# Mục lục

[Mục lục 1](#_Toc66614083)

[Danh mục hình vẽ 5](#_Toc66614084)

[Danh mục bảng 6](#_Toc66614085)

[Chương 1 Giới thiệu 7](#_Toc66614086)

[1.1 Công nghệ sử dụng trong dự án quản lý công việc 7](#_Toc66614087)

[1.2 Một số thuật ngữ cần chú ý 7](#_Toc66614088)

[Chương 2 Mô hình phân quyền RBAC và cấu trúc CSDL trong dự án quản lý công việc 9](#_Toc66614089)

[2.1 Mô hình phân quyền RBAC 9](#_Toc66614090)

[2.2 Cơ sở dữ liệu của dự án Quản Lý Công Việc 9](#_Toc66614091)

[2.2.1 Mô hình 9](#_Toc66614092)

[2.2.2 Company 12](#_Toc66614093)

[2.2.3 User 12](#_Toc66614094)

[2.2.4 Role 13](#_Toc66614095)

[2.2.5 UserRole 14](#_Toc66614096)

[2.2.6 RoleType 14](#_Toc66614097)

[2.2.7 OrganizationalUnit 16](#_Toc66614098)

[2.2.8 Privilege 16](#_Toc66614099)

[2.2.9 Link 17](#_Toc66614100)

[2.2.10 Component 18](#_Toc66614101)

[2.2.11 Action 18](#_Toc66614102)

[Chương 3 Project quản lý công việc 19](#_Toc66614103)

[3.1 Coding convention 19](#_Toc66614104)

[3.1.1 Đặt tên Rest Services 19](#_Toc66614105)

[3.1.2 MongoDB 21](#_Toc66614106)

[3.1.3 NodeJS 23](#_Toc66614107)

[3.1.4 ReactJS 23](#_Toc66614108)

[3.2 Bảo mật trên website 25](#_Toc66614109)

[3.2.1 Các kỹ thuật được sử dụng trong bảo mật 25](#_Toc66614110)

[3.2.2 Xác thực tài khoản người dùng 27](#_Toc66614111)

[3.2.3 Xác thực quyền của người dùng 28](#_Toc66614112)

[Chương 4 Cài đặt và cấu hình 29](#_Toc66614113)

[4.1 Cấu hình xác thực truy cập trong MongoDB 29](#_Toc66614114)

[4.2 Cài đặt cấu hình và chạy project 31](#_Toc66614115)

[4.3 Hướng dẫn cài đặt các plugins cho developer 35](#_Toc66614116)

[4.3.1 Plugin cho visual code 35](#_Toc66614117)

[4.3.2 Plugin cho trình duyệt Chrome 36](#_Toc66614118)

[Chương 5 Cấu trúc project 38](#_Toc66614119)

[5.1 Thư mục Client 38](#_Toc66614120)

[5.1.1 Common components 39](#_Toc66614121)

[5.1.2 Helpers 40](#_Toc66614122)

[5.1.3 Lang 47](#_Toc66614123)

[5.1.4 Layout 47](#_Toc66614124)

[5.1.5 Modules 47](#_Toc66614125)

[5.1.6 React-routers 47](#_Toc66614126)

[5.1.7 Redux 50](#_Toc66614127)

[5.1.8 index.js app.js và app.css 52](#_Toc66614128)

[5.1.9 Các file khác 52](#_Toc66614129)

[5.2 Cấu trúc thư mục server 53](#_Toc66614130)

[5.2.1 Index.js 53](#_Toc66614131)

[5.2.2 Helpers 53](#_Toc66614132)

[5.2.3 Log 54](#_Toc66614133)

[5.2.4 Middleware 55](#_Toc66614134)

[5.2.5 Model 58](#_Toc66614135)

[5.2.6 Modules 58](#_Toc66614136)

[5.2.7 Seed 59](#_Toc66614137)

[5.2.8 Upload 59](#_Toc66614138)

[5.2.9 Package.json 59](#_Toc66614139)

[5.3 Cách tạo mới một module 59](#_Toc66614140)

[5.3.1 Mã nguồn Client 59](#_Toc66614141)

[5.3.2 Mã nguồn Server 65](#_Toc66614142)

[5.4 Cách tạo mới một common component 70](#_Toc66614143)

[5.5 Giải thích module example 71](#_Toc66614144)

[5.5.1 Mã nguồn Client theo mô hình 1 71](#_Toc66614145)

[5.5.2 Mã nguồn client theo mô hình 2 89](#_Toc66614146)

[5.5.3 Mã nguồn Server 94](#_Toc66614147)

[5.6 Hướng dẫn phân quyền cho trang và component 96](#_Toc66614148)

[5.6.1 Phân quyền cho trang 96](#_Toc66614149)

[5.6.2 Phân quyền cho component 96](#_Toc66614150)

[5.7 Hướng dẫn cài đặt các tính năng realtime 99](#_Toc66614151)

[5.7.1 Socket.io 99](#_Toc66614152)

[5.7.2 Firebase 106](#_Toc66614153)

[5.8 Hướng dẫn sử dụng chức năng upload file 109](#_Toc66614154)

[5.8.1 Quy tắc đường dẫn lưu trên server 110](#_Toc66614155)

[5.8.2 Mã nguồn client 110](#_Toc66614156)

[5.8.3 Mã nguồn server 113](#_Toc66614157)

[5.9 Cách dùng biến redux lưu kết quả từ nhiều lời gọi service 118](#_Toc66614158)

[5.10 Đợi đủ dữ liệu mới render component 119](#_Toc66614159)

[5.11 Cách code chuẩn tắc 119](#_Toc66614160)

[5.11.1 Viết code theo đoạn 119](#_Toc66614161)

[5.11.2 Comment code 119](#_Toc66614162)

[5.11.3 Cách dùng biến state, props phía client 121](#_Toc66614163)

[5.11.4 Một số lưu ý chung 121](#_Toc66614164)

[5.11.5 Cách populate truy vấn đúng trong mã nguồn server 121](#_Toc66614165)

[5.12 Hướng đẫn dùng song ngữ 122](#_Toc66614166)

[5.13 Cơ sở dữ liệu 124](#_Toc66614167)

[5.13.1 Hướng dẫn tạo mới collection 124](#_Toc66614168)

[5.13.2 Xóa dữ liệu 126](#_Toc66614169)

[5.13.3 Các câu lệnh truy vấn cơ bản 126](#_Toc66614170)

[5.13.4 Hướng dẫn cài đặt quan hệ nhiều nhiều 127](#_Toc66614171)

[Chương 6 Hướng dẫn về React 128](#_Toc66614172)

[6.1 Class component 128](#_Toc66614173)

[6.1.1 Vòng đời component 128](#_Toc66614174)

[6.1.2 Hướng dẫn phân biệt và sử dụng getDerivedStateFromProps và shouldComponentUpdate 128](#_Toc66614175)

[6.1.3 Trình tự vòng đời ReactJS 129](#_Toc66614176)

[6.2 Function component 130](#_Toc66614177)

[6.2.1 React.memo 130](#_Toc66614178)

[6.2.2 React Hooks 132](#_Toc66614179)

[Chương 7 Hướng dẫn sử dụng webpack build server 144](#_Toc66614180)

[7.1 Cài đặt webpack 144](#_Toc66614181)

[7.2 Chạy webpack 145](#_Toc66614182)

[Tài liệu tham khảo 147](#_Toc66614183)

# Danh mục hình vẽ

[**Hình 1** Ví dụ về JSON 7](#_Toc66614184)

[**Hình 2** Mô hình CSDL 10](#_Toc66614185)

[**Hình 3** Ví dụ về coding convention trong MongDB 22](#_Toc66614186)

[**Hình 4** JSX đa dòng 24](#_Toc66614187)

# Danh mục bảng

[**Bảng 1** Company Collection 12](#_Toc66614188)

[**Bảng 2** User Collection 12](#_Toc66614189)

# 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
* OrganizationalUnit
* 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, organizational unit, resource (link, component, …) sẽ có một thuộc tính là company – để xác định user, role, organizational unit, 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

**Bảng 1** Company Collection

|  |  |  |
| --- | --- | --- |
| 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

**Bảng 2** User Collection

|  |  |  |
| --- | --- | --- |
| 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.
* **Manager**: 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 **Deputy Manager**, **Employee**. Việc phân quyền này có thể linh động cho từng công ty, được quản lý bởi **Admin**.
* **Deputy Manager**: 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 **OrganizationalUnit** (*đơn vị, phòng ban, …*) nào đó. Với một **RoleChucDanh,** người dùng sẽ có quyền tương ứng với một tài nguyên mà trong phạm vi **OrganizationalUnit** 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: **Manager**, phó phòng kinh doanh, nhân viên phòng kinh doanh. Tương tự, phó phòng kinh doanh kế thừa **Deputy Manager**, 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.

