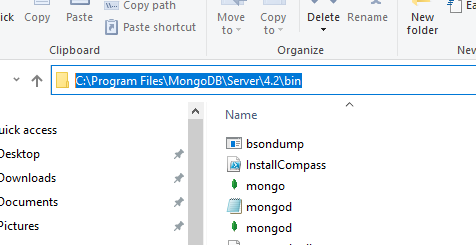
## Cấu hình xác thực truy cập trong MongoDB

(**Chú ý:  text màu đỏ: thay đổi giá trị tùy ý** )

1. Thêm biến môi trường của Mongo (Nếu cần)
   * Chuột phải vào biểu tượng This PC->Properties->Advanced system settings
   * Chọn tab Advanced->Environment Variables
   * Ở phần System variables chọn Path->Edit



* + Tab Edit environment variables hiện lên -> chọn New và thêm đường dẫn đến thư mục **bin** của **MongoDB**

****

* + Chọn OK để kết thúc

1. Tắt tiến trình chạy mặc định của MongoDB trên máy tính.
   * Mở TaskManager
   * Tắt tiến trình của MongoDB Database Server
2. Mở cửa sổ command và gõ lệnh mongod để khởi động lại MongoDB (C1)
3. Mở thêm một cửa sổ command và gõ lệnh mongo (C2)
4. Tại C2 :
   * use admin
   * db.createUser( { user: "*username*", pwd: "*password*" roles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ] } );
   * Tắt C1 và C2.
5. Mở command và gõ lệnh mongod --auth ( tab C1)
6. Mở tiếp command và gõ lệnh mongo -u “*username*” -p “*password*”  --authenticationDatabase “admin” (tab C2)
7. Tại tab C2: gõ lệnh use <tên database của project, ví dụ: *qlcv*>
8. Gõ tiếp lệnh (tạo user là username-qlcv có mật khẩu là *password-qlcv* với quyền là **dbAdmin** cho csdl qlcv

db.createUser( {

user: "*username-qlcv*",

pwd: "*password-qlcv*",

roles: [

{ role: "dbAdmin", db: "*qlcv*" }

]

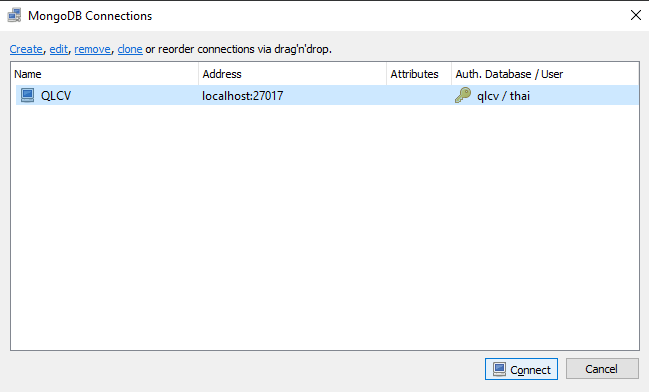
 } );

phân quyền là **dbAdmin** có thể CRUD trong phạm vi của csdl *qlcv*

1. Test thử trên Robomongo:

Chọn kết nối đến cơ sở dữ liệu *qlcv* và nhập *username/password* như ở bước 9 để kết nối đến cơ sở dữ liệu *qlcv*

**

**

**Tham khảo thêm tại**: <https://docs.mongodb.com/manual/tutorial/enable-authentication/>

## Cài đặt và cấu hình project

**Bước 1: Clone project QLCV**

*git clone https://github.com/VNISTResearch/qlcv.git*

**Bước 2: Cấu hình cho server**

Chạy lệnh ***npm install ,*** nếu xuất hiện cảnh báo về phiên bản cũ, chạy lệnh : ***npm audit fix*** để cập nhật.

