# Mục lục

[Mục lục 1](#_Toc64838856)

[Danh mục hình vẽ 5](#_Toc64838857)

[Danh mục bảng 6](#_Toc64838858)

[Chương 1 Giới thiệu 7](#_Toc64838859)

[1.1 Công nghệ sử dụng trong dự án quản lý công việc 7](#_Toc64838860)

[1.2 Một số thuật ngữ cần chú ý 7](#_Toc64838861)

[Chương 2 Mô hình phân quyền RBAC và cấu trúc CSDL trong dự án quản lý công việc 9](#_Toc64838862)

[2.1 Mô hình phân quyền RBAC 9](#_Toc64838863)

[2.2 Cơ sở dữ liệu của dự án Quản Lý Công Việc 9](#_Toc64838864)

[2.2.1 Mô hình 9](#_Toc64838865)

[2.2.2 Company 12](#_Toc64838866)

[2.2.3 User 12](#_Toc64838867)

[2.2.4 Role 13](#_Toc64838868)

[2.2.5 UserRole 13](#_Toc64838869)

[2.2.6 RoleType 14](#_Toc64838870)

[2.2.7 Department 16](#_Toc64838871)

[2.2.8 Privilege 16](#_Toc64838872)

[2.2.9 Link 17](#_Toc64838873)

[2.2.10 Component 17](#_Toc64838874)

[2.2.11 Action 18](#_Toc64838875)

[Chương 3 Project quản lý công việc 19](#_Toc64838876)

[3.1 Coding convention 19](#_Toc64838877)

[3.1.1 Đặt tên Rest Services 19](#_Toc64838878)

[3.1.2 MongoDB 21](#_Toc64838879)

[3.1.3 NodeJS 23](#_Toc64838880)

[3.1.4 ReactJS 23](#_Toc64838881)

[3.2 Bảo mật trên website 25](#_Toc64838882)

[3.2.1 Các kỹ thuật được sử dụng trong bảo mật 25](#_Toc64838883)

[3.2.2 Xác thực tài khoản người dùng 27](#_Toc64838884)

[3.2.3 Xác thực quyền của người dùng 28](#_Toc64838885)

[3.3 Cấu trúc thư mục 28](#_Toc64838886)

[3.3.1 Cấu trúc của project 28](#_Toc64838887)

[3.3.2 Cấu trúc thư mục Server 28](#_Toc64838888)

[3.3.3 Cấu trúc thư mục Client 32](#_Toc64838889)

[Chương 4 Cài đặt và cấu hình 45](#_Toc64838890)

[4.1 Cấu hình xác thực truy cập trong MongoDB 45](#_Toc64838891)

[4.2 Cài đặt cấu hình và chạy project 47](#_Toc64838892)

[Chương 5 Cấu trúc project 52](#_Toc64838893)

[5.1 Thư mục Client 52](#_Toc64838894)

[5.1.1 Common components 53](#_Toc64838895)

[5.1.2 Helpers 54](#_Toc64838896)

[5.1.3 Lang 56](#_Toc64838897)

[5.1.4 Layout 56](#_Toc64838898)

[5.1.5 Modules 57](#_Toc64838899)

[5.1.6 React-routers 57](#_Toc64838900)

[5.1.7 Redux 59](#_Toc64838901)

[5.1.8 index.js app.js và app.css 61](#_Toc64838902)

[5.1.9 Các file khác 61](#_Toc64838903)

[5.2 Cấu trúc thư mục server 62](#_Toc64838904)

[5.2.1 Index.js 62](#_Toc64838905)

[5.2.2 Helpers 62](#_Toc64838906)

[5.2.3 Log 63](#_Toc64838907)

[5.2.4 Middleware 64](#_Toc64838908)

[5.2.5 Model 67](#_Toc64838909)

[5.2.6 Modules 67](#_Toc64838910)

[5.2.7 Seed 68](#_Toc64838911)

[5.2.8 Upload 68](#_Toc64838912)

[5.2.9 Package.json 68](#_Toc64838913)

[5.3 Cách tạo mới một module 68](#_Toc64838914)

[5.3.1 Mã nguồn Client 68](#_Toc64838915)

[5.3.2 Mã nguồn Server 73](#_Toc64838916)

[5.4 Giải thích module example 78](#_Toc64838917)

[5.4.1 Mã nguồn Client theo mô hình 1 78](#_Toc64838918)

[5.4.2 Mô hình Example2 96](#_Toc64838919)

[5.4.3 Mã nguồn Server 100](#_Toc64838920)

[Chương 6 Hướng dẫn về React 104](#_Toc64838921)

[6.1 Class component 104](#_Toc64838922)

[6.1.1 Vòng đời component 104](#_Toc64838923)

[6.1.2 Hướng dẫn phân biệt và sử dụng getDerivedStateFromProps và shouldComponentUpdate 104](#_Toc64838924)

[6.1.3 Trình tự vòng đời ReactJS 105](#_Toc64838925)

[6.2 Function component 106](#_Toc64838926)

[6.2.1 React.memo 106](#_Toc64838927)

[6.2.2 React Hooks 108](#_Toc64838928)

[Chương 7 Hướng dẫn sử dụng webpack build server 120](#_Toc64838929)

[7.1 Cài đặt webpack 120](#_Toc64838930)

[7.2 Chạy webpack 121](#_Toc64838931)

[Tài liệu tham khảo 123](#_Toc64838932)

# Danh mục hình vẽ

[**Hình 1** Ví dụ về JSON 7](#_Toc64838933)

[**Hình 2** Mô hình CSDL 10](#_Toc64838934)

[**Hình 3** Ví dụ về coding convention trong MongDB 22](#_Toc64838935)

[**Hình 4** JSX đa dòng 24](#_Toc64838936)

# Danh mục bảng

Nội dung

[Mục lục 1](#_Toc64838937)

[Danh mục hình vẽ 5](#_Toc64838938)

[Danh mục bảng 6](#_Toc64838939)

[Chương 1 Giới thiệu 7](#_Toc64838940)

[1.1 Công nghệ sử dụng trong dự án quản lý công việc 7](#_Toc64838941)

[1.2 Một số thuật ngữ cần chú ý 7](#_Toc64838942)

[Chương 2 Mô hình phân quyền RBAC và cấu trúc CSDL trong dự án quản lý công việc 9](#_Toc64838943)

[2.1 Mô hình phân quyền RBAC 9](#_Toc64838944)

[2.2 Cơ sở dữ liệu của dự án Quản Lý Công Việc 9](#_Toc64838945)

[2.2.1 Mô hình 9](#_Toc64838946)

[2.2.2 Company 12](#_Toc64838947)

[2.2.3 User 12](#_Toc64838948)

[2.2.4 Role 13](#_Toc64838949)

[2.2.5 UserRole 13](#_Toc64838950)

[2.2.6 RoleType 14](#_Toc64838951)

[2.2.7 Department 16](#_Toc64838952)

[2.2.8 Privilege 16](#_Toc64838953)

[2.2.9 Link 17](#_Toc64838954)

[2.2.10 Component 17](#_Toc64838955)

[2.2.11 Action 18](#_Toc64838956)

[Chương 3 Project quản lý công việc 19](#_Toc64838957)

[3.1 Coding convention 19](#_Toc64838958)

[3.1.1 Đặt tên Rest Services 19](#_Toc64838959)

[3.1.2 MongoDB 21](#_Toc64838960)

[3.1.3 NodeJS 23](#_Toc64838961)

[3.1.4 ReactJS 23](#_Toc64838962)

[3.2 Bảo mật trên website 25](#_Toc64838963)

[3.2.1 Các kỹ thuật được sử dụng trong bảo mật 25](#_Toc64838964)

[3.2.2 Xác thực tài khoản người dùng 27](#_Toc64838965)

[3.2.3 Xác thực quyền của người dùng 28](#_Toc64838966)

[3.3 Cấu trúc thư mục 28](#_Toc64838967)

[3.3.1 Cấu trúc của project 28](#_Toc64838968)

[3.3.2 Cấu trúc thư mục Server 28](#_Toc64838969)

[3.3.3 Cấu trúc thư mục Client 32](#_Toc64838970)

[Chương 4 Cài đặt và cấu hình 45](#_Toc64838971)

[4.1 Cấu hình xác thực truy cập trong MongoDB 45](#_Toc64838972)

[4.2 Cài đặt cấu hình và chạy project 47](#_Toc64838973)

[Chương 5 Cấu trúc project 52](#_Toc64838974)

[5.1 Thư mục Client 52](#_Toc64838975)

[5.1.1 Common components 53](#_Toc64838976)

[5.1.2 Helpers 54](#_Toc64838977)

[5.1.3 Lang 56](#_Toc64838978)

[5.1.4 Layout 56](#_Toc64838979)

[5.1.5 Modules 57](#_Toc64838980)

[5.1.6 React-routers 57](#_Toc64838981)

[5.1.7 Redux 59](#_Toc64838982)

[5.1.8 index.js app.js và app.css 61](#_Toc64838983)

[5.1.9 Các file khác 61](#_Toc64838984)

[5.2 Cấu trúc thư mục server 62](#_Toc64838985)

[5.2.1 Index.js 62](#_Toc64838986)

[5.2.2 Helpers 62](#_Toc64838987)

[5.2.3 Log 63](#_Toc64838988)

[5.2.4 Middleware 64](#_Toc64838989)

[5.2.5 Model 67](#_Toc64838990)

[5.2.6 Modules 67](#_Toc64838991)

[5.2.7 Seed 68](#_Toc64838992)

[5.2.8 Upload 68](#_Toc64838993)

[5.2.9 Package.json 68](#_Toc64838994)

[5.3 Cách tạo mới một module 68](#_Toc64838995)

[5.3.1 Mã nguồn Client 68](#_Toc64838996)

[5.3.2 Mã nguồn Server 73](#_Toc64838997)

[5.4 Giải thích module example 78](#_Toc64838998)

[5.4.1 Mã nguồn Client theo mô hình 1 78](#_Toc64838999)

[5.4.2 Mô hình Example2 96](#_Toc64839000)

[5.4.3 Mã nguồn Server 100](#_Toc64839001)

[Chương 6 Hướng dẫn về React 104](#_Toc64839002)

[6.1 Class component 104](#_Toc64839003)

[6.1.1 Vòng đời component 104](#_Toc64839004)

[6.1.2 Hướng dẫn phân biệt và sử dụng getDerivedStateFromProps và shouldComponentUpdate 104](#_Toc64839005)

[6.1.3 Trình tự vòng đời ReactJS 105](#_Toc64839006)

[6.2 Function component 106](#_Toc64839007)

[6.2.1 React.memo 106](#_Toc64839008)

[6.2.2 React Hooks 108](#_Toc64839009)

[Chương 7 Hướng dẫn sử dụng webpack build server 120](#_Toc64839010)

[7.1 Cài đặt webpack 120](#_Toc64839011)

[7.2 Chạy webpack 121](#_Toc64839012)

[Tài liệu tham khảo 123](#_Toc64839013)

# 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

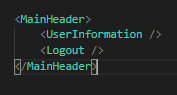
#### Quy tắc chung

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

#### 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

#### 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

Các bước thực hiện: (**Chú ý:  text màu đỏ: thay đổi giá trị tùy ý** )

**Bước 1**: Thêm biến môi trường của Mongo (Nếu cần)

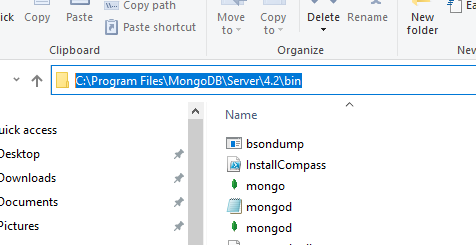
Chuột phải vào biểu tượng This PC->Properties->Advanced system settings

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

**Bước 2**: Tắt tiến trình chạy mặc định của MongoDB trên máy tính.

Mở TaskManager

Tắt tiến trình của MongoDB Database Server

**Bước 3**: Mở cửa sổ command và gõ lệnh mongod để khởi động lại MongoDB (C1)

**Bước 4**: Mở thêm một cửa sổ command và gõ lệnh mongo (C2)

**Bước 5**: Tại C2 :

use admin

db.createUser( { user: "*username*", pwd: "*password*" roles: [ { role: "userAdminAnyDatabase", db: "admin" }, "readWriteAnyDatabase" ] } );

Tắt C1 và C2.