### OrganizationalUnit

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 |
| **managers** | *ObjectId* | Trưởng đơn vị |
| **deputyManagers** | *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 /organizational-units/units-that-user-is-managers/:id 🡪 GET /super-admin/organizational-units?managerOfUnit=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: organizationalUnitName )

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

( VD: DeputyManager,…)

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

### ReactJS

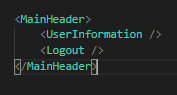
#### Quy tắc chung

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.

# Cài đặt và cấu hình

## Cấu hình xác thực truy cập trong MongoDB

Phần này là tùy chọn. Nếu muốn cài đặt bảo mật cho MongoDB, làm theo hướng dẫn trong phần này. Nếu không cần bảo mật có thể bỏ qua.

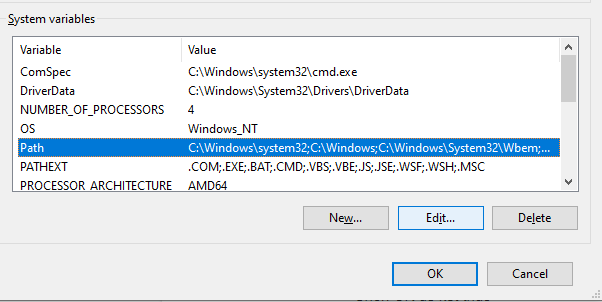
Các bước thực hiện: (**Chú ý:  text màu đỏ: thay đổi giá trị tùy ý** )

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

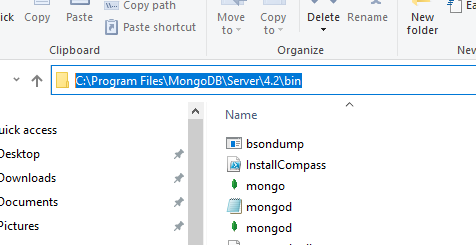
Chuột phải vào biểu tượng This PC->Properties->Advanced system settings

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*" }

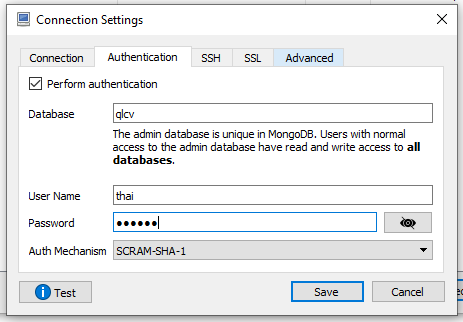
]

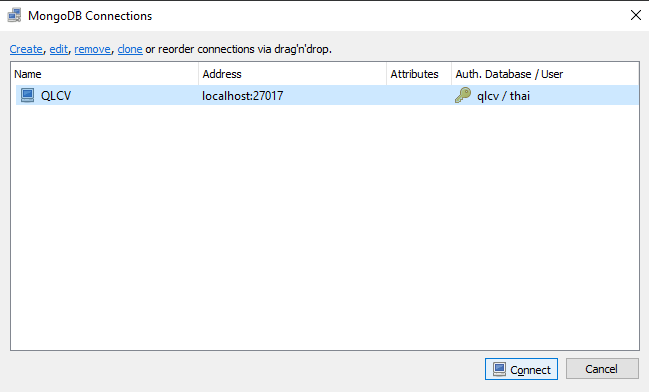
 } );

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>

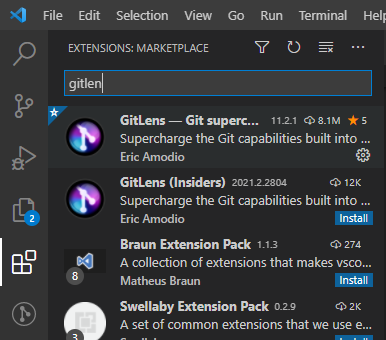
* Lưu cấu hình file **.env** lại và ở trong thư mục **client**, tiến hành chạy lệnh npm start hoặc yarn start trên Terminal

## Hướng dẫn cài đặt các plugins cho developer

### Plugin cho visual code

#### GitLens

Gitlens là một công cụ hỗ trợ Git trên VS Code, giúp chúng ta biết gần nhất ai edit một dòng code, comment cho commit trên file code đó như thế nào, và lịch sử file code thay đổi như thế nào. Nếu không hiểu code đang viết gì thì chúng ta tìm ra tác giả dòng code đó để hỏi.



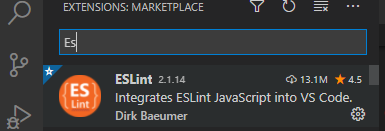
Tìm kiếm extensions GitLens, nằm đầu tiên và click vào install. Restart lại VS Code.

#### ESLint

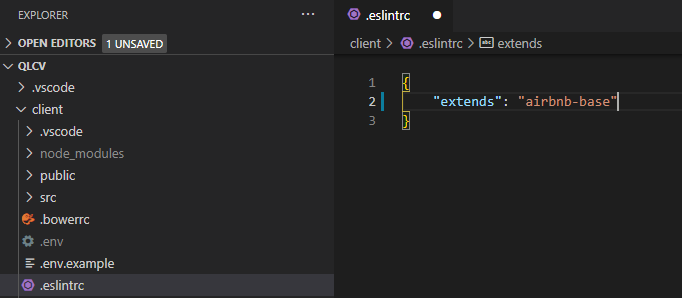
ESLint là tiện ích giúp ta đảm bảo code có format chuẩn tắc. Để sử dụng linter, ta phải mất công setup rất nhiều rule (dùng *let* hoặc *const* thay cho *var*, thụt vào 2 dòng đầu dòng, …). Thay vì lựa chọn và setup từng rule một, phần đông developer sử dụng những bộ rule đã có sẵn như rule của airbnb: <https://github.com/airbnb/javascript>

Cd vào thư mục client và server chạy: npm install -D eslint eslint-plugin-import eslint-config-airbnb-base

Mở VS Code lên, mở project, tìm và install extension ESLint



Tạo một file mang tên **.eslintrc** trong cùng thư mục. Đây là file chứa config của ESLint. Trong file này, ta cấu hình ESLint dùng style của AirBNB.



### Plugin cho trình duyệt Chrome

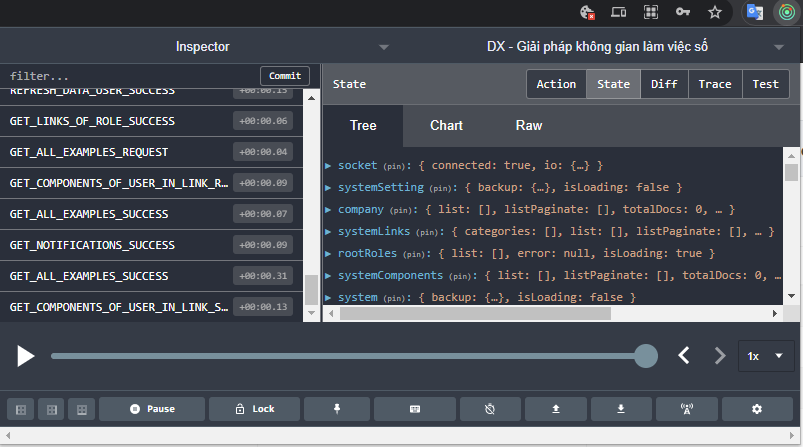
#### Redux dev tools

Redux dev tools là 1 extension cài được trên các browser chrome hay firefox giúp ta xem kho trạng thái redux của app một cách trực quan nhất mà không cần đến view console. ngoài ra ta còn xem list các commit, các actions step by step. Giúp chúng ta biết được đường đi nước bước, logic từng màn hình một cách dễ dàng nhất.

Link cài:

<https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd>

Sau khi cài xong click vào biểu tượng extension trên Chrome.