Tạo file *.env* với cấu trúc giống như trong file *.env.example* bằng lệnh : **cp .env.example .env**, sau đó sửa nội dung của file .env :

* DB\_AUTHENTICATION: xác thực truy cập csdl. Ví dụ: ***true***
* TOKEN\_SECRET: chuỗi kí tự đặc biệt (đặt tùy ý – ví dụ: **congtyvnist** ) dùng để mã hóa token JWT gửi về cho client.
* PORT: cổng dùng để chạy server ( ví dụ : **8000** ).
* WEBSITE : link của website bên phía client ( ví dụ : **http://localhost:3000** )
* SYSTEM\_ADMIN\_NAME: tên tài khoản system admin
* SYSTEM\_ADMIN\_EMAIL: địa chỉ email tài khoản system admin
* SYSTEM\_ADMIN\_PASSWORD: mật khẩu tài khoản system admin
* DEVELOPMENT: cho giá trị false hoặc true. Nên cho giá trị là false. Khi cần bỏ qua các bước xác thực service thì đặt giá trị là true.
* DB\_HOST : host của csdl MongoDB
* DB\_PORT : cổng sử dụng của csdl – mặc định là 27017
* DB\_NAME : tên csdl
* DB\_USERNAME : tên tài khoản được đăng ký xác thực trong MongoDB
* DB\_PASSWORD : mật khẩu
* DB\_BACKUP :có thực hiện backup dữ liệu hay không

Ví dụ cấu hình:

DB\_AUTHENTICATION = true // có bật xác thực truy cập trong MongoDB hay không

DB\_HOST = localhost // host của csdl MongoDB

DB\_PORT = 27017 // cổng sử dụng của csdl – mặc định là 27017

DB\_NAME = qlcv // tên csdl

DB\_USERNAME = thai // tên tài khoản được đăng ký xác thực trong MongoDB

DB\_PASSWORD = 123456 // mật khẩu

DB\_BACKUP = true // có thực hiện backup dữ liệu hay không

TOKEN\_SECRET = QUANLYCONGVIEC

PORT = 8000

WEBSITE = http://localhost:3000

SYSTEM\_ADMIN\_NAME = DatTT

SYSTEM\_ADMIN\_EMAIL = dattt@soict.hust.edu.vn

SYSTEM\_ADMIN\_PASSWORD = 123456

DEVELOPMENT = false

Di chuyển đến thư mục **server/seed** và chạy lệnh ***node initDatabase.js*** để thiết lập các dữ liệu mặc định trong cơ sở dữ liệu. Sau đó chạy lệnh ***node sampleCompanyData.js*** để tạo thêm dữ liệu. Có thể chạy tiếp lệnh ***node organizationalUnitKpiData.js*** để tạo thêm dữ liệu nữa.

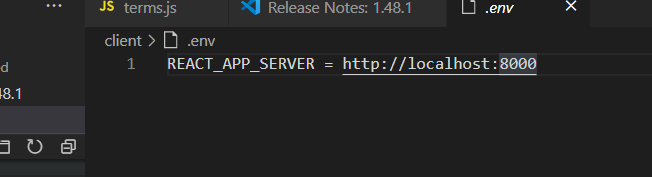
Chạy server: lệnh chạy server có thể được cấu hình lại trong file package.json, nhưng mặc định sẽ có hai kiểu khởi chạy.

* ***npm run dev*** : server sẽ khởi động lại mỗi khi mã nguồn có sự thay đổi mới. ( Điều kiện: máy đã cài đặt gói **nodemon**, nếu chưa cài đặt chạy lệnh ***npm install nodemon***)
* ***npm start*** : server sẽ không khởi lại khi mã nguồn có sự thay đổi.