**Bước 6**: Mở command và gõ lệnh mongod --auth ( tab C1)

**Bước 7**: Mở tiếp command và gõ lệnh mongo -u “*username*” -p “*password*”  --authenticationDatabase “admin” (tab C2)

**Bước 8**: Tại tab C2: gõ lệnh use <tên database của project, ví dụ: *qlcv*>

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*" }

]

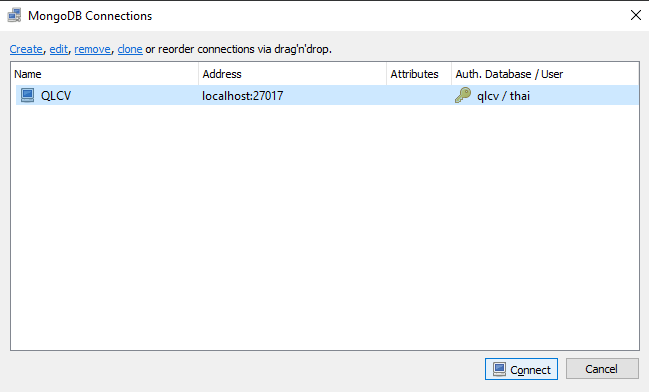
 } );

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

**Bước 9**: 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 cấu hình và chạy project

Cần đăng kí email công ty và xin được cấp quyền truy cập vào repo. Clone project tại repo: <https://github.com/VNISTResearch/qlcv.git>

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

Sau khi clone project về sẽ thấy project QLCV có 2 thư mục chính là client, server. Client (front-end) viết bằng Reactjs, Server (backend) viết bằng Nodejs

Yêu cầu: cài đặt Nodejs, MongoDB. Cài đặt thêm ứng dụng Robo 3T để quản lý và thao tác với MongoDB dễ dàng hơn

**Cài đặt Server:**

Cd vào thư mục server. Chạy lệnh npm install (hoặc yarn 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: MongoDB có cài đặt xác thực truy cập csdl không. Nếu có, đặt giá trị này 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
* 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: cài đặt giá trị true nếu muốn bỏ qua các bước xác thực cho service (ví dụ dùng khi test service với POSTMAN)

Ví dụ cấu hình:

DB\_AUTHENTICATION = true // có 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

**Lưu ý:** Mỗi khi muốn đổi tham số trong file .env, cần khởi động lại server

Để có dữ liệu minh họa sẵn (seed data) khi cài đặt server, mở terminal, **cd** vào thư mục **server**, chạy lệnh: npm run init. Lệnh này được định nghĩa trong file **server/package.json**, sẽ tạo đầy đủ các loại dữ liệu minh họa cho ứng dụng. Một số lệnh khác tạo ít dữ liệu minh họa hơn có thể tham khảo thêm trong file **package.json**

...

"scripts": {

"build": "(webpack) && (nodemon build/backend.js)",

"watch": "webpack --watch",

"test": "echo \"Error: no test specified\" && exit 1",

"start": "node index.js",

"dev": "nodemon index.js",

"db": "node seed/initDB.js",

"viavet": "node seed/viavet.js",

"sc": "node seed/initSampleCompanyDB.js",

"sp": "node seed/initSamplePackage.js",

"skpi": "node seed/initOrganizationalUnitKPIData.js",

"shr": "node seed/initHumanResourceData.js",

"init-db": "node seed/initDB.js",

"init-sample-company": "node seed/initSampleCompanyDB.js",

"init-sample-organizational-unit-kpi": "node seed/initOrganizationalUnitKPIData.js",

"init-human-resource-data": "node seed/initHumanResourceData.js",

"init": "node seed/initDB.js; node seed/initSampleCompanyDB.js; node seed/initOrganizationalUnitKPIData.js; node seed/initHumanResourceData.js"

},

...

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.

Sau khi chạy xong cả phía client lẫn server thì tiến hành đăng nhập vào hệ thống với các tài khoản nằm trong file qlcv/server/seed/initSampleCompanyDB.js

Mật khẩu của tất cả các tài khoản đều là vnist123@. Portal là vnist. Một số email tài khoản:

* Admin: [admin.vnist@gmail.com](mailto:admin.vnist@gmail.com)
* Giám đốc: [nva.vnist@gmail.com](mailto:nva.vnist@gmail.com)
* Thành viên ban giám đốc: [vtc.vnist@gmail.com](mailto:vtc.vnist@gmail.com)
* Trưởng phòng kinh doanh: [nvd.vnist@gmail.com](mailto:nvd.vnist@gmail.com)
* Nhân viên phòng kinh doanh: [nhungnt.vnist@gmail.com](mailto:nhungnt.vnist@gmail.com)

const users = await User(vnistDB).insertMany([

{

name: "Super Admin VNIST",

email: "super.admin.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

{

name: "Admin VNIST",

email: "admin.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

{

name: "Nguyễn Văn An",

email: "nva.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

{

name: "Trần Văn Bình",

email: "tvb.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

{

name: "Vũ Thị Cúc",

email: "vtc.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

{

name: "Nguyễn Văn Danh",

email: "nvd.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

{

name: "Trần Thị Én",

email: "tte.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

{

name: "Phạm Đình Phúc",

email: "pdp.vnist@gmail.com",

password: hash,

company: vnist.\_id,

},

**Cài đặt Client:**

* **Cd** đến thư mục **client**
* Cài đặt các gói thư viện khai báo trong file **package.json** bằng lệnh: npm install hoặc (yarn install), với điều kiện máy tính phải cài **yarn** hoặc **npm**.
* Tạo 1 file **.env** (qlcv/client/.env) có nội dung như sau để khai báo địa chỉ của server để gọi các service (có thể copy cho nhanh bằng cách chạy lệnh sau: **cp .env.example .env**)

REACT\_APP\_SERVER = <http://localhost:8000>

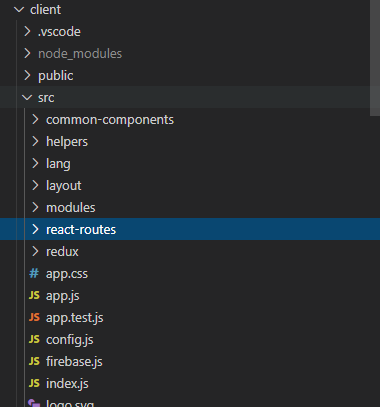
* Lưu cấu hình file **.env** lại và ở trong thư mục **client**, tiến hành chạy lệnh npm start hoặc yarn start trên Terminal

# Cấu trúc project

## Thư mục Client

Thư mục client chứa mã nguồn frontend của hệ thống, sử dụng ngôn ngữ Reactjs. Cấu trúc của project về cơ bản giống với frame của 1 project ReactJS. Tuy nhiên có sự khác biệt là ở trong thư mục src, cấu trúc mã nguồn sẽ 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 ở client là : public và src.



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

**src**: chứa mã nguồn cho phần giao diện. Dưới đây sẽ giải thích các thành phần trong thư mục src này.

### Common components

Thư mục này chứa các component thường dùng, được sử dụng chung cho các module trong project. Một số component là JS thuần, một số component là wrapper của React Component, một số component tự viết từ đầu. Cách tạo các component như vậy giúp tăng tính tái sử dụng, tránh lặp code, và đặc biệt giúp ứng dụng linh hoạt với các sửa đổi. Sau này, nếu cần sửa đổi component, thay thư viện component, sẽ phải sửa đổi ít nhất có thể.

Mỗi component ứng với một thư mục con trong thư mục common-components/src. File chỉ mục common-components/index.js tổng hợp lại các component này, để thuận tiện cho việc import.

Ví dụ, component hay sử dụng nhất là **component modal**: Component này giúp tạo các bootstrap **modal (giống các popup window)**, khi dùng chỉ cần truyền các tham số cần thiết vào.

Để sử dụng component, trước tiên phải import component đó bằng dòng lệnh:

import { DialogModal} from '../../../../common-components';

Tùy vào cấp độ thư mục của module, cần chỉnh sửa lại đường dẫn tương đối ở trên.

Sau đó, trong phần render trong ReactJS, ta gọi component như sau:

<DialogModal

     modalID="modal-create-user" isLoading={user.isLoading}

     title={translate('manage\_user.add\_title')}

     func={this.save}

     disableSubmit={!this.isFormValidated()}>

     {/\* Trong này sẽ là code giao diện ---- ví dụ\*/}

     <div className={`form-group`}>

         <label>Họtên</label>

         <input type="text" className="form-control"

onChange={this.handleUserName} />

     </div>

</DialogModal>

Giải thích các thuộc tính

**modalID**: id cho modal (bắt buộc). Các modal cần có id khác nhau.

**title**: Tiêu đề cho modal

**disableSubmit**: Thường dùng để validate form bên trong modal, nếu bằng true thì vô hiệu hóa nút **lưu** không thể bấm được, false thì ngược lại

**func**: Hàm được gọi khi bấm nút submit trên modal

Các component khác sử dụng tương tự. Tham khảo các module đã sử dụng các component này và học theo.

### Helpers

Thư mục này gồm các file chứa các hàm tiện ích, được sử dụng nhiều ở các module, quan trọng nhất là file requestHelper.js, file này chứa các function giao tiếp với **server api** của project, cụ thể là function sendRequest(), đảm nhiệm việc gửi request tới server:

#### Gửi request tới server

export async function sendRequest(options, showSuccessAlert = false,

showFailAlert = true, module, successTitle = 'general.success',

 errorTitle = 'general.error') {

    const requestOptions = {

        url: options.url,

        method: options.method,

        data: options.data,

        params: options.params,

        responseType: options.responseType,

        headers: await AuthenticateHeader()

    };

    return axios(requestOptions).then(res => {

        const messages = Array.isArray(res.data.messages) ?

res.data.messages : [res.data.messages];

        showSuccessAlert && toast.success(

            <ServerResponseAlert

                type='success'

                title={successTitle}

                content={messages.map(message => `${module}.${message}`)}

            />,

            { containerId: 'toast-notification' }

        );

        return Promise.resolve(res);

    }).catch(err => {

        let messages;

        if (!err.response) {

            showServerDisconnectedError();

        } else {

            messages = Array.isArray(err.response.data.messages)

?err.response.data.messages : [err.response.data.messages];

        }

        if (messages) {

            if (checkErrorAuth(messages[0]))

                showAuthResponseAlertAndRedirectToLoginPage();

            else if (messages[0] === 'acc\_log\_out') {

                clearStorage();

            }

            else {

                showFailAlert && toast.error(

                    <ServerResponseAlert

                        type='error'

                        title={errorTitle}

                        content={messages.map(message =>

`${module}.${message}`)}

                    />,

                    { containerId: 'toast-notification' }

                );

            }

        }

        return Promise.reject(err);

    })

}

Hàm bao gồm 6 tham số:

Tham số Options: chứa các thông tin của request như:

**url**: Địa chỉ api mà cần gọi

**method**: Phương thức (GET, POST, DELETE,….)

**data**: Dữ liệu gửi kèm khi call api

**params**: là các thông số URL được gửi khi call api

**responseType**: Loại dữ liệu máy chủ sẽ phản hồi

**headers**: dữ liệu gửi trong header có dạng key-value, trong project đang gọi tới hàm AuthenticateHeader()

import { key } from './pub.json'

function encryptMessage(message) {

    const publicKey = key;

    const jsEncrypt = new JSEncrypt();

    jsEncrypt.setPublicKey(publicKey);

    return jsEncrypt.encrypt(message);

}

const AuthenticateHeader = async () => {

    const fpAgent = await FingerprintJS.load();

    const result = await fpAgent.get();

    const fingerprint = result.visitorId;

    return {

        "crtp": encryptMessage(window.location.pathname),

        "fgp": encryptMessage(fingerprint.toString()),

        "utk": getStorage('jwt'),

        "crtr": encryptMessage(getStorage('currentRole'))

    }

}

Hàm này sẽ mã hóa **3** trường thông tin được gửi đi trong headers khi gọi api bao gồm:

**crtp:** viết tắt của từ **current-page,** là trang hiện tại mà người dùng đang truy và gọi **api.** Ví dụ “/task-management”.

**fgp:** viết tắt của fingerprint, sử dụng gói **fingerprintjs** để tăng cường bảo mật khi truy cập trên hệ thông website

**crtr**: viết tắt của current-role, là **role** hiện tại mà người dùng có để gọi **api.** (một tài khoản có thể có nhiều role với các chức danh khác nhau).

Còn lại một trường không được mã hóa đó là **utk** (JWT: json web token)

Ngoài ra còn có các file tiện ích khác như validate form, format string, ….

#### Validate form

### Lang

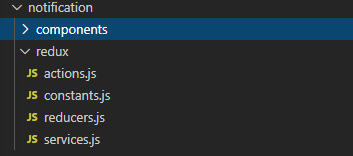
Thư mục này chưa các file ngôn ngữ của hệ thống, phục vụ cho việc sử dụng đa ngôn ngữ trên website, và hiện tại có 2 file **en.js** và **vi.js** lần lượt ngôn ngữ là tiếng anh và tiếng việt.