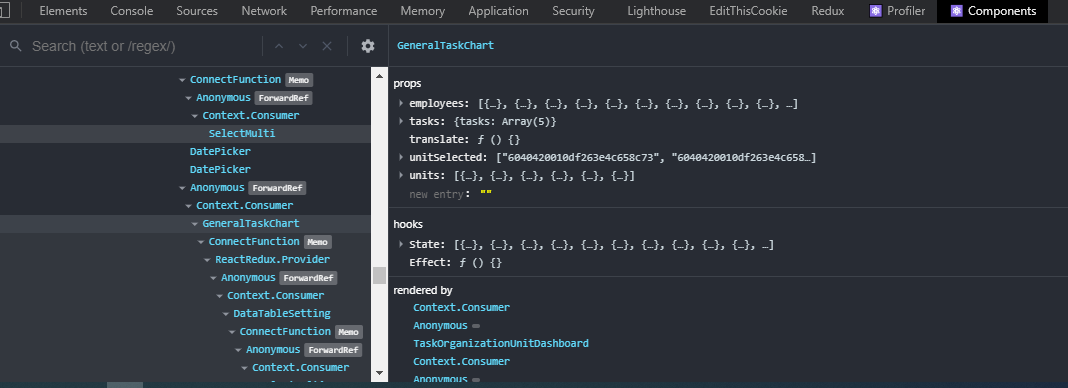


#### React Developer Tool

React developer tools là một tiện ích mở rộng cho trình duyệt đến từ Facebook. Tiện ích cung cấp công cụ cho phép chúng ta kiểm tra xem dữ liệu và cấu trúc JSX có ổn không, hỗ trợ developer debug ứng dụng React.

Link cài cho Chrome

<https://chrome.google.com/webstore/detail/react-developer-tools/fmkadmapgofadopljbjfkapdkoienihi?hl=en>



Để sử dụng, bật google dev tool lên (f12), sử dụng mục **components** để xem các components trên trang, các props hiện có trên components

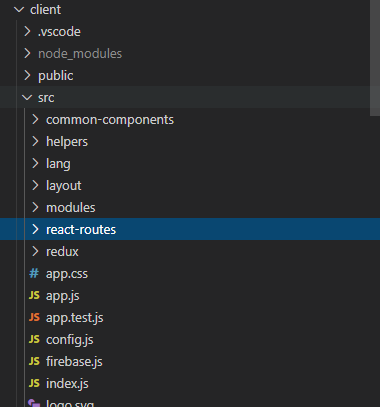
Tab profiler dùng để xem quá trình chạy, render ra component.

# 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 khung 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 chỉ cần sửa đổi ít nhất có thể ở các common components này

Mỗi component ứng với một thư mục con trong thư mục common-components/src. File **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 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

##### Tham số này 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 gọi api

**params**: là các tham số URL được gửi khi call api (VD tham số x, y trong http://a.b.c?x=5&y=6)

**responseType**: chỉ định kiểu dữ liệu mà server sẽ trả về, có thể là 'arraybuffer', 'blob', 'document', 'json', 'text', 'stream'. Mặc định là kiểu **json**

**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 trả về đối tượng với 4 trường thông tin, trong đó sẽ mã hóa **3** trường thông tin 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), do trường này đã được mã hóa sẵn

Ví dụ: Gọi api sử dụng hàm tiện ích **sendRequest .**

function get(params) {

    return sendRequest(

        {

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

            method: "GET",

            params,

        },

        false,

        true,

        "super\_admin.user"

    );

}

##### Tham số showSuccessAlert

Tham số này nhận vào giá trị true hoặc false, mặc định bằng false. Đặt giá trị là true nếu muốn hiển thị alert thông báo khi call api thành công.

##### Tham số showFailAlert

Tham số này nhận vào giá trị true hoặc false, mặc định bằng true. Đặt giá trị true nếu muốn hiển thị alert thông báo khi call api thất bại hay không.

##### Tham số successTitle

Tham số này nhận giá trị là key translate trong file vn/en.js để hiển thị tiêu đề cho thông báo khi call api thành công. Giá trị mặc định là “**general.success**”, ứng với tiêu đề mặc định

##### Tham số successTitle

Tham số này nhận giá trị là key translate trong file vn/en.js để hiển thị tiêu đề cho thông báo khi call api thất bại. Giá trị mặc định là “**general. error**”, ứng với tiêu đề mặc định

Ví dụ gọi api sử dụng hàm tiện ích **sendRequest:**

function get(params) {

    return sendRequest(

        {

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

            method: "GET",

            params,

        },

        false,

        true,

        "super\_admin.user"

    );

}

##### Gửi dữ liệu đi kèm tới server

Khi call api tới server, có 3 cách để truyền dữ liệu:

**Cách 1 - Qua URL param**. Cách này truyền dữ liệu qua tham số trên URL sau dấu ?, ví dụ: truyền limit và page thông qua url param,

url sẽ có dạng: https://localhost:8000/crm/customers?limit=5&page=1

function getCustomers(params) {

    return sendRequest({

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

        method: 'GET',

        params: {

            limit: 5,

            page: 1

        },

    }, false, true, 'crm.customer');

}

**Cách 2 - Qua Request param**. Tham số cũng được đưa vào URL. Ví dụ: id của khách hàng được truyền qua request param như sau

function getCustomer(id) {

    return sendRequest({

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

        method: 'GET',

    }, false, true, 'crm.customer');

}

**Cách 3 - Qua Request body**. Cách truyền này thường dùng khi dùng phương thức POST, PUT, PATCH, ví dụ: khi thêm mới một bảng lương, thì dữ liệu của bảng lương sẽ gửi trong body thông qua giá trị của **data** trong tham số **options** của hàm **sendRequest**

function createSalary(data) {

    return sendRequest({

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

        method: 'POST',

        data: data,

    }, true, true, 'human\_resource.salary');

}

#### Validate form

##### Lớp ValidationHelper

File hỗ trợ việc validate các form trong các module đó là file **validationHelper.js ,** file này cài đặt lớp ValidationHelper có các phương thức cơ bản để kiểm tra, bắt lỗi nhập liệu trên các form. Các phương thức chia làm 2 nhóm. Phương thức cấp 1 là nhóm phương thức cơ sở nhất. Các phương thức cấp 2 sẽ gọi lại các phương thức cơ sở này.

Phương thức cấp 1: validateEmpty, validateInvalidCharacter, validateLength, validateMinimumLength, validateMaximumLength, validateNumberInput

Phương thức cấp 2: validateName, validateDescription, validatePassword

Trong mỗi module, nếu cần thêm các kiểu validate data đặc biệt khác, cần viết lớp kế thừa lớp ValidationHelper. Lớp đó sẽ kế thừa được các phương thức cấp 1 và cấp 2, tăng tính tái sử dụng mã nguồn. Nếu hàm validate dữ liệu đó có tính tái sử dụng cao, bổ sung thêm hàm validate đó vào lớp gốc ValidationHelper.

Chú ý: có thể mã hóa thông điệp được translate trong file vn/en.js. Ví dụ, trong file vn.js ta định nghĩa key number\_inpu\_error như sau

number\_input\_error: "Giá trị phải từ {min} đến {max}"

Sau đó giá trị translate được lấy ra như sau

translate('general.validate.number\_input\_error', {min, max})

Ví dụ, phương thức cấp 1 **validateEmpty**: dùng để kiểm tra xem giá trị nhập vào có rỗng không. Phương thức gồm 2 tham số, tham số thứ nhất translate là biến được truyền vào để sử dụng song ngữ cho message, tham số thứ 2 là **value** - giá trị cần validate.

static validateEmpty = (translate, value) => {

    if(!value)

        return { status: false, message: translate('general.validate.empty\_error') };

    return { status: true };

}

Tương tự là phương thức **validateInvalidCharacter** để kiểm tra giá trị nhập có chứa kí tự đặc biệt hay không, phương thức **validateLength** kiểm tra độ dài của giá trị nhập có hợp lệ, phương thức **validateEmail** kiểm tra email có hợp lệ hay không.

Các phương thức này nếu giá trị nhập hợp lệ thì sẽ trả về đối tượng với một thuộc tính là trạng thái **status = true,** nếu không hợp lệ sẽ trả ra đối tượng với thuộc tính trạng thái **status = false** và thuộc tính **message** chứa thông điệp báo lỗi cho người dùng.

##### Ví dụ áp dụng

Form thêm mới cần kiểm tra xem mã phiếu có trống hay không, nếu trống thì không cho lưu. Trước tiên, ta import phương thức vào module, tùy cấp độ thư mục chỉnh lại đường dẫn cho đúng.

import ValidationHelper from '../../../../helpers/validationHelper';

Ở hàm xử lý sự kiện **onChange** khi thay đổi mã phiếu, gọi hàm handleDateCreateChange. Trong đó: **value** là giá trị người dùng nhập, **translate** hỗ trợ song ngữ. Để validate, gọi tới phương thức **validateEmpy** từ lớp ValidationHelper để kiểm tra xem giá trị nhập có trống hay không, giá trị biến message sẽ có giá trị nếu value rỗng, ngược lại sẽ bằng undefined, tiến hành set state lại message lỗi để hiển thị ra giao diện người dùng

handleDateCreateChange = (value) => {

    let { translate } = this.props;

    let { message } = ValidationHelper.validateEmpty(translate, value);

    this.setState({

        dateCreate: value,

        errorOnDateCreate: message

    });

}

#### Cấu hình bảng danh sách dữ liệu

Với các bảng danh sách dữ liệu, sử dụng file tiện ích tableConfiguration.js. File này chứa các hàm tiện ích dùng để: (i) lấy thông tin cấu hình( ẩn, hiện cột, giới hạn số bản ghi) của table trong localStorge ra màn hình danh sách, và (ii) ghi thông tin cấu hình bảng mà người dùng thiết lập vào localStorage.