**Bước 3: Cấu hình cho client**

Tương tự di chuyển vào thư mục client và chạy lệnh: ***npm install***

Tạo file cấu hình bên client , di chuyển vào thư mục **client,** chạy lệnh : **cp .env.example .env** để tạo ra file .**env** và chỉnh sửa nội dung như yêu cầu trong file. Ví dụ:

****

Chạy client : ***npm start***

Nếu có lỗi trong quá trình cài đặt, cấu hình hay chạy project, vui lòng điền thông tin về lỗi vào đây:

<https://docs.google.com/spreadsheets/d/1zSrSCBM7nXXBDrjveTh58IdIRaDlvXuPy_nhDoh-MoU/edit?usp=sharing>

# Hướng dẫn về React

## Class component

### Hướng dẫn phân biệt và sử dụng getDerivedStateFromProps và shouldComponentUpdate

**getDerivedStateFromProps(props, state)**

Hàm này được gọi mỗi khi component hiện tại update state, hoặc component cha truyền props mới cho component con. Và khi dùng redux, hàm này được gọi khi biến redux được cập nhật. Ngoài ra, khi component được mount lần đầu tiên, hàm này cũng được gọi.

Hàm này là hàm static.

props: thuộc tính props mới nhất

state: state mới nhất

1. Khi trước đó gọi this.setState(state1 => state2), hàm getDerivedStateFromProps sẽ được gọi, tham số state cho hàm này sẽ là state2.

2. Khi component cha update props cho component con, hàm getDerivedStateFromProps của component con sẽ được gọi với tham số props mới đó.

Hàm này trả về tham số là state mới (VD: sửa state2 thành state3). Nếu hàm này trả về null, không thay đổi gì state.

**shouldComponentUpdate(nextProps, nextState)**

Hàm này được gọi sau hàm getDerivedStateFromProps, mỗi khi state/props được update. Lưu ý, khi component được mount lần đầu, hàm này không được gọi.

Hàm này trả về true/false. Nếu trả về true, hàm render sẽ được gọi (sau khi render thực hiện xong, sẽ gọi hàm componentDidUpdate). Nếu trả về false, hàm render sẽ không được gọi.

Trong hàm này có thể sử dụng các giá trị this.props, nextProps, this.state, nextState.

1. this.props: props cũ trước khi nhận props mới (nextProps) của component cha truyền vào. Nếu hàm này được gọi do updateState, this.props sẽ như nextProps

2. nextProps: props mới nhận được từ component cha

3. this.state: state cũ. VD nếu trước đó gọi this.setState(state1 => state2), state cũ ở đây sẽ là state1.

4. nextState: state mới. VD trước đó gọi this.setState(state1 => state2), state mới ở đây là state2. Tuy nhiên, nếu state2 này được cập nhật trước đó thành state3 trong hàm getDerivedStateFromProps, nextState sẽ là state3

**Một số lưu ý**

* getDerivedStateFromProps được gọi trước shouldComponentUpdate
* getDerivedStateFromProps có thể có thêm cơ hội sửa lại state một lần nữa, trước khi shouldComponentUpdate được gọi.
* getDerivedStateFromProps là hàm static. Tuy nhiên, lưu ý nó có tham số props và state
* shouldComponentUpdate rất quan trọng để tăng hiệu năng website, nhưng cũng làm tăng độ khó hiểu. Cần phải chú ý tìm điểm cân bằng.

### Trình tự vòng đời ReactJS

Mounting: Các phương thức sau lần lượt được gọi

* constructor()
* render()
* componentDidMount()
* static getDerivedStateFromProps()

Updating: Các phương thức sau lần lượt được gọi

* render()
* componentDidUpdate()
* shouldComponentUpdate()
* static getDerivedStateFromProps()
* getSnapshotBeforeUpdate()

Unmounting:

* componentWillUnmount()

Error handling: có thể dùng 2 phương thức sau để xử lý lỗi

* componentDidCatch()
* static getDerivedStateFromError()

## Function component

### React.memo

React.memo không phải là một React hook. React.memo là một higher order component (HOC). React.memo chỉ dùng cho functional component. Tương đương với React.memo là PureComponent trong class component.