### Layout

Thư mục chứa các component phần layout – bố cục trên 1 trang web, gồm : Header, SideBar, Content, Footer

### Modules

Chứa các module của hệ thống. Mỗi module sẽ gồm 2 phần chính: **components (**nơi chứa file giao diện .jsx, css,…**)** và **redux**



**redux:** Nơi chứa các file thao tác với server và xử lý dữ liệu nhận về, lưu vào kho redux**,**  bao gồm 4 file:

* 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-routers

Thư mục chứa các file cấu hình định tuyến cho website, gồm 3 file: authRoute, privateRoute, và routes

authRoute, privateRoute là các wrapper Route tiện ích. Khi tạo 1 trang mới, không cần sửa 2 file này, chỉ cần khai báo trang trong routes, sử dụng 2 tiện ích nói trên.

#### authRoute

**authRoute.js**: là Route tiện ích bọc lấy React Route, giúp trả về component (là một thuộc tính truyền vào AuthRoute) khi khớp đường dẫn (cũng là một thuộc tính truyền vào). Tiện ích Route này dùng để điều hướng trang. Chỉ khi người dùng chưa đăng nhập (chưa có giá trị json web token trong local storage), họ mới truy cập được vào trang với đường dẫn tương ứng. Nếu người dùng đã đăng nhập, các trang với đường dẫn truyền vào sẽ không vào được nữa, vì khi truy cập, authRoute này sẽ luôn điều hướng họ về trang **/home**.

import React from 'react';

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

import { getStorage } from '../config';

export const AuthRoute = ({ auth, component: Component, layout: Layout, ...rest }) => (

<Route {...rest} render={props => {

var logged = getStorage(); // Lấy ra json web token

return logged === null ? <Component {...props} /> : <Redirect to='/home' />;

}} />

)

Có 3 trang sử dụng authRoute, là trang giới thiệu sản phẩm (/), trang login (/login), và trang reset mật khẩu (/reset-password). Trang reset mật khẩu khi được gọi phải có mã otp và email đi kèm. Khi người dùng quên mật khẩu, hệ thống gửi email cho họ, có đường link. Click vào đường link này (với mã otp và email là tham số URL), họ sẽ đến trang reset mật khẩu.

#### privateRoute

**privateRoute.js**: tiện ích Route này cũng bọc lấy React Route như authRoute, giúp trả về component (là một thuộc tính truyền vào PrivateRoute) khi khớp đường dẫn (cũng là một thuộc tính truyền vào).

Tiện ích Route này được sử dụng để điều hướng trang. Chức năng hơi ngược so với authRoute. Nếu người dùng chưa đăng nhập, Route này điều hướng về trang **login** (/login). Nếu người dùng đã đăng nhập, Route này sẽ kiểm tra người dùng có quyền truy cập vào trang đó hay không, nếu có sẽ trả về component tương ứng như bình thường (theo layout là thuộc tính truyền vào). Nếu không được phép vào thì sẽ redirect về trang **home**

**Redirect về trang home:**

 if(link !== '/' && checkURL(link, auth.links) !== true){

                return <Redirect to='/home'/>

            }

**Trả lại Layout chứa Component tương ứng khi được phép truy cập**:

return <Layout arrPage={ arrPage } pageName={ pageName }

isLoading={ isLoading }><Component {...props}/>

</Layout>

#### routes

**routes.js**: file cấu hình định tuyến cho website (tùy địa chỉ URL, gọi ra component tương ứng trong các module)

Ví dụ 1, để cấu hình route cho trang login. Trang này sử dụng authRoute.js (theo yêu cầu nghiệp vụ). Các bước như sau:

**Bước 1**: import component

import Login from "../modules/auth/components/login";

**Bước 2**: sử dụng AuthRoute

<AuthRoute

    exact

    auth={auth}

    path="/login"

    component={Login}

/>

Với **path** (đường dẫn) là “/login”, **component** là Login đã import ở bước trên

Để minh họa cấu hình định tuyến cho 1 trang bất kì (các trang này chỉ được truy cập khi người dùng đã đăng nhập), xét ví dụ 2, định tuyến trang notificationsử dụng PrivateRoute như sau

<PrivateRoute

    isLoading={false}

    key={"notifications"}

    arrPage={[

    { link: "/", name: "home", icon: "fa fa-home" },

    {

    link: "/notifications",

    name: "notifications",

    icon: "fa fa-bell",

    },

        ]}

    auth={auth}

    exact={true}

    link={"/notifications"}

    path={"/notifications"}

    pageName={"notifications"}

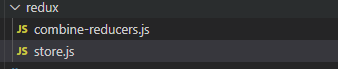
    layout={Layout}

    component={Notifications}

/>

### Redux

Đây là thư mục chứa các file, cấu hình bật redux (redux thunk) cụ thể ở đây là file store.js, và file combine-reducers.js có chức năng combine các file reducer từ các module lại .



#### Store.js

Store.js: đăng ký sử dụng redux. Khi phát triển các module mới, không cần quan tâm tới file này. Có thể dùng tiện ích log middleware trong gói redux-logger, ghi lại các thay đổi với kho trạng thái redux như sau

import { createStore, applyMiddleware, compose } from "redux";

import thunkMiddleware from "redux-thunk";

import { createLogger } from 'redux-logger';

import rootReducer from "./combine-reducers";

const composeEnhancers = window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

var store = createStore(

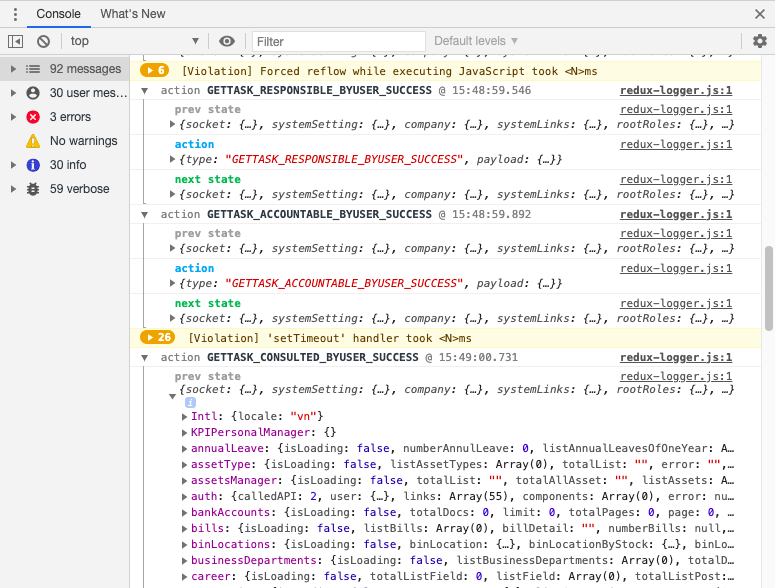
rootReducer,

composeEnhancers(applyMiddleware(thunkMiddleware, createLogger()))

);

export default store;

Khi bật console trình duyệt sẽ thấy lịch sử thay đổi kho trạng thái redux



Tuy nhiên, bật log làm chậm chương trình. Nên tắt log đi như sau

import { createStore, applyMiddleware, compose } from "redux";

import thunkMiddleware from "redux-thunk";

import rootReducer from "./combine-reducers";

const composeEnhancers = window.\_\_REDUX\_DEVTOOLS\_EXTENSION\_COMPOSE\_\_ || compose;

var store = createStore(

rootReducer,

composeEnhancers(applyMiddleware(thunkMiddleware))

);

export default store;

#### combine-reducers.js

Mỗi module sẽ có một biến trạng thái redux lưu trong kho. Cần đăng ký biến trạng thái đó trong file này (xem ví dụ viết module ở client)

### index.js app.js và app.css

Là 2 file entry point của front-end.

Trong file index.js, khai báo sử dụng redux và tiện ích đa ngôn ngữ trong gói react-redux-multilingual

Trong file app.js, khai báo các tiện ích với socket, firebase, và toastify.

Ngoài style css của template trong thư mục public, css của toàn bộ hệ thống định nghĩa trong file app.css. Những style chung phải được khai báo ở đây và chú thích rõ ràng.

Ngoài ra, style riêng cho từng module sẽ được khai báo trong mỗi module. Chú ý là dù khai báo riêng trong từng module, và người dùng đang truy cập ở module khác, các style này vẫn được áp dụng vào module đang truy cập.

### Các file khác

Cuối cùng là các file common khác như js, css, .env

**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.

## Cấu trúc thư mục server

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

### Index.js

Khi chạy server nodejs thì sẽ chạy file này đầu tiên, bên trong cấu hình khởi động server và combine các router api của các module

### Helpers

Tương tự như ở client , thư mục này gồm các file chứa các hàm tiện ích, các file quan trọng gồm có:

**Config.js:** file này định nghĩa các ROLE\_TYPES, LINKS, COMPONENTS, ROOT\_ROLES, LINK\_CATEGORY phục vụ cho việc tạo dữ liệu (seed DB) trên hệ thống, cũng như tạo các giá trị hằng.

**dbHelper.js:** chứa các hàm như connect()để kiểm tra và chuyển kết nối cơ sở dữ liệu tới **db** tương ứng, dựa vào tham số **portal**, hàm initModels()dùng để khởi tạo **model** nếu chưa tồn tại, và các hàm backup và restore dữ liệu.

Và còn nhiều hàm tiện ích khác, phục vụ cho việc xử lý chuỗi, mảng, …

### Log

Thư mục này gồm 2 thành phần:

* File /log/index.js: chứa hàm tiện ích ghi log. Log được ghi lại khi thực hiện mọi truy vấn đến api, được gọi trong file controller của các module
* Thư mục history: chứa các log đã ghi. Toàn bộ hệ thống có thư mục ghi log riêng, và từng công ty có thư mục ghi log riêng

Ví dụ: thực hiện ghi log ở file controller của module document khi call api get document.

const Logger = require(`../../logs`); // import hàm

Để ghi log khi truy vấn **success** thì thêm đoạn sau vào phần try trong controler

await Logger.info(req.user.email, "get\_documents\_success", req.portal);

Để ghi log khi truy vấn **fail** thì thêm đoạn sau vào phần catch trong controler

exports.getDocuments = async (req, res) => {

try {

const documents = await DocumentServices.getDocuments(

req.portal,

req.query,

req.user.company.\_id,

req.currentRole

);

await Logger.info(req.user.email, "get\_documents", req.portal);

res.status(200).json({

success: true,

messages: ["get\_documents\_success"],

content: documents,

});

} catch (error) {

await Logger.error(req.user.email, "get\_documents", req.portal);

res.status(400).json({

success: false,

messages: Array.isArray(error) ? error : ["get\_documents\_faile"],

content: error,

});

}

};

### Middleware

Chứa các file xác thực trước khi chạy tới controller, gồm 2 file: index.js và servicePermission.js

#### servicePermission.js

**ServicePermission.js:** file cấu hình các link, xem trang nào được truy cập những api nào.

#### index.js

File này giúp xác thực người dùng từ request gửi đến. Tiến hành 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.

**Index.js:**trong file cóhàm **authFunc()** là một **middleware** kiểm tra:

**1. Người dùng đã xác thực, jwt hợp lệ hay ko:**

const token = req.header("utk"); //JWT nhận từ người dùng

if (!token) throw ["access\_denied"]; //Kiểm tra nếu không có token thì

từ chối truy cập api.

Như giới thiệu ở phần cấu trúc client, client sẽ mã hóa crtp, crtr, fgp. Do đó khi nhận request gửi từ client, server phải giải mã:

let crtp, crtr, fgp;

if (process.env.DEVELOPMENT === "true") {

    crtp = req.header("crtp");

    crtr = req.header("crtr");

    fgp = req.header("fgp");

} else {

    crtp = decryptMessage(req.header("crtp"));

    crtr = decryptMessage(req.header("crtr"));

    fgp = decryptMessage(req.header("fgp"));

}

Việc client phải mã hóa trước khi gửi cho server gây ra bất tiện cho việc test api. Do vậy, ta có thể bật hoặc tắt giải mã theo mong muốn. Nếu muốn dùng postman (hoặc các phần mềm test api khác) thì set biến (và phải khởi động lại server)

DEVELOPMENT=true //trong file .env thư mục server

Sau khi nhận token thì tiến hành giải mã

let verified;

    try {

    verified = await jwt.verify(token, process.env.TOKEN\_SECRET);

    } catch (error) {

    throw ["access\_denied"];

}

**2. Xác định db truy vấn cho request**

req.portal = !req.user.company

                ? process.env.DB\_NAME

                : req.user.company.shortName;

Hệ thống cung cấp dịch vụ chuyển đổi số cho doanh nghiệp. Mỗi doanh nghiệp sẽ có database riêng.