#### Sinh mã ID

Để sinh mã Id, sử dụng file generateCode.js. File này chứa hàm tự động sinh mã code theo thời gian, do đó đảm bảo được mỗi lần generate đều duy nhất, phục vụ có các form cần sinh tự động mã.

export const generateCode = (code) => {

    const date = new Date();

    const year = String(date.getFullYear());

    const month = padLeft(date.getMonth() + 1, 10, '0');

    const day = padLeft(date.getDate(), 10, '0');

    const hour = padLeft(date.getHours(), 10, '0');

    const minute = padLeft(date.getMinutes(), 10, '0');

    const second = padLeft(date.getSeconds(), 10, '0');

    const milisecond = padLeft(date.getMilliseconds(), 10, '0');

    const stringCode = code + year + month + day + "." + hash("" + hour + minute + second + milisecond);

    return stringCode;

}

### 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.

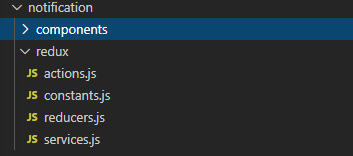
Xem mục 5.12 để biết cách làm việc chuẩn tắc với song ngữ.

### 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"}

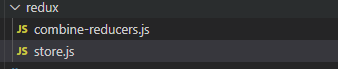
    layout={Layout}

    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(

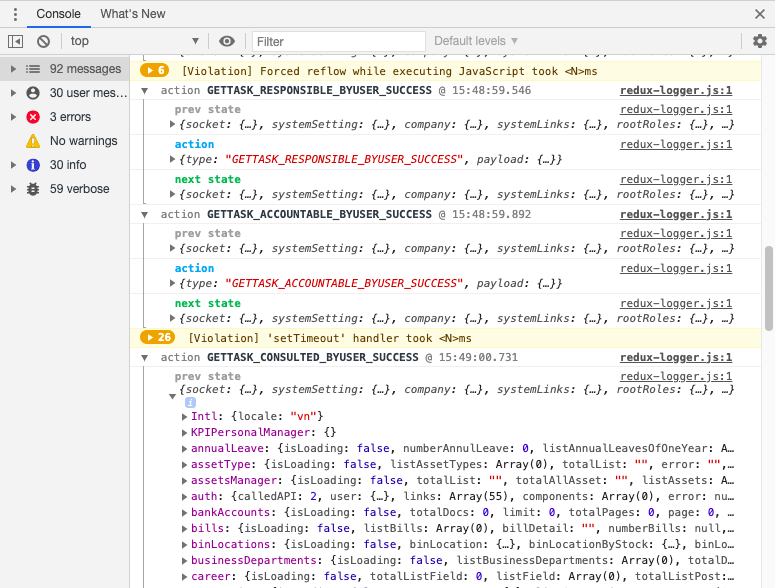
rootReducer,

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.

**emailHelper.js:** file chứa hàm tiện ích phục vụ cho việc gửi mail

**functionHelper:** file chứa các hàm tiện ích như hàm decryptMessage dùng để giải mã các req trong header khi client gửi qua, các hàm như freshObject, freshArray dùng để làm sạch các object, mảng, loại bỏ các giá trị như undefined, null,…

Và còn nhiều hàm tiện ích khác, phục vụ cho việc xử lý khác nhau

### 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
* **initSampleCompanyDB.js**: file này sau khi chạy sẽ khởi tạo công ty vnist làm ví dụ, và khởi tạo các dữ liệu liên quan đến vnist như: tài khoản người dùng, tạo các role, và các dữ liệu mẫu khác.
* **initOrganizationalUnitKPIData.js:** file này có nhiệm vụ tạo dữ liệu mẫu cho module kpi (kpi đơn vị và kpi cá nhân)
* **initHumanResourceData.js:** file này có nhiệm vụ tạo dữ liệu mẫu cho module quản lý nhân sự.

### 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)

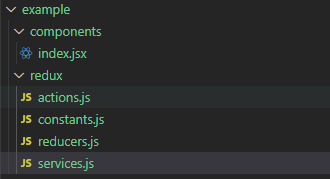
## Cách tạo mới một module

### 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",

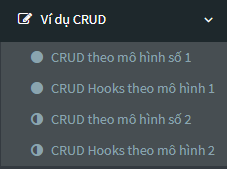
example: "ví dụ tạo mới 1 trang",

….,}

**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"}

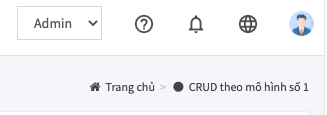
    layout={Layout}

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

/>

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



* **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 example ở trên, ta truyền vào component **Example**

#### 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"

    },

}

Với mỗi một trang trong module, cần thêm cấu hình url, description, category của trang. Sau đó phân quyền những role nào được truy cập vào và hoạt động trên trang đó, bằng cách sử dụng thuộc tính **roles** là một mảng, chứa tên các **role** được cho phép hoạt động trên trang

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')

}]

Nếu muốn một số role khi truy cập trang này có thêm các components cho riêng các role đó (các role khác bị ẩn đi), thêm khai báo thuộc tính components. Chi tiết xem mục 5.6.2

#### 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 no

        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

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.

##### 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

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).

**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

Các file service.js để xử lý yêu cầu dịch vụ trong module. Cần định nghĩa rõ ràng các tham số đầu vào cho mỗi một chức năng trong file 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"));

## Cách tạo mới một common component

Truy cập vào thư mục : **qlcv/client/src/common-component/src,** tiến hành tạo một thư mục, đặt tên có ý nghĩa với component định thực hiện.

Ví dụ: tạo component có tên là **error-label**, thì tiến hành tạo thư mục **quill-editor** bên trong đường dẫn ở trên, và tạo các file .jsx, .css(nếu có), khi viết nên có 1 file README.txt để viết hướng dẫn sử dụng.



Sau khi viết xong thì export module đã viết

import React, { Component } from 'react';

class ErrorLabel extends Component {

    constructor(props) {

        super(props);

        this.state = {}

    }

    render() {

        const { content } = this.props;

        return (

            <React.Fragment>

                {

                    content && <div className="help-block">{content}</div>

                }

            </React.Fragment>

         );

    }

}

export { ErrorLabel };

Sau đó mở file index.js: (**qlcv/client/src/common-component/src/index.js)**

Thêm đoạn dưới vào file để combine component.

export \* from './src/error-label/errorLabel';

=> Tạo mới thành công common-component

## 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

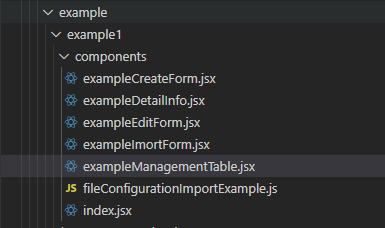
#### Tạo các components giao diện

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)

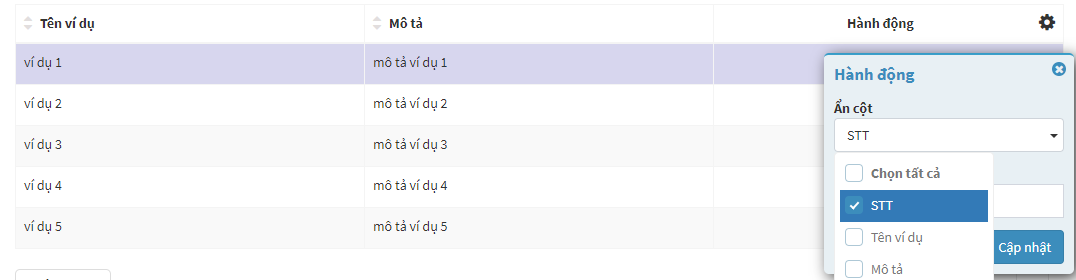
            }

        });

        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}

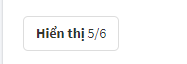
/>

**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 });

        }

    }

### Mã nguồn client theo mô hình 2

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** có nhiệm vụ set id ví dụ vào state sau đó hiển thị modal xem chi tiết lên

handleShowDetailInfo = (id) => {

        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 phân quyền cho trang và component

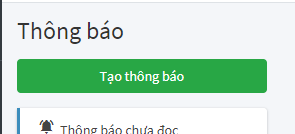
### Phân quyền cho trang

Phân quyền cho trang giải thích trong mục 5.3.2.1

### Phân quyền cho component

Một component là một thành phần UI nào đó trên một trang, ví dụ như nút tạo thông báo, nút thêm mới mẫu công việc. Phân quyền component là chỉ cho một số các role được phép thấy và thao tác trên component đó khi truy cập vào trang tương ứng (các role này đương nhiên đã được phân quyền vào trang đó)

Ví dụ chỉ có người dùng có quyền Admin/super admin mới có component nút tạo thông báo khi truy cập vào trang thông báo. Các role khác không có quyền nên nút tạo mới đó sẽ không được hiển thị.