Ý nghĩa như sau. Khi component con đặt trong component cha, nếu component cha render lại, component con sẽ luôn luôn được render lại, dù các thuộc tính truyền vào vẫn giữ nguyên. Để tăng hiệu năng và tránh render không cần thiết, có thể dùng React.memo. React.memo chỉ check xem thuộc tính props (không kiểm tra state) có sự thay đổi không. Nếu không thay đổi, React không render lại component con nữa mà lấy kết quả cũ.

React.memo chỉ thực hiện so sánh nông (shallowly compare[[1]](#footnote-1)) ở props. Muốn so sánh sâu thì ta tự truyền vào hàm so sánh như sau

function MyComponent(props) {

/\* render using props \*/

}

function areEqual(prevProps, nextProps) {

/\*

return true if passing nextProps to render would return

the same result as passing prevProps to render,

otherwise return false

\*/

}

export default React.memo(MyComponent, areEqual);

Ví dụ ứng dụng Memo (Trong ví dụ này, Child là functional component và App là class component)

const Child = React.memo(props => {

return <React.Fragment>{props.name}</React.Fragment>;

});

class App extends React.Component {

state = {

value: 1,

name: "Example"

};

handleClick = () => {

this.setState({

value: this.state.value + 1

});

};

render() {

return (

<React.Fragment>

<Child name={this.state.name} />

<div>{this.state.value}</div>

<button onClick={this.handleClick}>+</button>

</React.Fragment>

);

}

}

Child là component, nhận thuộc tính name từ component App truyền vào. Nếu không dùng Memo, mỗi khi bấm nút, state của App được cập nhật 🡪 App được render lại, và Child cũng được render lại, mặc dù prop name vẫn giữ nguyên. Để hiệu năng tối ưu hơn, ta dùng Memo như trên. Chỉ khi prop name thay đổi, Child mới được render lại.

### React Hooks

#### Giới thiệu

Tài liệu tham khảo: <https://reactjs.org/docs/hooks-reference.html>

React Hooks chính thức được giới thiệu trong phiên bản React 16.8. Nó cho phép chúng ta sử dụng state và các tính năng khác của React với Functional Component mà không phải dùng đến Class Component. Có 2 loại: Hooks cơ bản và Hooks mở rộng (Basic Hooks và Additional Hooks)

Basic Hooks: useState, useEffect, useContext

Additional Hooks: useReducer, useCallback, useMemo, useRef, useImperativeHandle, useLayoutEffect, useDebugValue

Người dùng cũng có thể tự tạo ra các Hooks của mình

#### Lợi ích của React Hooks

**a. Đơn giản hóa vòng đời React, code ngắn gọn hơn**

Ví dụ, chỉ cần useEffect cũng thay thế được cho cả 3 hàm componentDidMount, componentDidUpdate và componentWillUnMount. Do componentDidUpdate chỉ gọi khi component được render lần đầu, những lần render sau hàm này không được gọi nữa, mà thay bằng hàm componentDidUpdate. Nên nếu có công việc chung cần làm sau khi render, phải tách công việc chung đó ra thành 1 hàm riêng và gọi hàm đó trong componentDidMount và componentDidUpdate.

**b. Hướng đến phong cách lập trình hàm, thay vì lập trình hướng đối tượng**

Tốc độ nhanh hơn, code biên dịch ra ngắn gọn hơn

**c.Tránh wrapper hell, tái sử dụng logic xử lý trong component như sau**

Ví dụ, cần lập trình các component có chung logic xử lý gì (đều dùng state repos, loading với logic xử lý như nhau). Cách đơn giản nhất là copy paste đoạn code xử lý với repos, loading như sau vào các component.

componentDidMount () {

this.updateRepos(this.props.id)

}

componentDidUpdate (prevProps) {

if (prevProps.id !== this.props.id) {

this.updateRepos(this.props.id)

}

}

updateRepos = (id) => {

this.setState({ loading: true })

fetchRepos(id)

.then((repos) => this.setState({

repos,

loading: false

}))