Nếu **user** không có **company** thì **port** mặc định lấy theo **DB\_Name** cấu hình trong file **.env**, ngược lại **user** có **company** thì **portal** chính là tên viết tắt của công ty (shortName).

Sau đó khởi tạo Model nếu chưa có:

initModels(connect(DB\_CONNECTION, req.portal), Models);

Tiếp tục kiểm tra role của người dùng có hợp lệ hay không:

const currentRole = crtr; // role hiện tại của người dùng

if (!ObjectId.isValid(currentRole)) {

   throw ["role\_invalid"]; //trả về lỗi nếu current role là một giá trị

 không xác định

}

   req.currentRole = currentRole;

   const role = await Role(connect(DB\_CONNECTION, req.portal))

.findById(currentRole); //current role của người dùng

if (role === null) throw ["role\_invalid"];

Sau đó, kiểm tra xem **fingerPrint** có hợp lệ hay không:

//chữ ký của trình duyệt người dùng

const fingerprint = fgp;

if (verified.fingerprint !== fingerprint)

    throw ["fingerprint\_invalid"];

Kiểm tra xem current role có đúng là của người dùng hay không?

const userId = req.user.\_id;

const userrole = await UserRole(connect(DB\_CONNECTION, req.portal))

    .findOne({ userId, roleId: role.\_id });

if (userrole === null)

   throw ["user\_role\_invalid"];

   /\*\*

   \* Riêng đối với system admin của hệ thống thì bỏ qua bước này

   \*/

  if (role.name !== "System Admin") {

   /\*\*

   \* Kiểm tra công ty của người dùng có đang được kích hoạt hay không?

   \*/

    const company = await Company(connect(DB\_CONNECTION, process.env.DB\_NAME))

          .findById(req.user.company.\_id);

    if (!company.active) {

        //dịch vụ của công ty người dùng đã tạm dừng

        const resetUser = await User(connect(DB\_CONNECTION, req.portal))

            .findById(req.user.\_id);

        resetUser.tokens = [];

        //đăng xuất tất cả các phiên đăng nhập của người dùng khỏi hệ thống

        await resetUser.save();

            throw ["service\_off"];

    }

}

Kiểm tra xem current-role của người dùng có được phép truy cập vào trang này hay không?

const url = crtp;

const device = req.header("device");

if (!device) {

    if (checkPage) {

        const link = role.name !== "System Admin" ?

            await Link(connect(DB\_CONNECTION, req.portal))

.findOne({ url, deleteSoft: false }) :

            await Link(connect(DB\_CONNECTION, req.portal)).findOne({ url });

        if (link === null) throw ["url\_invalid"];

        const roleArr = [role.\_id].concat(role.parents);

        const privilege = await Privilege(connect(DB\_CONNECTION, req.portal)).findOne({

            resourceId: link.\_id,

            resourceType: "Link",

            roleId: {

                $in: roleArr,

            },

        });

        if (privilege === null) throw ["page\_access\_denied"];

    }

    /\*\*

    \* Kiểm tra xem user này có được gọi tới service này hay không?

    \*/

    const apiCalled = req.route.path !== "/" ? req.baseUrl + req.route.path  : req.baseUrl;

    const perLink = links.find(l => l.url === url);

    if (!perLink) throw ['url\_invalid\_permission']

    if (perLink.apis[0] !== '@all') {

        const perAPI = perLink.apis.some(api => api.path === apiCalled

 && api.method === req.method);

        if (!perAPI) throw ['api\_permission\_invalid'];

    }

}

### Model

Thư mục chứa các MongoDB collections của từng module, sử dụng Mongoose

### Modules

Thư mục này gồm các thư mục con là tên của từng module. Nếu module to, lại chia tiếp thành các module con là các thư mục con của thư mục module.

Mỗi module gồm 4 file (với moduleName là tên của module trong hệ thống: document, task, …)

README.txt: file mô tả về chức năng của module, cấu trúc của module và sự liên quan của module với các module khác

modulename.route.js**:** định nghĩa các đường định tuyến cho các của hệ thống mà người dùng gọi ở ứng dụng front-end để yêu cầu sử dụng dịch vụ. Có thể lồng thêm middleware nhằm thực hiện nhiệm vụ xác thực người dùng và lấy thêm các thông tin khác (như database, công ty của người dùng, …) trước khi request được chuyển tiếp sang cho controller.

modulename.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).

modulename.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

Cách thiết kế API/service phải theo đúng chuẩn như hướng dẫn trong mục 3.1.1

Trong thư mục modules có chứa sẵn 1 thư mục con là “ *\_sample-module “* – đây là một thư mục module mẫu. Trong “\_sample-module” tạo sẵn 4 file nói trên.

### Seed

Chứa các file tạo dữ liệu mẫu cho hệ thống, nhằm tiện cho dev và test, không cần mất công tạo lại dữ liệu từ đầu

Một số file seed đã tạo:

* initDB.js: Bắt buộc phải chạy seed này để khởi tạo hệ thống (tạo account người quản trị hệ thống và các tài nguyên khác như các trang, các component, các role). Các seed khác không bắt buộc
* …

### Upload

Thư mục lưu trữ các loại file mà client gửi lên server

### Package.json

Khai báo các thư viện cần được cài đặt cho backend. Đồng thời định nghĩa một số lệnh chạy từ command line (build server với webpack để tinh gọn và che giấu code, seed dữ liệu, chạy server)

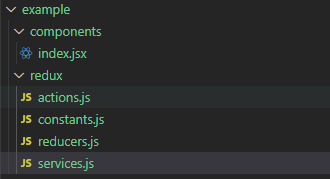
## Cách tạo mới một module

### Mã nguồn Client

#### Bước 1 - Tạo mã nguồn Redux

Vào thư mục modules (client/src/modules) tạo 1 folder.Ví dụ: module có tên **example**

Tiếp tục truy cập vào folder **example** tạo thêm 2 thư mục con có tên là **components** và **redux.** Và tạo các file như hình dưới



Giả sử trong file index.jsx cần call 1 api lấy tất cả các đối tượng dữ liệu example, có thể cài đặt mã nguồn như sau:

File **constants.js**  thêm đoạn code dưới đây:

export const exampleConstants = {

    GET\_ALL\_EXAMPLES\_REQUEST: "GET\_ALL\_EXAMPLES\_REQUEST",

    GET\_ALL\_EXAMPLES\_SUCCESS: "GET\_ALL\_EXAMPLES\_SUCCESS",

    GET\_ALL\_EXAMPLES\_FAILURE: "GET\_ALL\_EXAMPLES\_FAILURE",

}

File **service.js** thêm code như dưới:

import { sendRequest } from '../../../../helpers/requestHelper';

// tùy vào cấp độ thư mục mà chỉnh lại đường dẫn tương đối cho đúng

export const exampleServices = {

    getExamples,

}

function getExamples(queryData) {

    return sendRequest(

        {

            url: `${process.env.REACT\_APP\_SERVER}/examples`,

            method: "GET",

            params: {

                exampleName: queryData !== undefined ?

 queryData.exampleName : "",

                page: queryData !== undefined ? queryData.page : null,

                limit: queryData !== undefined ? queryData.limit : null

            }

        },

false, // Nếu có truy vấn thành công thì không hiện thông báo

true, // Nếu có truy vấn thất bại thì hiện thông báo

"manage\_example"

    );

}

File **Actions.js**  thêm đoạn code như sau:

import { exampleConstants } from './constants';

import { exampleServices } from './services';

export const exampleActions = {

    getExamples

}

function getExamples(queryData) {

    return (dispatch) => {

        dispatch({

            type: exampleConstants.GET\_ALL\_EXAMPLES\_REQUEST

        });

        exampleServices

            .getExamples(queryData)

            .then((res) => {

                dispatch({

                    type: exampleConstants.GET\_ALL\_EXAMPLES\_SUCCESS,

                    payload: res.data.content

                });

            })

            .catch((error) => {

                dispatch({

                    type: exampleConstants.GET\_ALL\_EXAMPLES\_FAILURE,

                    error

                });

            });

    }

}

File **reducer.js** thêm đoạn code sau:

import { exampleConstants } from './constants';

const initialState = {

    lists: [],

    isLoading: true,

}

export function examples(state = initialState, action) {

switch (action.type) {

case exampleConstants.GET\_ALL\_EXAMPLES\_REQUEST:

return {

                ...state,

                isLoading: true

            }

case exampleConstants.GET\_ALL\_EXAMPLES\_FAILURE:

return {

                ...state,

                isLoading: false,

                error: action.error

            }

case exampleConstants.GET\_ALL\_EXAMPLES\_SUCCESS:

return {

                ...state,

                lists: action.payload.data,

                isLoading: false

            }

default:

             return state

}

}

#### Bước 2 - Cấu hình sidebar

Truy cập vào thư mục sidebar (client/src/layout/sidebar/components) và mở file sidebar.jsx

**TH1: Nếu muốn thêm 1 trang đơn:**



Sử dụng component Item để tạo trang:

{/\* example \*/ }

<Item

    item={{

        name: "menu.manage\_examples ", // tên trang

        path: "/example", // đường dẫn của trang

        icon: "fa fa-newspaper-o",

    }}

/>

Components **Item** nhận vào từ props item 1 object cấu hình gồm:

**name** là tên hiển thị trên sidebar, “menu.example” là tên được cấu hình trong file lang/vn.js và en.js

menu: {

home: "Trang chủ",

example: "ví dụ tạo mới 1 trang",

….,}

**Path** là đường dẫn của trang trên url

**Icon** là biểu tượng hiển thị bên trái tên trang, giá trị truyền vào là class của icon (icon font awesome v4\*)

**TH2: Nếu muốn thêm vào sidebar các trang dạng dropdown**



Sử dụng component GroupItem để tạo trang:

<GroupItem

    groupItem={{

        name: "menu.manage\_examples",

        icon: " fa fa-newspaper-o",

        list: [

            {

                name: "menu.manage\_examples\_1",

                icon: "fa fa-circle",

                path: "/manage-examples-1",

            },

            {

                name: "menu.manage\_examples\_hooks\_1",

                icon: "fa fa-circle",

                path: "/manage-examples-2",

            }

        ],

    }}

/>

Tương tự như ở trường hợp 1, tên trang cấu hình trong file vn.js và en.js

#### Bước 3 - Định tuyến trang

Mở file routes.js (client/src/react-routes/routes.js) khai báo, tạo định tuyến truy cập các trang vừa tạo ở bước 2.

Import component example tạo ở bước 1

import Example from "../modules/example/components";

Dùng component privateRoute để tạo route:

Với **trường hợp 1 ở bước 2** thì cấu hình privateRoute như sau:

{/\* example \*/ }

<PrivateRoute

    isLoading={false}

    key={"example "}

    arrPage={[

        { link: "/", name: "home", icon: "fa fa-home" },

        {

            link: "/example", // url trang

            name: "example.name", // tên trang

            icon: " fa fa-newspaper-o ",

        },

    ]}

    auth={auth}

    exact={true}

    link={"/example"}

    path={"/example"}

    pageName={"example.name"}

    layout={Layout}

    component={Example} // component ứng với trang, tạo ở bước 1

/>

#### Bước 4 - Khai báo redux

Mở file combine-reducers.js (client/src/redux/ combine-reducers.js)

Import file **reducer** tạo ở bước 1, và thêm vào **combineReducers**

import { examples } from "../modules/example/redux/reducers";

const appReducer = combineReducers({

    examples, // thêm reducer đã import vào đây

})

### Mã nguồn Server

#### Bước 1 - Thêm trang vào kho trang, phân quyền cho trang

Mở file config.js (qlcv/server/helpers/config.js)

Thêm category cho module:

const LINK\_CATEGORY = {

    EXAMPLE: {

        name: "examples-management",

        description: "Quản lý ví dụ"

    },

}

Thêm cấu hình url, component, role nào được truy cập vào trang “/example”

const LINKS = [{

    url: '/example',

    description: `module example`,

    category: LINK\_CATEGORY.EXAMPLE.name,

    roles: [ // những role truy cập dc link

        ROOT\_ROLES.SUPER\_ADMIN.name,

        ROOT\_ROLES.ADMIN.name,

        ROOT\_ROLES.MANAGER.name,

        ROOT\_ROLES.DEPUTY\_MANAGER.name,

        ROOT\_ROLES.EMPLOYEE.name

    ],

    components: getComponentsInLink('/example')

}]

#### Bước 2 - Đăng ký các API mà trang được dùng

**Mở file servicesPermission.js (qlcv/server/middleware/)**

Thêm đoạn code sau vào links

exports.links = [

    {

        url: '/example', // trang muốn tạo

        apis:['@all']  // những api nào được gọi trên trang này @all là tất cả api được dùng trên trang này

    },

]

#### Bước 3 - Seed lại DB

Tiến hành chạy lại seed bằng cách cd vào thư mục server và chạy *npm run init*

Sau khi thực hiện 3 bước trên thì trang **/example** sẽ có thể truy cập được, và sidebar hiển thị trang vừa tạo

#### Bước 4 - Tạo model cho module example:

Truy cập vào thư mục models (qlcv/server/models) tạo một folder có tên là example, sau đó tạo 1 file example.model.js và thêm code như sau để tạo được một collecttion ‘Example’:

const mongoose = require('mongoose');

const Schema = mongoose.Schema;

const ExampleSchema = new Schema({

    exampleName: { // Tên Ví dụ

        type: String,

        required: true

    },

    description: { // Mô tả Ví dụ

        type: String

    }

});

module.exports = (db) => {

    if (!db.models.Example)

        return db.model('Example', ExampleSchema);

    return db.models.Example;

}

#### Bước 5 - Code nghiệp vụ

Tạo một folder có tên example trong thư mục qlcv/server/modules và tạo lần lượt 3 file trong folder example:example.route.js, example.controller.js, và example.service.js

##### Route

**example.route.js:**  khai báo 1 route có chức năng lấy tất cả example. Chức năng này chỉ được gọi bởi tài khoản đã được xác thực, nên ta dùng middleware auth.

const express = require('express');

const router = express.Router();

const ExampleController = require('./example.controller');

const { auth } = require(`../../middleware`);

router.get('/', auth, ExampleController.getExamples);

##### Controller

**example.controller.js:** khai báo 1 controller có chức năng xử lý lời gọi api để lấy tất cả example, trong file example.controller.js gọi đến service **getExamples**. 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.