Để thêm mới một component cho một trang nào đó, như nút tạo thông báo nói trên, ta làm như sau:

**Mã nguồn Server**

Mở file **config.js** nằm trong thư mục **qlcv/server/helpers**

const COMPONENTS = [{

    name: 'create-notification',

    description: 'Tạo thông báo mới',

    roles: [

        ROOT\_ROLES.ADMIN.name,

        ROOT\_ROLES.SUPER\_ADMIN.name,

    ],

    links: [

        '/notifications'

    ]

},

]

Thêm đoạn code trên vào mảng COMPONENTS, trong đó name và description là tên và mô tả của component, roles là những người dùng có role nào thì được sử dụng componnents này, và links tương ứng với trang sử dụng component, ở đây dùng trong trang /notifications.

Đồng thời, trogn trang notification cũng phải khai báo sử dụng components

{

url: '/notifications',

description: 'Thông báo',

category: LINK\_CATEGORY.COMMON.name,

roles: [

ROOT\_ROLES.SUPER\_ADMIN.name,

ROOT\_ROLES.ADMIN.name,

ROOT\_ROLES.MANAGER.name,

ROOT\_ROLES.DEPUTY\_MANAGER.name,

ROOT\_ROLES.EMPLOYEE.name,

],

components: getComponentsInLink('/notifications')

},

Sau khi thêm xong thì cd vào thư mục server và khởi tạo lại db chạy lệnh **npm run init**

**Mã nguồn Client**

Mở file **header.jsx** (client/src/layout/header/components)

Để client biết được người dùng với role đang chọn có quyền thao tác với những components nào, ta sử dụng service getLinksOfRole và getComponentsOfUserInLink

componentDidMount() {

        this.props.refresh();

        const currentRole = getStorage("currentRole");

        this.props.getLinksOfRole(currentRole)

            .then(res => {

                const links = res.data.content;

                const path = window.location.pathname;

                const linkId = this.getLinkId(path, links);

                this.props.getComponentsOfUserInLink(currentRole, linkId);

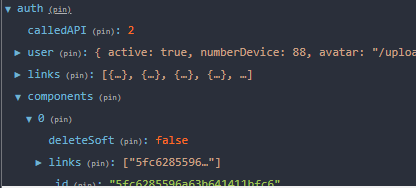
            })

    }

Trước tiên, service getLinkOfRole sẽ lấy tất cả những link (page) nào mà người dùng hiện tại được phép truy cập. Nếu lấy thông tin thành công, dựa vào window.location.pathname để lấy đường link hiện tại (ví dụ /notification), từ đó so khớp với danh sách các links người dùng truy cập, và lấy được id của page hiện tại.

Khi có được id của page hiện tại, gọi servive **getComponentOfUserInLink** để lấy tất cả các **component** mà người dùng với role hiện tại được phép truy cập trong page đang xét.

Ví dụ đăng nhập vào tài khoản Admin, xem bằng công cụ redux dev tool thì sẽ thấy được component mà Admin được dùng trên trang có link : **/notifications**



Nhờ đoạn code xử lý trên trong file header, khi phát triển 1 page mới, ta có thể check xem người dùng có được dùng component nào đó hay không bằng cách gọi hàm checkHasComponent với tham số truyền vào là id của component như sau.

return (

            <React.Fragment>

                <div className="" role="tabpanel">

                    <div className="tab-content">

                        <TabNotificationReceivered/>

                        {

                            this.checkHasComponent('create-notification') && <TabNotificationSent/>

                        }

                        <TabNotificationUnRead/>

                    </div>

                </div>

            </React.Fragment>

        );

## Hướng dẫn cài đặt các tính năng realtime

Có thể dùng socket.io hoặc firebase notification để cài đặt các tính năng thời gian thực cho ứng dụng.

### Socket.io

Socket.io là 1 thư viện giúp phát triển ứng dụng realtime. Ví dụ: khi nguyễn văn A (NVA) tạo mới một công việc có mặt của Nguyễn văn B (NVB), NVB sẽ lập tức nhận được thông báo trong thời gian thực, không cần phải reload lại trang.

Nguyên lý áp dụng socket.io vào project như sau. Khi một người truy cập vào website, sau khi họ đăng nhập (hoặc đã đăng nhập và lưu lại token từ trước), app client sẽ gửi kết nối tới server nodejs. Server sau đó bắt kết nối với client, lấy được socket id và user id của người dùng. Một người dùng có thể mở nhiều tab hoặc mở ứng dụng trên nhiều trình duyệt. Khi đó họ sẽ có cùng một userId nhưng socketId là khác nhau.

Một client có thể kích hoạt một sự kiện tới server (VD: khi NVA tạo mới công việc). Sau khi nhận sự kiện này, server sẽ emit (gửi dữ liệu) sang client với userId nào đó (VD: NVB). Client này sẽ nhận và xử lý dữ liệu server gửi đến.

Project hiện tại áp dụng socket.io vào phần gửi/nhận thông báo trên ứng dụng web.

#### Mã nguồn server

Cần cài đặt gói socket.io : npm i socket.io (đã khai báo sẵn trong package.json)

Mở file global.js (server/global.js). Trong file này có tạo kết nối giữa client và server, lưu lại thông tin của client kết nối tới server vào mảng CONNECTED\_CLIENT

global.CONNECTED\_CLIENTS = [];

    global.SOCKET\_IO = require("socket.io")(server);

//server lắng nghe dữ liệu từ client

    SOCKET\_IO.on("connection", function (socket) {

        CONNECTED\_CLIENTS.push({

            socketId: socket.id,

            userId: socket.handshake.query.userId,

        });

//server đóng kết nối với client

        socket.on("disconnect", function () {

            CONNECTED\_CLIENTS = CONNECTED\_CLIENTS.filter(

                (client) => client.socketId !== socket.id

            );

        });

        console.log("User connected: ", CONNECTED\_CLIENTS);

    });

Để hiểu cách server gửi dữ liệu cho client thông qua socket.io, xét ví dụ server gửi thông báo notification cho client.

Mã nguồn được cài đặt trong file notification.service.js (server/modules/notification) với hàm **createNotification** thực hiện tạo thông báo lưu vào db và gửi dữ liệu tới client bằng socket.io. Trong đó, notify là dữ liệu server muốn gửi qua cho client, CONNECTED\_CLIENT là mảng danh sách các socket-client kết nối tới server.

// Gửi thông báo cho các user - bằng socket trên web

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

    let noti = await Notification(

        connect(DB\_CONNECTION, portal)

    ).create({

        type: data.type,

….

    });

    const notify = { ...data, \_id: noti.\_id };

    const arr = CONNECTED\_CLIENTS.filter(

        (client) => client.userId === usersArr[i]

    );

    if (arr.length === 1)

        SOCKET\_IO.to(arr[0].socketId).emit("new notifications", notify);

}

Ở trên, ta lọc ra trong mảng CONNECTED\_CLIENT những socket client có userId trùng với userId trong mảng **userArr.** Mảng này là mảng những user sẽ nhận thông báo.Sau đó sử dụng phương thức tiện ích **emit** để bắn thông báo về client.