}

Nhưng như vậy sẽ bị lặp code. Giải pháp tốt hơn là dùng kỹ thuật Higher-Order Component[[2]](#footnote-2) như sau. Trước tiên định nghĩa component WithRepos

function withRepos (Component) {

return class WithRepos extends React.Component {

state = {

repos: [],

loading: true

}

componentDidMount () {

this.updateRepos(this.props.id)

}

componentDidUpdate (prevProps) {

if (prevProps.id !== this.props.id) {

this.updateRepos(this.props.id)

}

}

updateRepos = (id) => {

this.setState({ loading: true })

fetchRepos(id)

.then((repos) => this.setState({

repos,

loading: false

}))

}

render () {

return (

<Component

{...this.props}

{...this.state}

/>

)

}

}

}

Sau đó các component khác (ReposGrid, Profile) tái sử dụng lại repos, loading và logic liên quan trong component này như sau

// ReposGrid.js

function ReposGrid ({ loading, repos }) {

...

}

export default withRepos(ReposGrid)

// Profile.js

function Profile ({ loading, repos }) {

...

}

export default withRepos(Profile)

Hệ quả là khi tái sử dụng nhiều lần, sẽ dẫn đến code lồng nhau (wrapper hell) như sau

export default withHover(

withTheme(

withAuth(

withRepos(Profile)

)

)

)

Để khắc phục vấn đề này, chúng ta sẽ dùng React Hooks, nhưng phải tự tạo Hook cho mình (Custom Hook)

function useRepos (id) {

const [ repos, setRepos ] = React.useState([])

const [ loading, setLoading ] = React.useState(true)

React.useEffect(() => {

setLoading(true)

fetchRepos(id)

.then((repos) => {

setRepos(repos)

setLoading(false)

})

}, [id])

return [ loading, repos ]

}

Hook tự tạo này được dùng đơn giản như sau

function ReposGrid ({ id }) {

const [ loading, repos ] = useRepos(id)

...

}

function Profile ({ user }) {

const [ loading, repos ] = useRepos(user.id)

...

}

#### Hook useState

Sử dụng hook này, ta không sử dụng this.state, this.setState nữa.

Ví dụ, nếu không dùng hook useState, code như sau:

class TaskComponent extends Component {

constructor(props) {

super(props);

this.state = { isLoading: false }

}

onClick = () => {

this.setState({

isLoading: true,

})

}

render = () => {

return (

<div>

<p>You clicked {count} times</p>

<button onClick={this.onClick}>Click me</button>

</div>

)

}

}

Nếu dùng, code như sau

import { useState } from 'react'

function Counter() {

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

return (

<div>

<p>You clicked {count} times</p>

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

</div>

)

}

Hook useState là một hàm, nhận vào tham số là giá trị ban đầu của thuộc tính, trả về một mảng 2 phần tử, phần tử thứ nhất là biến state, phần tử thứ hai là hàm dùng để cập nhật giá trị của biến state đó.

Ví dụ, với lệnh const [count, setCount] = useState(0); ta có biến trạng thái count, ban đầu nhận giá trị là 0. Để thay đổi giá trị của count, ta gọi hàm setCount.

Muốn tính giá trị mới cho biến state theo giá trị cũ, ta truyền vào tham số cho hàm thay đổi trạng thái. VD: setCount(prevCount => prevCount - 1)}

Lưu ý:

* Ta không có hàm render nữa. Cả hàm Counter tương đương với hàm render. Mỗi lần giá trị count thay đổi, cả hàm Counter trên sẽ được chạy lại.
* Để code giống như cách làm state cũ khi không có hook, ta chỉ cần dùng useState một lần như sau const [state, setState] = useState({p1: 1, p2: 2, p3: 3}); Cách làm này giúp quản lý code dễ dàng hơn (biết được biến nào là state, dễ đổi tên, refactor)
* Cách làm với state cũ trong class (không dùng hook), state tự động được merge. **Nhưng useState không tự động merge**. Để yêu cầu merge, ta dùng cú pháp sau

setState(prevState => {

// Object.assign would also work

return { ...prevState, ...updatedValues };

});

Cách khác là ta có thể dùng hook useReducer

* Đôi khi giá trị ban đầu gán cho state cần tính toán mất chi phí cao, ta không truyền trực tiếp giá trị mà làm như sau

const [state, setState] = useState(() => {

const initialState = someExpensiveComputation(props);

return initialState;

});

Cách làm này giúp chương trình tối ưu hơn. initialState chỉ được tính trong lần render đầu tiên.