Quy định dữ liệu trả về client gồm có **3** trường: **success** (giá trị **true** hoặc **false**), **messages** (mã trả về cho client), **content** (nội dung trả về).

Mỗi truy vấn, dù gọi thành công hay thất bại thì vẫn được ghi lại log:

await Log.info(req.user.email, "GET\_ALL\_EXAMPLES\_SUCCESS",

req.portal); // ghi log nếu thành công

await Log.error(req.user.email, "GET\_ALL\_EXAMPLES\_FAILE", req.portal); // ghi log nếu thất bại

const ExampleService = require('./example.service');

const Log = require(`../../logs`);

exports.getExamples = async (req, res) => {

    try {

        let { page, limit, exampleName } = req.query;

        let data;

        let params;

        if (page === undefined || limit === undefined) {

            params = {

                exampleName: exampleName,

                page: 0,

                limit: 10

            }

            data = await ExampleService.getExamples(params, req.portal);

        } else {

            params = {

                exampleName: exampleName,

                page: Number(page),

                limit: Number(limit)

            }

            data = await ExampleService.getExamples(params, req.portal);

        }

        await Log.info(req.user.email, "GET\_ALL\_EXAMPLES\_SUCCESS",

 req.portal);

        res.status(200).json({

            success: true,

            messages: ["get\_all\_examples\_success"],

            content: data

        });

    } catch (error) {

        await Log.error(req.user.email, "GET\_ALL\_EXAMPLES\_FAILE", req.portal);

        res.status(400).json({

            success: false,

            messages: ["get\_all\_examples\_fail"],

            content: error.message

        });

    }

}

##### Service

**example.service.js:** khai báo 1 service có chức năng xử lý, tương tác với database để lấy tất cả example, tạo hàm getExamples nhận đầu vào là portal của công ty, và các params (limit, …).

Cụ thể:

connect(DB\_CONNECTION, portal) : hàm này sẽ thực hiện kết nối truy vấn tới db,

tham số portal sẽ quyết định truy vấn trên db nào, vì mỗi công ty sẽ có các db độc lập với nhau (thiết kế theo hệ thống **Multi**-**Tenant**).

Hàm countDocuments() sẽ đếm tống số ví dụ mà tìm được theo các điều kiện trong keySearch, phục vụ cho việc phân trang phía client

Hàm find() sẽ tìm kiếm tất cả nhũng ví dụ trong collection Example với portal mà user đang có, và KeySearch là điều kiện tìm kiếm (Nếu có, ví dụ: tìm kiếm example có tên là abc thì exampleName = abc).

const {Example} = require('../../models');

const {connect} = require(`../../helpers/dbHelper`);

exports.getExamples = async (params, portal) => {

    let keySearch;

    if (params.exampleName !== undefined && params.exampleName.length !== 0) {

        keySearch = {

            ...keySearch,

            exampleName: {

                $regex: params.exampleName,

                $options: "i"

            }

        }

    }

    let totalList = await Example(connect(DB\_CONNECTION, portal))

.countDocuments(keySearch);

    let ExampleCollection = await Example(connect(DB\_CONNECTION, portal))

.find(keySearch)

         .skip((params.page - 1) \* params.limit)

         .limit(params.limit);

    return { data: ExampleCollection, totalList }

}

#### Bước 6 - Khai báo Router cho service

Mở file index.js (qlcv/server/index.js) và khai báo 1 router (đường dẫn api cho module example require tới file route của module example)

router.use("/examples", require("./modules/example/example.route"));

## Giải thích module example

Trong project đã có sẵn module example dành cho các bạn mới join vào dự án. Module này thực hiện các thao tác CRUD với đối tượng example, gồm các trường name và description.

### Mã nguồn Client theo mô hình 1

Module example ở client có 2 folder là example1 và example 2, đều thực hiện các chác năng CRUD nhưng mô hình xử lý khác nhau

Khi cần hiển thị danh sách tất cả các Example, với mô hình **example1,** cách gọi api sẽ là: trong bảng **Example** có bao nhiêu trường thì sẽ lấy hết ra, ví vụ **name, description, creator, …**

=> Nếu bảng **Example** gồm rất nhiều trường (có những trường cần populate, chi phí cao), nhưng khi hiện thị, giao diện chỉ cần hiển thị một số trường trong đó, thì cách làm theo mô hình 1 này bị dư thừa dữ liệu, mất công lấy toàn bộ dữ liệu, ảnh hưởng tới hiệu năng.

Trong mô hình **example2**, việc truy vấn lấy danh sách dữ liệu và hiển thị ra sẽ khác với **example1**. **example2** cũng sẽ lấy danh sách các ví dụ ra, nhưng các trường thông tin không đầy đủ, ví dụ: bảng hiển thị chỉ cần cột **name** example thì chỉ gọi service trả về dữ liệu chỉ có trường **name.** Mục đích là tối ưu hiệu năng

Dưới đây ta xử lý theo mô hình 1

#### Code Redux

##### Redux constants.js

Đầu tiên, ta khởi tạo thư mục redux theo mô hình example1, với 3 file: constants, service, actions, và reducer

**File constants.js** định nghĩa tên cho các hành động, ví dụ lấy tất cả example và chỉnh sửa example.

export const exampleConstants = {

    GET\_ALL\_EXAMPLES\_REQUEST: "GET\_ALL\_EXAMPLES\_REQUEST",

    GET\_ALL\_EXAMPLES\_SUCCESS: "GET\_ALL\_EXAMPLES\_SUCCESS",

    GET\_ALL\_EXAMPLES\_FAILURE: "GET\_ALL\_EXAMPLES\_FAILURE",

    EDIT\_EXAMPLE\_REQUEST: "EDIT\_EXAMPLE\_REQUEST",

    EDIT\_EXAMPLE\_SUCCESS: "EDIT\_EXAMPLE\_SUCCESS",

    EDIT\_EXAMPLE\_FAILURE: "EDIT\_EXAMPLE\_FAILURE",

}

##### Redux service.js

**service.js** Thực hiện lời gọi tới api, sử dụng hàm sendRequest trong helpers như đã giới thiệu ở phần trước để thao tác với api

Trong hàm **getExamples** sử dụng tiện ích sendRequest với ý nghĩa các tham số như sau

* **url** là đường đẫn api
* **method** là phương thức thực hiện truy vấn
* **params** là các tham số được gửi kèm trên url api
* **false** (showSuccessAlert) là nếu truy vấn, server trả về thành công thì không cần hiển thị thông báo
* **true** (showFailAlert) là nếu truy vấn, server trả về lỗi thì hiện thông báo
* **module:** tên module, dùng để lấy các thông điệp theo ngôn ngữ Anh Việt ứng với mã kết quả trả về từ API

function getExamples(queryData) {

    return sendRequest(

        {

            url: `${process.env.REACT\_APP\_SERVER}/examples`,

            method: "GET",

            params: {

                exampleName: queryData !== undefined ? queryData.exampleName : "",

                page: queryData !== undefined ? queryData.page : null,

                limit: queryData !== undefined ? queryData.limit : null

            }

        },

        false,

        true,

        "manage\_example"

    );

}

function editExample(id, data) { // id của ví dụ

    return sendRequest(

        {

            url: `${process.env.REACT\_APP\_SERVER}/examples/${id}`,

            method: "PATCH",// phương thức

            data: data // dữ liệu mới sau được gửi trong body

        },

        true, // Thành công thì hiện thông báo

        true, // Thất bại cũng hiện thông báo

        "manage\_example"

    )

}

##### Redux actions.js

**File actions.js** định nghĩa các hành động cho các yêu cầu từ người dùng, như sửa, xóa example.

Hàm getExamples dưới đây khi lấy danh sách các example, có thể dispatch 3 hành động

Đầu tiên là hành động gửi request (GET\_ALL\_EXAMPLES\_REQUEST). Sau đó thực hiện gọi hàm getExamples ở file service để thực hiện call api, hàm getExamples trả về 1 promise.

Nếu gọi thành công, sẽ chạy vào **.then** sau đó dispatch một action có type là GET\_ALL\_EXAMPLES\_SUCCESS và payload là res.data.content (giá trị server trả về). Nếu gọi api thất bại, sẽ chạy vào .**catch**, rồi dispatch action có type là GET\_ALL\_EXAMPLES\_FAILURE và error mà server trả về. Lưu ý các dịch vụ trên server đã được chuẩn hóa, luôn trả về object có **3** trường: **success** (giá trị **true** hoặc **false**), **messages** (mã trả về cho client), **content** (nội dung trả về).

export const exampleActions = {

    getExamples,

    editExample

}

function getExamples(queryData) {

    return (dispatch) => {

        dispatch({

            type: exampleConstants.GET\_ALL\_EXAMPLES\_REQUEST

        });

        exampleServices

            .getExamples(queryData)

            .then((res) => {

                dispatch({

                    type: exampleConstants.GET\_ALL\_EXAMPLES\_SUCCESS,

                    payload: res.data.content

                });

            })

            .catch((error) => {

                dispatch({

                    type: exampleConstants.GET\_ALL\_EXAMPLES\_FAILURE,

                    error

                });

            });

    }

}

function editExample(id, data) {

    return (dispatch) => {

        dispatch({

            type: exampleConstants.EDIT\_EXAMPLE\_REQUEST

        });

        exampleServices

            .editExample(id, data)

            .then((res) => {

                dispatch({

                    type: exampleConstants.EDIT\_EXAMPLE\_SUCCESS,

                    payload: res.data.content

                });

            })

            .catch((error) => {

                dispatch({

                    type: exampleConstants.EDIT\_EXAMPLE\_FAILURE,

                    error

                });

            });

    }

}

##### Redux reducer.js

Khi hệ thống dispatch action có type REQUEST, sẽ chạy vào code trong reducer. Thiết lập isLoading = true là để xác nhận hệ thống đang trong quá trình gọi dịch vụ, giúp hiển trị icon loading xoay tròn khi bắt đầu lấy dữ liệu trên giao diện.

switch (action.type) {

        case exampleConstants.GET\_ALL\_EXAMPLES\_REQUEST:

        case exampleConstants.EDIT\_EXAMPLE\_REQUEST:

            return {

                ...state,

                isLoading: true

            }

}

Khi truy vấn thành công, sẽ chạy vào case type SUCCESS

**Đối với case exampleConstants.EDIT\_EXAMPLE\_SUCCESS:**

Ví dụ: Khi chỉnh sửa một example1, thì phải thay thế example1 với giá trị cũ trong state của redux và thay thế giá trị example1 mới, giá trị mới bây giờ là action.payload

case exampleConstants.GET\_ALL\_EXAMPLES\_SUCCESS:

    return {

        ...state,

        lists: action.payload.data, // Lưu dữ liệu trả về từ server vào

//biến lists

        totalList: action.payload.totalList, // lưu tống số bản ghi

        isLoading: false // set lại isLoading = false

    }

case exampleConstants.EDIT\_EXAMPLE\_SUCCESS:

    index = findIndex(state.lists, action.payload.\_id);

     if (index !== -1) {

        state.lists[index] = action.payload

   }

     return {

     ...state,

     isLoading: false

}

**Đối với case exampleConstants.CREATE\_EXAMPLE\_SUCCESS:**

Khi thực hiện truy vấn thành công dữ liệu trả về cần phải thêm vào state của redux

case exampleConstants.CREATE\_EXAMPLE\_SUCCESS:

    return {

        ...state,

        lists: [

            ...state.lists,// giữ state cũ,

            action.payload.example // add ví dụ mới thêm vào

        ],

        isLoading: false

    }

**Đối với case exampleConstants.DELETE\_EXAMPLE\_SUCCESS:**

action là dữ liệu server trả về, và chính là ví dụ muốn xóa,, vì thế để xóa khỏi redux thì thực hiện filter list state cũ, chỉ giữ lại những ví dụ khác với ví dụ cần xóa.