#### Mã nguồn client

Sử dụng gói **socket.io-client : npm i socket.io-client** (đã khai báo sẵn trong package.json)

Mã nguồn kết nối socket tới server được đặt trong file app.js (client/src/app.js). Mỗi lần mở website (khi đã đăng nhập trước đó), ứng dụng client sẽ gửi kết nối socket tới server. Do vậy, ta kiểm tra, nếu người dùng đã có userId (đã đăng nhập), ta dispatch một action có type là SocketConstants.CONNECT\_SOCKET\_IO

const userId = getStorage('userId');

        if(userId){

            const { socket } = store.getState();

            if(!socket.connected) store.dispatch({ type: SocketConstants.CONNECT\_SOCKET\_IO });

        }

Mở file reducer.js (client/src/modules/socket/redux), với type như trên, điều khiển sẽ chạy vào đoạn code kết nối tới **server**, gửi kèm data là **userId** như sau

const initSocket = {

    connected: false,

    io: undefined

}

export function socket(state=initSocket, action) {

    switch (action.type) {

        case SocketConstants.CONNECT\_SOCKET\_IO:

            const userId = getStorage('userId');

            if(userId){

                state.io = SocketIO(process.env.REACT\_APP\_SERVER, {

                    query: { userId }

                });

                state.connected = true;

                return {...state};

            }

            else return initSocket;

}

}

Trong kho redux của ứng dụng sẽ có thêm trạng thái (biến) socket. Trong socket có 2 thuộc tính là io và connected. Để gọi các hàm socket, ta sẽ gọi từ socket.io

Để hiểu cách client nhận dữ liệu từ server thông qua socket.io, tiếp tục xét ví dụ server gửi thông báo notification cho client. Mã nguồn được cài đặt trong file **notification.jsx** thuộc thư mục client/src/layout/header

componentDidMount() {

        this.props.getAllManualNotifications();

        this.props.getAllNotifications();

        this.props.socket.io.on('new notifications', data => {

            const { sound } = this.state;

            if (sound) {

                const audio = new Audio(audioFile);

                audio.volume = 0.7;

                audio.play();

            }

            this.props.receiveNofitication(data);

        });

    }

Trong hàm componentDidMount, muốn đăng ký sự kiện với key bất kỳ (Ví dụ **key = ‘new notification**’), ta lấy ra đối tượng socket.io lưu trong redux.

Trong đoạn code trên, khi nhận sự kiện new notification, ta bật âm thanh thông báo, sau đó gọi tới hàm **receiveNofitication** (và truyền vào data nhận từ serrver)để dispatch action có type là RECEIVE\_NOTIFICATION\_SUCCESS. Tiếp đó ta cập nhật thêm thông báo mới vào state của redux. Thông báo sẽ có tính thời gian thực, ta không cần phải refresh lại trang.

case NotificationConstants.RECEIVE\_NOTIFICATION\_SUCCESS:

            console.log("New notification: ", action.payload)

            return {

                ...state,

                receivered: {

                    ...state.receivered,

                    list: [action.payload, ...state.receivered.list],

                    paginate: [action.payload, ...state.receivered.paginate],

                },

                isLoading: false

            }

#### Ví dụ áp dụng trong module công việc

Xét ví dụ áp dụng socket.io trong module công việc, khi thêm mới một hoạt động. Giả sử công việc abc có người thực hiện là NVA và người phê duyệt là NVB, khi NVA thêm mới một hoạt động, NVB cần nhận được thông báo ngay. Tính realtime cần đảm bảo trong 2 trường hợp sau

**Trường hợp 1**: Khi NVA thêm mới một hoạt động cho công việc, NVB không mở màn hình chi tiết công việc abc (mà đang xem chi tiết một công việc khác, hoặc đang mở một trang web khác của hệ thống)

Khi đó, CONNECTED\_CLIENTS bên server sẽ là socketId và user Id của NVA và NVB. Khi NVA thêm mới hoạt động, app client sẽ gọi tới api thêm mới hoạt động phía server. Server sau khi xử lý lưu hoạt động vào DB sẽ gọi tới hàm tạo thông báo như sau.

const associatedData = {

dataType: "createTaskAction",

value: [taskAction[taskAction.length - 1]]

}

const accountableFilter = tasks.accountableEmployees.filter(obj => obj.toString() !== req.user.\_id.toString());

const associatedDataforAccountable = {

"organizationalUnits": tasks.organizationalUnit,

"title": "Phê duyệt hoạt động",

"level": "general",

"content": `<p><strong>${userCreator.name}</strong> đã thêm mới hoạt động cho công việc <strong>${tasks.name}</strong>, bạn có thể vào để phê duyệt hoạt động này <a href="${process.env.WEBSITE}/task?taskId=${tasks.\_id}" target="\_blank">${process.env.WEBSITE}/task?taskId=${tasks.\_id}</a></p>`,

"sender": userCreator.name,

"users": accountableFilter,

"associatedData": associatedData,

associatedDataObject: {

dataType: 1,

description: `<p><strong>${tasks.name}:</strong> ${userCreator.name} đã thêm mới hoạt động, phê duyệt ngay! </p>`

}

};

NotificationServices.createNotification(req.portal, tasks.organizationalUnit, associatedDataforAccountable);

Trong đó, đối tượng hằng **associatedDataforAccountable** là nội dung thông báo lưu vào db. Hàm createNotification sẽ thực hiện việc lưu đó, đồng thời dùng socket.io để gửi đối tượng này tới client (Mở file server/modules/notification/notification.service.js để xem mã nguồn)

Khi client nhận được **‘new notification’** sẽ cập nhật lại **state notification** trong **redux**, và client sẽ cập nhật số thông báo mới, hiển thị ở quả chuông trên thanh header (có kèm âm thanh thông báo)

**Trường hợp 2**: Cả NVA và NVB đang mở màn hình chi tiết công việc abc.

Ở trường hợp này, vì cả 2 đều mở màn hình chi tiết công việc abc, khi NVA thêm hoạt động, NVB ngoài nhận được thông báo như trường hợp 1, còn được tự động cập nhật lại danh sách hoạt động mà không cần refresh lại trang.

Để làm được như vậy, đối tượng **associatedDataForAccountable** có thêm thuộc tính  **associatedData** với dataType là createTaskAction và value là hoạt động vừa được thêm mới. Trường **associatedData** sẽ không được lưu trong db, chỉ gửi sang client

const associatedData = {

dataType: "createTaskAction",

value: [taskAction[taskAction.length - 1]]

}

const accountableFilter = tasks.accountableEmployees.filter(obj => obj.toString() !== req.user.\_id.toString());

const associatedDataforAccountable = {

"organizationalUnits": tasks.organizationalUnit,

"title": "Phê duyệt hoạt động",

"level": "general",

"content": `<p><strong>${userCreator.name}</strong> đã thêm mới hoạt động cho công việc <strong>${tasks.name}</strong>, bạn có thể vào để phê duyệt hoạt động này <a href="${process.env.WEBSITE}/task?taskId=${tasks.\_id}" target="\_blank">${process.env.WEBSITE}/task?taskId=${tasks.\_id}</a></p>`,

"sender": userCreator.name,

"users": accountableFilter,

"associatedData": associatedData,

associatedDataObject: {

dataType: 1,

description: `<p><strong>${tasks.name}:</strong> ${userCreator.name} đã thêm mới hoạt động, phê duyệt ngay! </p>`

}

};

NotificationServices.createNotification(req.portal, tasks.organizationalUnit, associatedDataforAccountable);

Trong hàm createNotification, sẽ sử dụng socket.io để emit dữ liệu sang client, gửi kèm theo trường **associatedData**

Mở client/src/modules/notification/redux/reducer.js. để xem cài đặt phía client. Khi nhận được data, nếu thấy có trường **associatedData,** sẽ lưu thêm trường này vào redux.

 case NotificationConstants.RECEIVE\_NOTIFICATION\_SUCCESS:

            console.log("New notification: ", action.payload)

            return {

                ...state,

                receivered: {

                    ...state.receivered,

                    list: [action.payload, ...state.receivered.list],

                    paginate: [action.payload, ...state.receivered.paginate],

                },

                associatedData: (action.payload && action.payload.associatedData) ? action.payload.associatedData : state.associatedData,

                isLoading: false

            }

Mở fie **action.jsx** nằm trong thư mục client/src/modules/task/task-perform. Ở cuối file ta lấy thêm state notifications trong redux ra

function mapState(state) {

    const { tasks, performtasks, user, auth, notifications } = state;

    return { tasks, performtasks, user, auth, notifications };

}

Trong hàm getDerivedStateFromProps, dùng if check xem nếu trong state notification có trường associatedData hay không, nếu có, tiếp tục check xem dataType có bằng **createTaskAction**  hay không.