#### Hook useEffect

Hook này giúp ta can thiệp vào vòng đời React. Hook useEffect tương đương với cả 3 hàm componentDidMount, componentDidUpdate, và componentWillUnmount. Thường phần gọi API/service sẽ đặt trong useEffect. Khi đó, cần kiểm tra dữ liệu đã tồn tại hay chưa, tránh rơi vào vòng lặp vô hạn

Hook này là một hàm, có 2 tham số. Tham số thứ nhất là một hàm f (được gọi là effect). Khi component render xong, hàm f sẽ được gọi (kể cả lần đầu tiên). Hàm f được quyền sử dụng các state và props của component. Hàm f sẽ được gọi trong một deferred event. Đợi sau khi browser vẽ xong giao diện, hàm f mới được gọi. Hàm f được đảm bảo sẽ được gọi trước lần render tiếp theo.

Tham số thứ hai là một mảng, chứa nhiều biến. Chỉ khi các biến này thay đổi giá trị, hàm f mới được gọi sau khi component render xong. Nếu mảng này rỗng, hàm f sẽ chỉ gọi trong lần render đầu tiên. Tham số thứ hai là tùy chọn. Nếu không truyền vào, hàm f sẽ luôn được gọi sau khi component render xong. Tham số thứ 2 giúp useEffect thay được cho componentDidMount và componentDidUpdate. Ví dụ:

function Counter() {

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

useEffect(() => {

document.title = `You clicked ${count} times`;

}, [title]);

return (

<div>

<p>You clicked {count} times</p>

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

Click me

</button>

</div>

);

}

Để thay thế cho hàm componentWillUnmount, useEffect sẽ trả về một hàm dọn dẹp (Bắt buộc phải trả về hàm). Ví dụ:

useEffect(() => {

const subscription = props.source.subscribe();

return () => {

// Clean up the subscription

subscription.unsubscribe();

};

});

#### Hook useMemo

Đôi khi, trong mỗi lần render ta phải tính toán một giá trị cần thiết nào đó. Nếu thao tác tính toán này có chi phí cao, mỗi lần component được render lại, cần có cơ chế tối ưu hóa. useMemo được thiết kế với mục đích này.

Hook useMemo là một hàm, nhận vào 2 tham số: tham số thứ nhất là một hàm, tham số thứ hai là một mảng các phụ thuộc. Cú pháp:

const memoizedValue = React.useMemo(() => {

fooFunction()

}, [dependencies])

Theo đó, giá trị hàm fooFunction() tính toán được và trả về sẽ được lưu vào biến memoizedValue. Mỗi lần render lại, nếu dependencies không thay đổi, React không cần gọi lại hàm fooFunction để tính toán giá trị memoizedValue, mà lấy luôn giá trị đã lưu của lần tính toán trước đó.

Ví dụ:

function App() {

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

const [wordIndex, setWordIndex] = useState(0);

const words = ["hey", "this", "is", "cool"];

const word = words[wordIndex];

const computeLetterCount = word => {

let i = 0;

while (i < 1000000000) i++;

return word.length;

};

const letterCount = useMemo(() => computeLetterCount(word), [word]);

return (

<div>

<p>"{word}" has {letterCount} letters</p>

<button onClick={() => {

const next = wordIndex + 1 === words.length ? 0 : wordIndex + 1;

setWordIndex(next);

}}

>Next word</button>

<br />

<br />

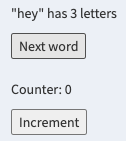
<p>Counter: {count}</p>

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

</div>

);

}



Trong ví dụ trên, component có 2 biến trạng thái là wordIndex và count. Tương ứng, ta có 2 nút, khi nhấn chuột sẽ thay đổi giá trị 2 biến trạng thái này. Bấm một trong 2 nút này, component đều được render lại.