case exampleConstants.DELETE\_EXAMPLE\_SUCCESS:

    return {

        ...state,

        lists: state.lists.filter(example => (example.\_id !== action.payload.content.\_id)),

        isLoading: false

    }

Khi dispatch action có type = “GET\_ALL\_EXAMPLES\_FAILURE” gọi truy vấn thất bại, dùng …state để dữ lại state ban đầu.

case exampleConstants.GET\_ALL\_EXAMPLES\_FAILURE:

case exampleConstants.EDIT\_EXAMPLE\_FAILURE:

    return {

        ...state,

        isLoading: false,

        error: action.error

    }

Dù thành công hay thất bại, đều thiết lập isLoading = fasle, để xác nhận đã gọi xong service

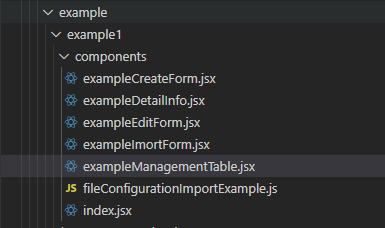
#### Tạo các components giao diện

Tạo thư mục chứa file xử lý giao diện components theo mô hình example1

Đối với project sử dụng Reactjs để viết thì có thể sử dụng **class component** hoặc **functional component**  để viết. Nên dùng functional Component.

##### Sử dụng class component

Mở thư mục **components** có đường dẫn như sau: qlcv/client/src/modules/example/example1/components



File exampleManagementTable.jsx là file giao diện có chức năng hiện thị danh sách các example theo dạng bảng.

**Để lấy danh sách example và hiển thị ra table theo mô hình example 1 làm như sau:**

Import các thư viện cần thiết: **withTranslate** (hỗ trợ song ngữ), exampleActions (action trong file redux, dùng để gọi service) , vì example1 dùng class component nên khai báo class như sau, khởi tạo page và limit mặc định để phân trang.

import React, { Component } from "react";

import { connect } from "react-redux";

import { withTranslate } from "react-redux-multilingual";

import { exampleActions } from "../redux/actions";

class ExampleManagementTable extends Component {

constructor(props) {

        super(props);

        this.state = {

            exampleName: "",

            page: 1,

            limit: 5,

        };

    }

}

Ở cuối file, hàm connect sẽ nhận vào 2 tham số:

function mapStateToProps(state) {

    const example = state.example1;

    return { example }

}

const mapDispatchToProps = {

    getExamples: exampleActions.getExamples

}

export default connect(mapStateToProps, mapDispatchToProps)(withTranslate(ExampleManagementTable));

Tham số thứ nhất **mapStateToProps** làm nhiệm vụ mapping global redux state vào props của component, ở đây chỉ cần dùng state **example** nên chỉ cần mapping state **example** vào props. Cần tránh mapping cả kho state của redux vào props của component, gây hiệu năng thấp. Trong project muốn sử dụng state của redux thì gọi thông qua **props,** ví dụ: **this.props.example.** Mỗi khi redux state **example** được cập nhật, component cũng cập nhật giá trị **example** của **props**.

Tham số thứ 2 **mapDispatchToProps**: để tiện gọi các hàm đã viết trong file action.js trong thư mục (example1/redux), nên mapping các hàm này vào **props.** **Component Example** sẽ gọi được service **getExamples** dễ dàng hơn

Sử dụng **withTranslate** bọc lấy tên class để có thể sử dụng song ngữ trên trang

Để gọi service lấy tất cả example, ta dùng **componentDidMount**, một life cycle của Reactjs để gọi hàm getExamples định nghĩa trong file service. Hàm này sẽ gọi api server tương ứng.

// gọi service lấy danh sách ví dụ

componentDidMount() {

let { exampleName, page, limit } = this.state;

this.props.getExamples({ exampleName, page, limit });

}

Sau khi gọi api, nếu thành công, server sẽ trả 1 mảng danh sách các ví dụ, được lưu vào biến **example** trong kho state redux. Biến này sẽ được lấy ra từ **props,** do ta dùng **mapStateToProps** như mô tảở trên

##### Tính năng song ngữ

Để sử dụng song ngữ trong component, dùng **translate** trong state của redux. Biến này cũng thông qua props.

Ví dụ lấy ra giá trị được cấu hình trong thư mục lang file en.js và vi.js: translate('manage\_example.exampleName')}

Để hiển thị ra table thì trong **return()** thêm đoạn code html để tạo giao diện và xử lý dữ liệu để hiển thị ra bảng

render() {

        const { example, translate } = this.props;

        // example giờ là mảng danh sách ví dụ

        return (

<div className="box-body qlcv">

     <table id="example1-table" className="table table-striped

table-bordered table-hover"

         <thead>

            <tr>

                <th className="col-fixed"

                    style={{ width: 60 }}>

                    {translate('manage\_example.index')}</th>

                <th>

                    {translate('manage\_example.exampleName')}

                </th>

                <th>

                    {translate('manage\_example.description')}

                </th>

                <th style={{

                    width: "120px",

                    textAlign: "center"

                }}>

                    {translate('table.action')}</th>

            </tr>

        </thead>

        <tbody>

{/\*  dùng map để hiển thị mảng danh sách ví dụ \*/}

            {(example && example.lists.length !== 0) &&

                example.lists.map((example, index) => (

                    <tr key={index}>

                        <td>{index + 1 + (page - 1) \* limit}</td>

                        <td>{example.exampleName}</td>

                        <td>{example.description}</td>

                        <td style={{ textAlign: "center" }}>

                            <a className="edit text-green"

                                style={{ width: '5px' }} title={translate('manage\_example.detail\_info\_example')} onClick={() => this.handleShowDetailInfo(example.\_id)}>

                                <i className="material-icons">visibility</i></a>

                            <a className="edit text-yellow"

                                style={{ width: '5px' }} title={translate('manage\_example.edit')}

                                onClick={() => this.handleEdit(example)}>

                                <i className="material-icons">edit</i></a>

                            <DeleteNotification

                                content={translate('manage\_example.delete')}

                                data={{

                                    id: example.\_id,

                                    info: example.exampleName

                                }}

                                func={this.props.deleteExample}

                            />

                        </td>

                    </tr>

                ))

            }

        </tbody>

    </table>

    {example.isLoading ?

        <div className="table-info-panel">{translate('confirm.loading')}</div> :

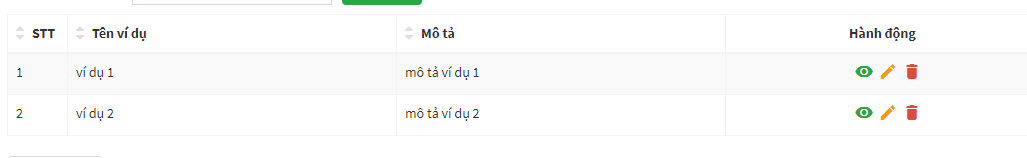
        (typeof lists === 'undefined' || lists.length === 0) && <div className="table-info-panel">{translate('confirm.no\_data')}</div>

    }

</div>

)}

Sau khi thực hiện các bước thì sẽ có giao diện như sau



##### Cấu hình ẩn hiện cột trong bảng

Bảng hiện ra có 4 cột. Nếu bảng có nhiều cột, có thể sử dụng tính năng cấu hình ẩn/hiện cột. Cấu hình sẽ được lưu vào local Storage để mỗi khi trang được tải lại, cấu hình vẫn giữ nguyên. Tương tự, có thể lưu cả cấu hình số bản ghi sẽ hiện thị lên bảng (limit). Các bước làm như sau.

Import component DataTableSetting và hàm tiện ích lưu cáu hình vào **localStorage**

import { DataTableSetting } from "../../../../common-components";

import { getTableConfiguration } from '../../../../../helpers/tableConfiguration';

Tiếp theo trong phần **constructor**, sửa lại như sau

constructor(props) {

    super(props);

    const tableId = "example1-table"; // khai bao id cho table

    const defaultConfig = { limit: 5 }

    const limit = getTableConfiguration(tableId, defaultConfig).limit;

    this.state = {

        exampleName: "",

        page: 1,

        limit: limit, // limit bây giờ sẽ là lấy trong localStorage

tableId

    };

}

Sửa id cho table từ id="example1-table" thành id = {this.state.tableId}

Thêm component DataTableSetting vừa import vào thẻ <th> cột action (hoặc để component ở vị trí bất kỳ nào khác, miễn là đảm bảo tính thẩm mỹ và chuẩn tắc). Component này nhận vào các thuộc tính như sau:

**tableId** trùng với id của **table** bước trên đã làm

**columnArr:** Mảng danh sách tên các cột của bảng

**setLimit:** Hàm xử lý gọi lại api lấy danh sách ví dụ với giới hạn số bản ghi mới

<th style={{ width: "120px", textAlign: "center" }}>

{translate('table.action')}

    <DataTableSetting

        tableId= {this.state.tableId}

        columnArr={[

            translate('manage\_example.index'),

            translate('manage\_example.exampleName'),

            translate('manage\_example.description'),

        ]}

        setLimit={this.setLimit}

    />

</th>

Hàm setLimit xử lý như sau:

setLimit = (number) => {

// number là giá trị limit mới mà người dùng muốn cấu hình

        const { exampleName, page } = this.state;

        this.setState(state => {

            return {

                ...state,

                limit: parseInt(number)

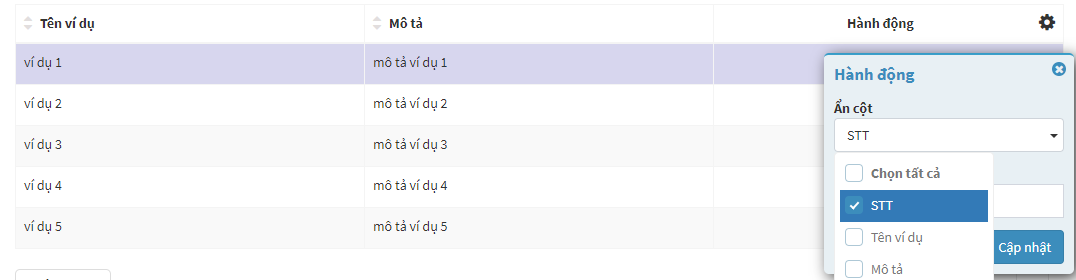
            }

        });

        this.props.getExamples({ exampleName, limit: parseInt(number), page });

}

Sau khi thực hiện, giao diện sẽ như hình dưới:



Trên giao diện sẽ xuất hiện bánh răng tròn, click vào sẽ hiển thị 1 modal cấu hình, nếu tick vào STT thì cột STT sẽ bị ẩn đi. Khi người dùng tải lại trang, hoặc đóng trình duyệt rồi mở lại, cấu hình vẫn được giữ nguyên.