Tiếp tục lấy danh sách các hoạt động mà NVB hiện có, push thêm hoạt động mới nhận được vào biến **res.** Cuối cùng gọi tới action **refreshDataAfterCreateAction** để cập nhật lại danh sách hoạt động cho NVB. Khi state redux thay đổi, components sẽ render và danh sách hoạt động ở màn hình NVB sẽ có đầy đủ các hoạt động như bên màn hình của NVA.

static getDerivedStateFromProps(props, prevState) {

    const { performtasks, notifications } = props;

    let state = {};

    if (notifications && notifications.associatedData && notifications.assoc iatedData.value && notifications.associatedData.value.length > 0 && performtasks && performtasks.task){

// Trường hợp thêm mới hoạt động

     if (notifications.associatedData.dataType === "createTaskAction") {

        const res = [...taskActions, notifications.associatedData.value[0]];

        console.log("noti", notifications.associatedData.value[0])

        props.refreshDataAfterCreateAction(res)

     }

}

}

case performTaskConstants.REFRESH\_DATA\_AFTER\_CREATE\_ACTION\_SUCCESS:

    return {

        ...state,

        task: {...state.task, taskActions: action.payload},

    }

### Firebase

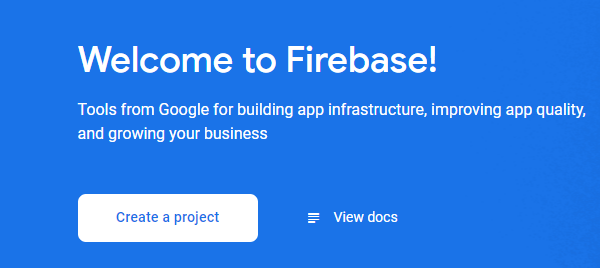
**Firebase** là một dịch vụ cơ sở dữ liệu thời gian thực được cung cấp bởi Google và hoạt động trên nền tảng đám mây, cho phép gửi thông báo real time giữa các thông báo giữa client và server qua dịch vụ Firebase Cloud Messaging (**FCM**)

#### Cách triển khai

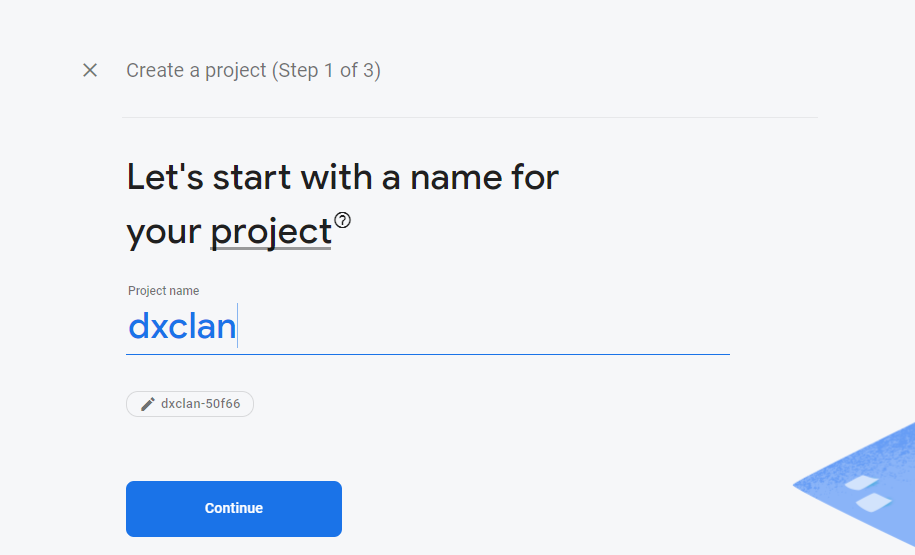
Thiết lập dự án firebase:

Truy cập vào địa chỉ <https://console.firebase.google.com/> và đăng nhập bằng tài khoản gmail.

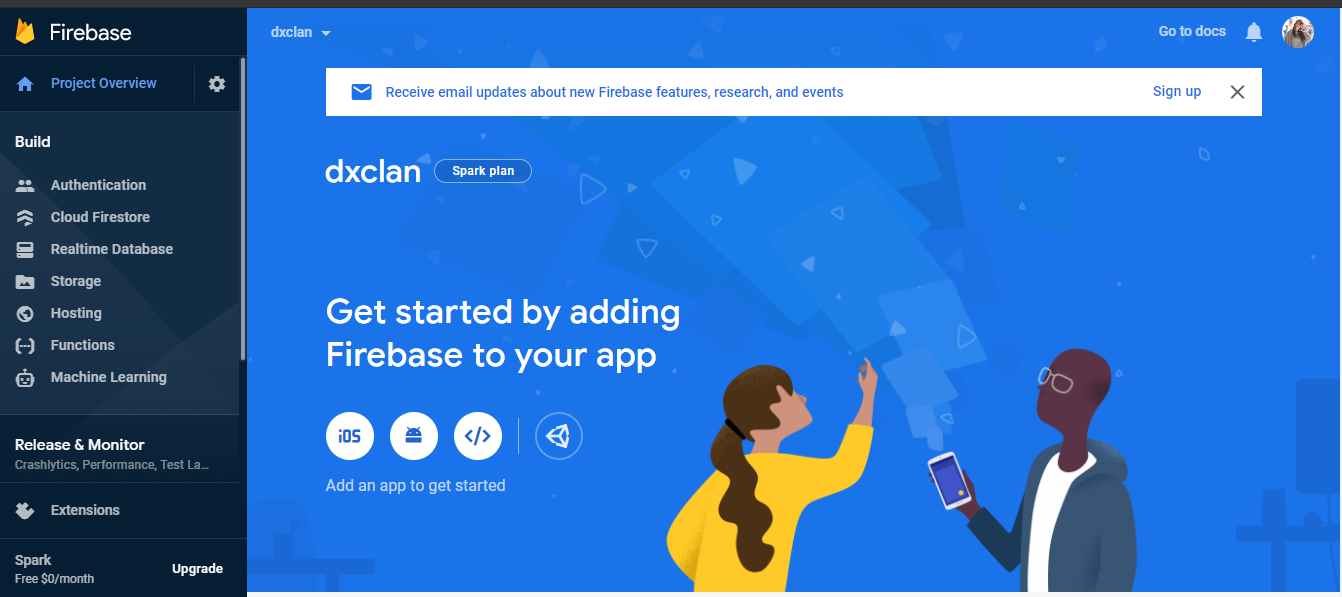
Sau đó click button “create a project” để khởi tạo dự án mới



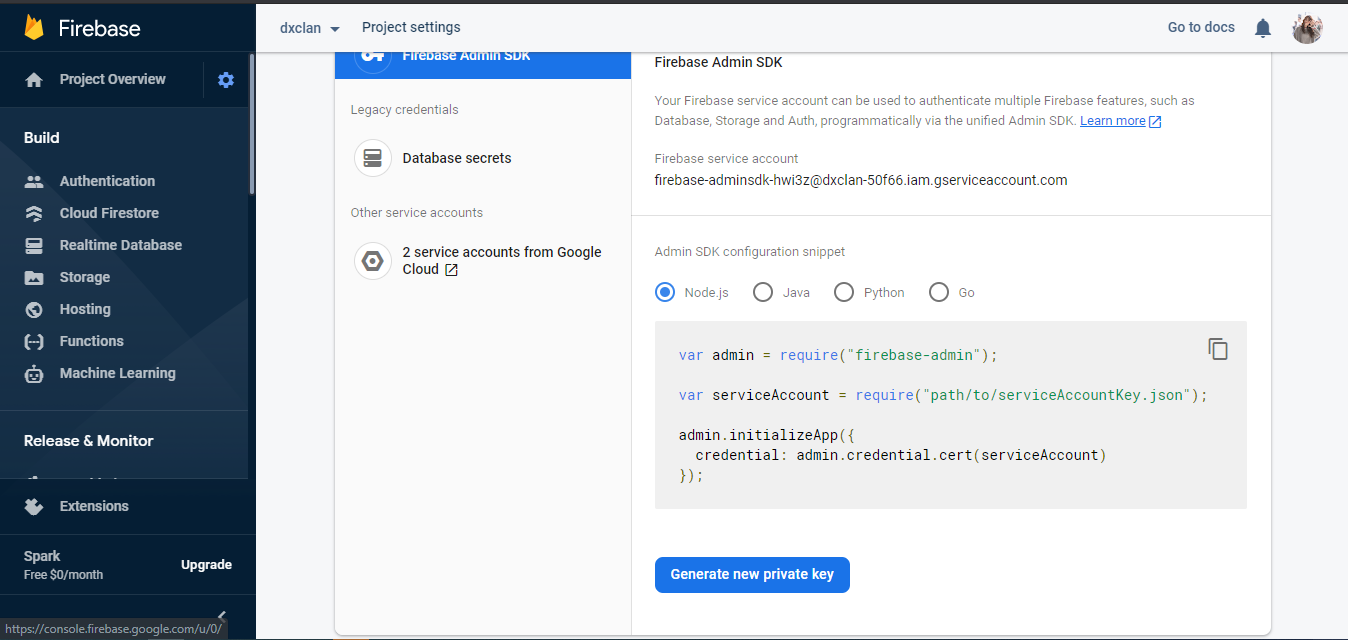
Tiếp tục nhập tên dự án và khởi tạo dự án mới và chờ cho firebase khởi tạo dự án mới.