Hàm computeLetterCount là một hàm giả lập, có chi phí cao. Để tối ưu tốc độ render, hàm này được sử dụng trong hook useMemo. Chỉ khi state wordIndex thay đổi, tức là word thay đổi, hàm computeLetterCount mới cần thiết phải tính toán lại. Nếu không dùng useMemo, dù bấm nút nào, tốc độ render của Component cũng đều chậm. Nhờ dùng useMemo, chỉ khi bấm nút “Next word” tốc độ render mới bị chậm (do wordIndex được cập nhật). Khi nhấn nút “Increment”, tốc độ render lại sẽ nhanh.

#### Hook useCallback

useCallback tương tự như useMemo. useMemo để lưu lại một giá trị, còn useCallback để lưu lại một hàm.

Trong một số trường hợp, ta cần truyền vào thuộc tính là một hàm (callback function). Xét ví dụ sau:

const App = () => {

const [count1, setCount1] = React.useState(0)

const [count2, setCount2] = React.useState(0)

const increaseCounter1 = () => {

setCount1(count1 => count1 + 1)

}

const increaseCounter2 = () => {

setCount2(count2 => count2 + 1)

}

return (

<>

<Counter value={count1} onClick={increaseCounter1}>Counter 1</Counter>

<Counter value={count2} onClick={increaseCounter2}>Counter 2</Counter>

</>

)

}

const Counter = React.memo(({ value, children, onClick }) => {

console.log('Render: ', children)

return (

<div>

<button onClick={onClick}>

{children}: {value}

</button>

</div>

)

})

Kết quả như sau



Trong ví dụ trên, Counter nhận vào 2 thuộc tính là value và onClick. Trong đó onClick là một callback function. Mặc dù Counter đã dùng React.memo, increaseCounter1 và increaseCounter2 đã được khai báo là const, nhưng mỗi khi nhấn vào nút Counter hoặc Counter2, cả 2 component đều được render lại. Nguyên nhân là do increaseCounter1 và increaseCounter2 bị thay đổi mỗi lần render lại component App. Counter thấy prop thay đổi, nên Counter luôn luôn bị render lại.

Xét tiếp ví dụ sau

let increaseCounter3 = () => {

}

const App = () => {

const [count1, setCount1] = React.useState(0)

const increaseCounter1 = () => {

setCount1(count1 => count1 + 1)

}

const increaseCounter2 = () => {

}

return (

<>

<Counter value={count1} onClick={increaseCounter1}>Counter 1</Counter>

<Counter value={"N/A"} onClick={increaseCounter2}>Counter 2</Counter>

<Counter value={"N/A"} onClick={increaseCounter3}>Counter 3</Counter>

</>

)

}

const Counter = React.memo(({ value, children, onClick }) => {

console.log('Render: ', children)

return (

<div>

<button onClick={onClick}>

{children}: {value}

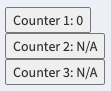
</button>

</div>

)

})

Kết quả giao diện



Trong ví dụ này, Counter 2 tương tự như Counter 3. Khác biệt là tham số onClick callback truyền vào. Khi bấm nút số 1, nút số 2 cũng render lại, trong khi nút số 3 không render lại. Nguyên nhân là do hàm increaseCounter3 được khai báo ở ngoài component.