##### Phân trang cho bảng

Khi sử dụng bảng để hiển thị danh sách dữ liệu, nếu số lượng dữ liệu nhiều, cần cài đặt phân trang, thực hiện như sau:

Import component PaginateBar, được viết sẵn để hỗ trợ việc hiển thị phân trang

import { PaginateBar } from "../../../../common-components";

Sau đó gọi component này dưới thẻ đóng của table (dưới </table>)

 <PaginateBar

      pageTotal={totalPage ? totalPage : 0}

      currentPage={page}

      display={lists && lists.length !== 0 && lists.length}

      total={example && example.totalList}

      func={this.setPage}

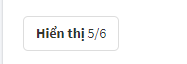
/>

**pageTotal**  là tổng số trang được tính bằng cách làm tròn giữa phép toán : tổng số bản ghi chia cho giới hạn mỗi trang

const totalPage = Math.ceil(example.totalList / limit);

**currentPage** là trang hiện tại, trên constructor để mặc định là ở trang 1

**display** giá trị truyền vào là số bản ghi đang được hiển thị (dùng để hiển thị số bản ghi / tổng số bản ghi),



**total** là tổng số bản ghi

**func** là hàm xử lý khi click next hoặc back trang, gọi đến hàm setPage như sau

setPage = (pageNumber) => {

    const { exampleName, limit } = this.state;

    this.setState(state => {

        return {

            ...state,

            page: parseInt(pageNumber)

        }

    });

    this.props.getExamples({ exampleName, limit, page: parseInt(pageNumber) });

}

Hàm setPage nhận vào là số page muốn xem, sau đó gọi service truy vấn để lấy ra danh sách bản ghi với số trang tương ứng

##### Cập nhật dữ liệu

Để chỉnh sửa 1 example và hiển thị ra table theo mô hình example 1 làm như sau

**Bước 1: Sửa file danh sách dữ liệu (exampleManagementTable)**

Import component edit example

import ExampleEditForm from "./exampleEditForm";

Khi click vào biểu tượng edit (bút chì) trong bảng danh sách ví dụ, hàm **handleEdit** được gọi.Vì mô hình số 1 lấy đầy đủ thông tin của dữ liệu, nên tham số example của hàm **handleEdit** là một ví dụ có đầy đủ thông tin như tên, mô tả, …

<a className="edit text-yellow"

    style={{ width: '5px' }}

    title={translate('manage\_example.edit')}

    onClick={() => this.handleEdit(example)}>

    <i className="material-icons">edit</i>

</a>

Hàm handleEdit xử lý như sau. Hàm này setState giá trị của một ví dụ vào biến currentRow sau đó mở modal edit bằng lệnh

window.$('#modal-edit-example').modal('show');

với ‘modal-edit-example' là id của modal chỉnh sửa ví dụ.

handleEdit = (example) => {

        this.setState((state) => {

            return {

                ...state,

                currentRow: example

            }

        });

        window.$('#modal-edit-example').modal('show');

    }

Sau đó, modal được mở ra, render ra component edit và truyền các thông tin của ví dụ như id, tên, mô tả, … thông qua **props**

{

    this.state.currentRow &&

        <ExampleEditForm

        exampleID={currentRow.\_id}

        exampleName={currentRow.exampleName}

        description={currentRow.description}

    />

}

**Bước 2: tạo file exampleEditForm.jsx**

Tương tự import các thư viện cần thiết

Sau khi người dùng click nút edit, thì component này sẽ nhận dược id, name, description thông qua **props**

static getDerivedStateFromProps = (nextProps, prevState) => {

        if (nextProps.exampleID !== prevState.exampleID) {

            return {

                ...prevState,

                exampleID: nextProps.exampleID,

                exampleName: nextProps.exampleName,

                description: nextProps.description,

                exampleNameError: {

                    message: undefined,

                    status: true

                }

            }

        } else {

            return null;

        }

    }

Sử dụng **getDerivedStateFromProps** để nhận các props từ component cha, và setState lại. đoạn lệnh if để tối ưu hiệu năng. Khi người dùng click edit ví dụ 1, sau đó tắt modal đi rồi lại vẫn click chọn edit ví dụ 1, điều kiện trong if không thỏa mãn, không cần cập nhật lại props của component, component sẽ không cần render lại, giúp tối ưu hiệu năng. Khi người dùng click edit ví dụ khác mới cần thiết lập prop mới và render lại.

Tiếp đến, xuống hàm render, truyền các giá trị như exampleName, description và value của thẻ input để hiển thị dữ liệu lên

render() {

        const { example, translate } = this.props;

        const { exampleName, exampleNameError, description } = this.state;

        return (

            <React.Fragment>

                <DialogModal

                    modalID={`modal-edit-example`}

isLoading={example.isLoading}

                    formID={`form-edit-example`}

                    title={translate('manage\_example.edit\_title')}

                    disableSubmit={!this.isFormValidated()}

                    func={this.save}

                    size={50}

                    maxWidth={500}

                >

                    <form id={`form-edit-example`}>

                        <div className={`form-group ${exampleNameError.status ? "" : "has-error"}`}>

                            <label>{translate('manage\_example.exampleName')}<span className="text-red">\*</span></label>

                            <input type="text" className="form-control" value={exampleName} onChange={this.handleExampleName} />

                            <ErrorLabel content={exampleNameError.message} />

                        </div>

                        <div className={`form-group`}>

                            <label>{translate('manage\_example.description')}</label>

                            <input type="text" className="form-control" value={description} onChange={this.handleExampleDescription} />

                        </div>

                    </form>

                </DialogModal>

            </React.Fragment>

        );

    }

Thẻ input lưu tên ví dụ sẽ có sự kiện onChange, mỗi khi người dùng thay đổi giá trị, giá trị mới sẽ được set vào state. Nếu giá trị bằng rỗng thì validateHelper sẽ trả ra message báo lỗi. Tương tự đối với thẻ input mô tả ví dụ

handleExampleName = (e) => {

    const { value } = e.target; // giá trị mới

    // xử lý validate

    let { translate } = this.props;

    let result = ValidationHelper.validateName(translate, value, 6, 255);

    this.setState(state => {

        return {

            ...state,

            exampleName: value, // exampleName gán bằng giá trị mới

            exampleNameError: result,

        }

    });

}

Khi người dùng bấm nút submit, gọi tới hàm save

save = () => {

        if (this.isFormValidated) {

            const { exampleID, exampleName, description } = this.state;

            this.props.editExample(exampleID, { exampleName, description });

        }

    }

Hàm save sẽ chạy khi không có lỗi, nếu **this.isFormValidated = true**  nghĩa là không có lỗi nhập liệu thì gọi service chỉnh sửa thông tin

##### Sử dụng function component (viết bằng React hook)

**Bước 1: mã nguồn xem danh sách dữ liệu**

Mở file exampleManagementTable.jsx nằm trong thư mục:

client/src/modules/example/example2/components-hook

Đàu tiên cần import thư viện cần thiết. Khác với class component, ở đây dùng useState (chức năng tương tự setState) và useEffect (tương tự như componentDidMount, shouldComponentUpdate,…)

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

import { connect } from "react-redux";

import { withTranslate } from "react-redux-multilingual";

import { exampleActions } from "../redux/actions";

import { DataTableSetting, PaginateBar } from "../../../../common-components";

Thay vì dùng class, bây giờ khai báo một arrow function, phần khai báo mapStateToProps và mapDispatchToProps, connect() tương tự như Hướng dẫn ở class Components.

Dùng useState để định nghĩa trước state

const ExampleManagementTable = (props) => {

const [state, setState] = useState({

        exampleName: "",

        description: "",

        page: 1,

        limit: 5

    })

}

Để gọi service lấy danh sách example sử dụng useEffect, **this** được loại bỏ trong react hook

useEffect(() => {

        let { exampleName, limit } = state;

        props.getOnlyExampleName({ exampleName, page, limit });

    }, [])

const { example, translate } = props;

Phần code giao diện html để hiển thị dữ liệu, phân trang, cấu hình bảng để trong return(), tương tự như cách viết ở class components phần trên `

**Bước 2: mã nguồn sửa dữ liệu**

Mở file exampleEditForm.jsx nằm trong thư mục:

client/src/modules/example/example2/components-hook

Khai báo biến state gồm các giá thuộc tính exampleID, exampleName,….

const [state, setState] = useState({

exampleID: undefined,

        exampleName: "",

        description: "",

        exampleNameError: {

            message: undefined,

            status: true

        }

    })

Kiểm tra nếu Id của example click chỉnh sửa lần trước với lần click tiếp theo, nếu 2 example khác nhau thì mới setState, nhằm tăng hiệu năng, tránh render lại

const { exampleName, description, exampleNameError, exampleID } = state;

    // setState từ props mới

    if (props.exampleID !== exampleID) {

        setState({

            ...state,

            exampleID: props.exampleID,

            exampleName: props.exampleName,

            description: props.description,

            exampleNameError: {

                message: undefined,

                status: true

            }

        })

    }

Sau đó truyền các giá trị vừa setState vào value của các thẻ input để hiển thị dữ liệu lên form

return (

    <React.Fragment>

        <DialogModal

            modalID={`modal-edit-example-hooks`} isLoading={example.isLoading}

            formID={`form-edit-example-hooks`}

            title={translate('manage\_example.edit\_title')}

            disableSubmit={!isFormValidated}

            func={save}

            size={50}

            maxWidth={500}

        >

            <form id={`form-edit-example-hooks`}>

                <div className={`form-group ${exampleNameError ? "" : "has-error"}`}>

                    <label>{translate('manage\_example.exampleName')}<span className="text-red">\*</span></label>

                    <input type="text" className="form-control" value={exampleName} onChange={handleExampleName} />

                    <ErrorLabel content={exampleNameError.message} />

                </div>

                <div className={`form-group`}>

                    <label>{translate('manage\_example.description')}</label>

                    <input type="text" className="form-control" value={description} onChange={handleExampleDescription} />

                </div>

            </form>

        </DialogModal>

    </React.Fragment>

);

Thẻ input có bắt sự kiện **onChange** để nếu người dùng nhập giá trị mới thì sẽ setState lại sau đó gọi truy vấn lưu vào database

Ví dụ khi tên ví dụ người dùng thay đổi thì sẽ gọi hàm **handleExampleName** để lưu lại giá trị mới

const handleExampleName = (e) => {

        const { value } = e.target;

        let result = ValidationHelper.validateName(translate, value, 6, 255);

        setState({

            ...state,

            exampleName: value,

            exampleNameError: result

        });

    }

Nếu không có lỗi validate thì hàm lưu sẽ gọi service editExample cập nhật giá trị mới của ví dụ vào db

const save = () => {

        if (isFormValidated) {

            props.editExample(exampleID, { exampleName, description });

        }

    }

### Mô hình Example2

Ví dụ: Truy vấn lấy danh sách tất cả các ví dụ.

Project cũng dc viết theo 2 cách theo dạng **class components** và **function components**

#### Sử dụng class components

##### Bước 1: mã nguồn lấy danh sách các ví dụ theo mô hình example 2

Mở file exampleManagementTable.jsx nằm trong folder example/example2/components

Ý nghĩa cấu trúc file tương tự như ở example1. Khác biệt là ta sẽ dùng service lấy danh sách example mà chỉ lấy tên ví dụ, bỏ qua các trường khác như trường mô tả

const mapDispatchToProps = {

    getOnlyExampleName: exampleActions.getOnlyExampleName

}

Ở **componentDidMount** gọi service **getOnlyExampleName**

 componentDidMount() {

        let { exampleName, page, limit } = this.state;

        this.props.getOnlyExampleName({ exampleName, page, limit });

 }

Nếu thành công dữ liệu trả về, tiến hành map để hiển thị dữ liệu ra bảng,

Các chức năng, phân trang, hiện thị cấu hình bảng (ẩn hiện cột) thì tương tự như ở example 1, có thể đọc lại phần trên

##### Bước 2: liên kết với component xem ví dụ

Tiến hành import component ExampleEditForm

import ExampleEditForm from "./exampleEditForm";

Khi người dùng click vào nút con mắt, sẽ chạy hàm **handleShowDetailInfo** nhận tham số là **id** của ví dụ đã click vào .

<a className="edit text-green"

    style={{ width: '5px' }}

    title={translate('manage\_example.detail\_info\_example')}

    onClick={() => this.handleShowDetailInfo(example.\_id)}>

    <i className="material-icons">visibility</i>

</a>

Hàm **handleShowDetailInfo** là một hàm async. Sau khi set id ví dụ vào state thì mới hiển thị form xem chi tiết lên.

handleShowDetailInfo = async (id) => {

        await this.setState((state) => {

            return {

                ...state,

                exampleId: id

            }

        });

        window.$(`#modal-detail-info-example`).modal('show');

    }

Sau khi setState xong, exampleId sẽ có giá trị.Tatruyền **exampleId** là **id** của ví dụ cần xem qua component **ExampleDetailInfo** thông qua props.

{

    <ExampleDetailInfo

     exampleId={this.state.exampleId}

    />

}

##### Bước 3: mã nguồn xem ví dụ

Mở file exampleDetailInfo.jsx nằm trong folder example/example2/components

Sau khi nhận được Id của ví dụ cần hiển thị chi tiết, tiến hành gọi service lấy danh sách ví dụ. Service này trả về đầy đủ thông tin của ví dụ

 shouldComponentUpdate(nextProps) {

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

            this.props.getExampleDetail(nextProps.exampleId);

            return false;

        }

        return true;

    }

Sau khi gọi thành công thì lấy dữ liệu ra,

if (this.props.example.currentDetailExample) {

      currentDetailExample = this.props.example.currentDetailExample;

}

Và sử dụng **map** để hiển thị thông tin của example ra components.

#### Sử dụng function components

##### Bước 1: mã nguồn danh sách dữ liệu

Tương tự như example1, file exampleManagementTable.jsx trong thư mục example/example2/components-hooks

import các thư viện cần thiết

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

import { connect } from "react-redux";

import { withTranslate } from "react-redux-multilingual";

import { exampleActions } from "../redux/actions";

Khai báo component **ExampleManagementTable** dưới dạng function

const ExampleManagementTable = (props) => {

    const [state, setState] = useState({

        exampleName: "",

        description: "",

        page: 1,

        limit: 5

    })

Để gọi service lấy danh sách example (lấy 1 phần dữ liệu), sử dụng useEffec. Do tham số thứ 2 của useEffect là một mảng rỗng, hàm này tương đương với componentDidMount trong class component và chỉ chạy 1 lần.

useEffect(() => {

        let { exampleName, limit } = state;

        props.getOnlyExampleName({ exampleName, page, limit });

    }, [])

##### Bước 2: liên kết với component xem ví dụ

Khi người dùng click xem chi tiết ví dụ, gọi hàm handleShowDetailInfo

<a className="edit text-green"

    style={{ width: '5px' }}

    title={translate('manage\_example.detail\_info\_example')}

    onClick={() => handleShowDetailInfo(example.\_id)}>

    <i className="material-icons">visibility</i>

</a>

Hàm handleShowDetailInfo nhận đầu vào là id của ví dụ cần xem, và lưu vào state, sau đó mở modal xem chi tiết với ‘modal-detail-info-example’ là id của modal chứa form Xem chi tiết ví dụ.

const handleShowDetailInfo = (id) => {

        setState({

            ...state,

            exampleId: id

        });

        window.$(`#modal-detail-info-example`).modal('show');

    }

Sau khi lưu id vào State thì truyền id này qua component thông qua props

return (

    <React.Fragment>

        {

            <ExampleDetailInfo

                exampleId={state.exampleId}

            />

        }

)

##### Bước 3: mã nguồn sửa ví dụ

Mở component ExampleDetailInfo nằm trong thư mục example/example2/components-hooks

Sau khi nhận id từ component ExampleManagementTable, trong useEffect gọi tới service getExampleDetail để lấy danh sách ví dụ với đầy đủ các trường thông tin của ví dụ. Tham số thứ 2 của useEffect là id của ví dụ cần xem. Cách truyền này giúp truy vấn được gọi lại khi id thay đổi

useEffect(() => {

        props.exampleId && props.getExampleDetail(props.exampleId);

    }, [props.exampleId])

Khi gọi thành công, tiến hành hiển thị thông tin danh sách ví dụ lên form xem chi tiết

const { translate, example } = props;

let currentDetailExample = {};

if (example.currentDetailExample) {

    currentDetailExample = example.currentDetailExample;

}

return (

    <React.Fragment>

        <DialogModal

            modalID={`modal-detail-info-example`} isLoading={example.isLoading}

            title={translate('manage\_example.detail\_info\_example')}

            formID={`form-detail-example`}

            size={50}

            maxWidth={500}

            hasSaveButton={false}

            hasNote={false}

        >

            <form id={`form-detail-example`}>

                <div className={`form-group`}>

                    <label>{translate('manage\_example.exampleName')}</label>

                    <p>{currentDetailExample.exampleName}</p>

                </div>

                <div className={`form-group`}>

                    <label>{translate('manage\_example.description')}</label>

                    <p>{currentDetailExample.description ? currentDetailExample.description : "No description"}</p>

                </div>

            </form>

        </DialogModal>

    </React.Fragment>

);

### Mã nguồn Server

Trong thư mục server, mở file example.route.js (server/modules/example)

File này định nghĩa các route thực hiện các thao tác CRUD, mọi service đều phải chạy qua **middleware auth**

const express = require('express');

const router = express.Router();

const ExampleController = require('./example.controller');

const { auth } = require(`../../middleware`);

router.get('/', auth, ExampleController.getExamples);

router.get('/getOnlyExampleName', auth, ExampleController.getOnlyExampleName);

router.get('/:id', auth, ExampleController.getExampleById);

router.post('/', auth, ExampleController.createExample);

router.patch('/:id', auth, ExampleController.editExample);

router.delete('/:id', auth, ExampleController.deleteExample);

Đối với api lấy tất cả example (**getExamples**) xem mục 5.3.2.5

Các controller module example đều đặt trong khối try catch. Lời gọi service xử lý dịch vụ nằm trong try, nếu xử lý truy vấn thành công, trả về status 200 và dữ liệu cho phía client, nếu xử lý truy vấn thất bại, trả về status 400 và trả về mã lỗi cho client.

Mở file example.service.js đường dẫn (server/modules/example)

Service getOnlyExampleName dùng để lấy 1 phần thông tin của ví dụ.

Truy vấn vào collection Example, sử dụng hàm **connect**  để kết nối tới portal ứng với user

keySeach là điều kiện truy vấn nếu có, hàm **countDocuments** để lấy tổng số bản ghi sau khi truy vấn, hàm **find()** dùng để tìm kiếm các bản ghi thỏa mãn điều kiện trong **keySearch**, và tham số thứ 2 của hàm **find() { exampleName: 1 }** dùng để chỉ định lấy trường exampleName, các trường còn lại không được lấy. **Skip()** dùng để nhảy, bỏ qua số bản ghi, và **Limit**() dùng để giới hạn số bản ghi trả về

exports.getOnlyExampleName = async (params, portal) => {

    let keySearch;

    if (params.exampleName !== undefined && params.exampleName.length !== 0) {

        keySearch = {

            ...keySearch,

            exampleName: {

                $regex: params.exampleName,

                $options: "i"

            }

        }

    }

    let totalList = await Example(connect(DB\_CONNECTION, portal))

.countDocuments(keySearch);

    let ExampleCollection = await Example(connect(DB\_CONNECTION, portal))

.find(keySearch, { exampleName: 1 })

        .skip(params.page \* params.limit)

        .limit(params.limit);

    return { data: ExampleCollection, totalList }

}

Service getExampleById, dùng để lấy thông tin của ví dụ dựa vào id, phục vụ cho chức năng edit, và chức năng xem chi tiết của mô hình số 2.

Tương tự, Sau khi kết nối vào Model Example, tiến hành dùng hàm findById để tìm kiếm bản ghi theo id, với tham số truyền vào là id của ví dụ cần tìm, id này được lấy trong req.query bên controller.

// Lấy ra Ví dụ theo id

exports.getExampleById = async (id, portal) => {

    let example = await Example(connect(DB\_CONNECTION, portal)).findById({ \_id: id });

    if (example) {

        return { example };

    }

    return -1;

}

Service editExample dùng để chỉnh sửa một ví dụ, có 3 tham số là **id** của ví dụ cần sửa lấy trong req.query bên controller, data là dữ liệu mới muốn chỉnh sửa lấy trong req.body bên controller, và tham số thứ 3 là portal.

Dựa vào id truyền vào, thực hiện truy vấn dùng **findById** lấy được thông tin của ví dụ cần chỉnh sửa. Sau đó dùng hàm **update** để cập nhật giá trị mới.

Cuối cùng, sau khi cập nhật, dùng **findById** để tìm ví dụ vừa được cập nhật và trả về cho **client**

// Chỉnh sửa một Ví dụ

exports.editExample = async (id, data, portal) => {

    let oldExample = await Example(connect(DB\_CONNECTION, portal)).findById(id);

    if (!oldExample) {

        return -1;

    }

    await Example(connect(DB\_CONNECTION, portal)).update({ \_id: id }, { $set: data });

    let example = await Example(connect(DB\_CONNECTION, portal)).findById({ \_id: oldExample.\_id });

    return example;

}

Service deleteExample, dùng để xóa một ví dụ.

Sử dụng hàm **findByIdAndDelete** để tìm kiếm thông tin của ví dụ trong collection Example dựa vào **id** lấy được trong **req.query** bên controller. Khi tìm thấy, xóa bản ghi này đi và trả về cho client bằng cách gọi **findByIdAndDelete.**

// Xóa một Ví dụ

exports.deleteExample = async (id, portal) => {

    let example = Example(connect(DB\_CONNECTION, portal)).findByIdAndDelete({ \_id: id });

    return example;

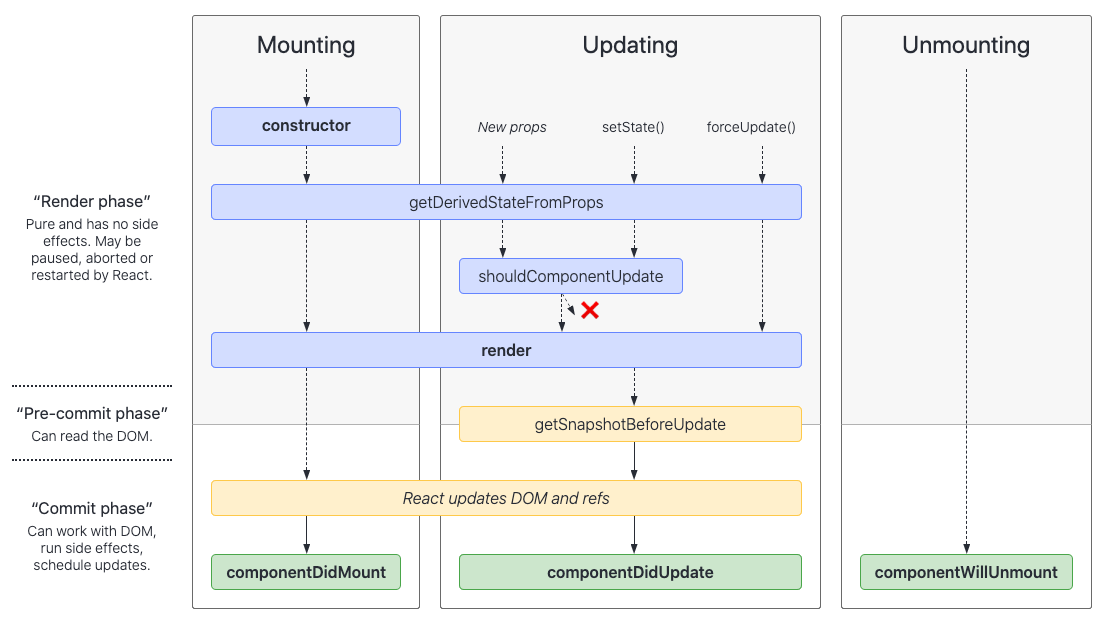
}

# Hướng dẫn về React

## Class component

### Vòng đời component

<https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/>



### 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 một trong 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`;

}, [count]);

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 />

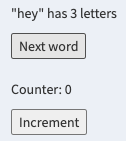
<p>Counter: {count}</p>

<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}

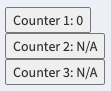
</button>

</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(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.

# Hướng dẫn sử dụng webpack build server

## Cài đặt webpack

Để bắt đầu chúng ta cài đặt webpack bằng lệch sau:

**npm install --save-dev webpack webpack-cli**

Tạo file webpack.config.js cùng cấp thư mục với file server/index.js

var webpack = require('webpack');

var path = require('path');

var fs = require('fs');

var nodeModules = {};

fs.readdirSync('node\_modules')

.filter(function (x) {

return ['.bin'].indexOf(x) === -1;

})

.forEach(function (mod) {

nodeModules[mod] = 'commonjs ' + mod;

});

module.exports = {

entry: './index.js',

target: 'node',

output: {

path: path.join(\_\_dirname, 'build'),

filename: 'backend.js'

},

// externals: nodeModules,

module: {

rules: [

{

test: /\.txt$/i,

use: 'raw-loader',

},

{

test: /\.log$/i,

use: 'raw-loader',

},

{

test: /\.js$/,

exclude: [

path.resolve(\_\_dirname, '/global.js'),

], // bỏ qua các thư mục node\_module

enforce: 'pre',

use: ['source-map-loader'],

},

],

},

plugins: [

new webpack.IgnorePlugin(/\.(css|less)$/),

],

mode: 'production',

devtool: 'inline-source-map'

}

**Entry**: Là file webpack bắt đầu làm việc, là file import các module của toàn project, ở đây là file **index.js**

**Output:** Path: Là đường dẫn đến file bundle.Filename: Tên file khi bundle

**Externals**: Trả về danh sách các thư viện nằm trong node\_module để không bundle các thư viện.

**Target**: “node” cài đặt cho môi trường nodejs

**Loader:** Là nơi chứa một mảng các compile các ngôn ngữ khác nhau

**Test**: Là chuỗi regex, khi phân tích các file, webpack chỉ tìm các file có định dạng đó. Ví dụ: /\.js$/

## Chạy webpack

**Thêm vào file packet.json:**

"scripts": {

    "build": "(webpack) && (nodemon build/backend.js)",

}

Trong đó:

* (webpack) là câu lệnh để chạy webpack, build các module js trong project
* (nodemon build/backend.js) để chạy file mà chương trình vừa bundle.

**Mở termimal, gõ:**

cd server && npm run build để build project

Ảnh có chứa văn bản

Mô tả được tạo tự động

Chạy thành công webpack.

# 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)