Để khắc phục component render lại khi thuộc tính truyền vào là hàm, ta cần ghi nhớ lại hàm truyền vào bằng cách sử dụng useCallback hook như sau:

const App = () => {

const [count1, setCount1] = React.useState(0)

const [count2, setCount2] = React.useState(0)

const increaseCounter1 = React.useCallback(() => {

setCount1(count1 => count1 + 1)

}, [])

const increaseCounter2 = React.useCallback(() => {

setCount2(count2 => count2 + 1)

}, [])

return (

<>

<Counter value={count1} onClick={increaseCounter1}>Counter 1</Counter>

<Counter value={count2} onClick={increaseCounter2}>Coutner 2</Counter>

</>

)

}

const Counter = React.memo(({ value, children, onClick }) => {

console.log('Render: ', children)

return (

<div>

<button onClick={onClick}>

{children}: {value}

</button>

</div>

)

})

Hook useCallback là một hàm, nhận vào 2 tham số là hàm callback và mảng. Với cách làm này, hàm callback sẽ được lưu lại trong bộ nhớ, chỉ được cập nhật nếu một phần tử trong mảng có sự thay đổi. Nếu mảng là rỗng như trong ví dụ trên, hàm callback chỉ được tạo duy nhất một lần, không bao giờ thay đổi

Sử dụng useCallback(fn, deps) là hoàn toàn tương đương với useMemo(() => fn, deps). Ví dụ trên có thể sửa lại như sau

const App = () => {

const [count1, setCount1] = React.useState(0)

const [count2, setCount2] = React.useState(0)

const increaseCounter1 = React.useMemo(() => (() => {

setCount1(count1 => count1 + 1)

}), [])

const increaseCounter2 = React.useMemo(() => (() => {

setCount2(count2 => count2 + 1)

}), [])

return (

<>

<Counter value={count1} onClick={increaseCounter1}>Counter 1</Counter>

<Counter value={count2} onClick={increaseCounter2}>Coutner 2</Counter>

</>

)

}

const Counter = React.memo(({ value, children, onClick }) => {

console.log('Render: ', children)

return (

<div>

<button onClick={onClick}>

{children}: {value}

</button>

</div>

)

})

useMemo tương tự như useCallback. Cả 2 hook này đều là hàm nhận vào hai tham số, tham số thứ nhất là 1 hàm f, tham số thứ hai là một mảng. Khác biệt là useMemo sẽ gọi hàm f và trả về giá trị mà hàm f trả về, còn useCallback sẽ trả về hàm f, và không thực thi hàm f.

#### Hướng dẫn sử dụng chung

Trong cùng một component, có thể sử dụng số lượng các hook (VD useState và useEffect) tùy ý. Nhưng bắt buộc các hooks này phải được đặt trực tiếp trong thân hàm, không nằm trong if, switch, vòng lặp for, while, hay trong các hàm lồng nhau. Ví dụ, cách làm sau là không hợp lệ

function Counter(props) {

if (props.shouldCount) {

const [count, setCount] = useState(0) // Lỗi

}

}

Để lấy thuộc tính từ cha truyền vào, sử dụng tham số cho hàm như sau. Tên tham số props đặt tùy ý. Các thuộc tính từ cha truyền vào sẽ được lưu trong tham số props này. Khi một props thay đổi, component sẽ được render lại (hàm sẽ được gọi lại)

function ExampleCreateForm(props) {

const [count, setCount] = useState(props.start);

return (

<div>

<p>You clicked {count} times</p>

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

</div >

)

}

Hoặc cũng có thể làm như sau (lấy luôn tên các props truyền vào)

function ExampleCreateForm({start, end, example}) {

const [count, setCount] = useState(props.start);

return (

<div>

<p>You clicked {count} times</p>

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

</div >

)

}

Hook chỉ được sử dụng với functional component, không dùng với class component.

# Tài liệu tham khảo

https://www.tutorialspoint.com/nodejs/index.htm

https://reactjs.org/tutorial/tutorial.html

https://redux.js.org/basics/basic-tutorial/

https://www.tutorialspoint.com/mongodb/index.htm

1. <https://github.com/facebook/react/blob/v16.8.6/packages/shared/shallowEqual.js> [↑](#footnote-ref-1)
2. <https://reactjs.org/docs/higher-order-components.html> [↑](#footnote-ref-2)