Sau khi khởi tạo xong dự án mới. Giao diện như hình dưới:

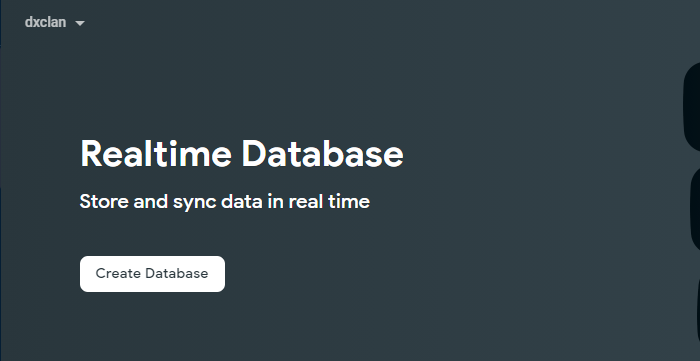


Click vào biểu tượng setting bánh răng ở góc trên bên trái, và chọn **project setting**, sau đó chuyển đến tab **service accounts** và click vào button **generate new private key**

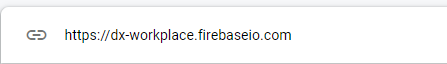


Khi click vào sẽ tự động tải xuống tệp .**json**

Tiếp tục, ở thanh sidebar bên trái click vào menu **Realtime database**



Tiến hành tạo database, next theo hướng dẫn, và url dưới chính là databaseURL



Sau khi thiết lập xong cấu hình firebase thì tiến hành cấu hình cho server

**Server nodejs:**

Firebase push notification hỗ trợ rất nhiều các **Admin SDK** trong đó có Nodejs

Cd vào thư mục server và cài đặt firebase-admin bằng lệnh: npm i firebase-admin (Đã khai báo trong file package.json)

Copy file **json** đã tải xuống lúc cấu hình firebase bên trên vào trong thư mục **server.**

Mở file global.js nằm trong thư mục **server,** import thư viện vừa được cài vào, và import file **json** trên vào file global.js

const admin = require("firebase-admin");

const serviceAccount = require("./dx-workplace-firebase-adminsdk-wxsr2-726a4a58e4.json");

Mở file **global.js** nằm trong thư mục **server,** và tiến hành khởi tao firebase

global.FIREBASE\_ADMIN = admin.initializeApp({

        credential: admin.credential.cert(serviceAccount),

        databaseURL: "https://dx-workplace.firebaseio.com",

    });

Với credential là file **json,** và **databaseURL** cũng tạo ở trên.

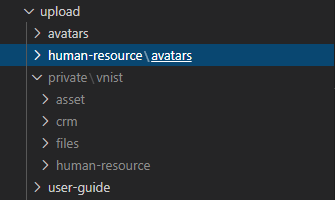
**// TODO: Còn tiếp**

## Hướng dẫn sử dụng chức năng upload file

Code upload file được thực hiện trên cả client và server. Khi client tải một/nhiều file lên máy chủ, các file đó sẽ được gửi qua một form và được mã hóa dưới dạng dữ liệu multipart/form-data. Bên server sử dụng 1 gói có tên **multer** là một middleware giúp dễ dàng xử lý dữ liệu multipart/form-data khi người dùng upload file. Sau khi multer xử lý, ta sẽ lưu đường dẫn tới file vào trong DB. Các file upload lên cần đúng chuẩn, được đặt hết vào trong thư mục upload bên server.

File sau khi tải lên server sẽ được lưu trong thư mục upload.

* File hướng dẫn sử dụng được lưu trong thư mục upload/user-guide, dùng chung cho tất cả các công ty. Thư mục này là public, không cần bảo mật. Người dùng có thể tải file về qua đường dẫn URL mà không cần đăng nhập
* Avatars người dùng lưu trong **upload/avatars/company (**với company là tên công ty, tương ứng với portal**)**. Thư mục này cũng là public.
* Tất cả các loại file tài liệu khác đều được bảo mật, do đó sẽ được lưu trong thư mục upload/private/**company (**với company là tên công ty, tương ứng với portal**)**



### Quy tắc đường dẫn lưu trên server

Với thư mục public avatars, quy tắc lưu trữ như sau (tên công ty chính là tên portal)

**upload/avatars/tên công ty** (chữ cái đều viết thường)

Ví dụ: ảnh của user1 của công ty vnist sẽ được lưu tại đường dẫn:

**upload/avatars/vnist/user1.png**

Quy tắc lưu trữ với các module private khác:

**upload/private/tên công ty/tên module/tên module con nếu có** (chữ cái đều viết thường)

Ví dụ: ảnh cơ cấu tổ chức của công ty vnist sẽ được lưu tại đường dẫn:

**upload/private/vnist/unit/org-unit.png**

### Mã nguồn client

Client đã có sẵn component hỗ trợ lấy dữ liệu file từ thẻ **input** và chuyển thành dạng blob, đó là common-component **uploadFile** (qlcv/client/src/common-component)

Form nào cần sử dụng chức năng tải file lên server thì import component này

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

Sau đó gọi component UploadFile, thêm props onChange để bắt được sự kiện người dùng thay đổi file đính kèm.

Có 2 trường hợp: upload một file và upload nhiều file.

#### Upload một file

Mặc định component UploadFile chỉ cho chọn 1 file

<div className={`form-group`}>

    <label htmlFor="file">file đính kèm</label>

    <UploadFile onChange={this.handleChangeFile} />

</div>

Hàm handleChangeFile dùng để bắt sự kiện thay đổi file.

handleChangeFile = (files) => {

    if (files && files.length > 0) {

        this.setState({

            urlFile: files[0].urlFile,

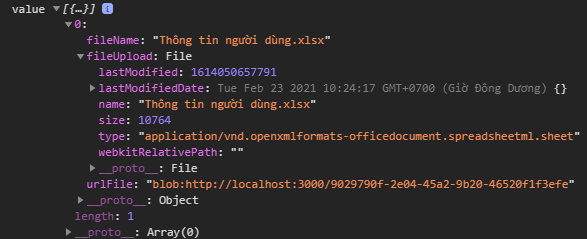
            fileUpload: files[0].fileUpload,

        })

    }

}

Tham số files truyền vào hàm handleChangeFile là một mảng các file, mỗi phần tử có 3 thuộc tính là fileName, fileUpload và urlFile. Ví dụ chi tiết nội dung như sau:



Trước khi chuyển giá trị fileUpload vừa lưu qua bên server thì phải mã hóa dưới dạng form-data, sau đó thì mới gửi sang.

Sử dụng hàm **convertJsonObjectToFormData** trong file tiện ích **jsonObjectToFormDataObjectConverter.js** nằm trong thư mục helper bên client để đính kèm dữ liệu khác cùng với file muốn upload vào form data trước khi submit.

Ví dụ, ngoài **fileUpload** ở trên ra còn có, họ tên người dùng, email,….

data = {

name: “minh”,

email: [minh@gmail.com](mailto:minh@gmail.com),

fileUpload: {…}

}

Khi đấy ta thực hiện mã hóa dưới dạng formdata như sau:

Import hàm xử lý

import { convertJsonObjectToFormData } from '../../../../../helpers/jsonObjectToFormDataObjectConverter';

Trước khi submit form, cần gọi hàm **convertJsonObjectToFormData** với tham số đầu vào là dữ liệu cần submit tới server

const dataNew = convertJsonObjectToFormData(data);

tiếp tục thêm khóa cho đối tượng fileUpload:

dataNew.append(‘avatar’, data.fileUpload);

Sau đó tiến hành submit thông tin lên server

this.props.changeInformation(dataNew);

#### Upload nhiều file

Để bật chế độ cho đính kèm nhiều file, mở ví dụ minh họa xem trong file \client\src\modules\asset\user\purchase-request\components\PurchaseRequestCreateForm.jsx

Tương tự như trường hợp upload một file, trước tiên cần import component UploadFile. Sau đó khi sử dụng component này, thêm thuộc tính **multiple = true**

<div className="form-group">

    <label>{translate('human\_resource.profile.attached\_files')}</label>

    <UploadFile multiple={true} onChange={this.handleChangeFile} />

</div>

Hàm handleChangeFile đối với chế độ multiple sẽ khác:

handleChangeFile = (files) => {

        const recommendFiles = files.map(x => ({

            url: x.urlFile,

            fileUpload: x.fileUpload

        }))

        this.setState({

recommendFiles,

        });

    }

Đầu vào là một mảng các file, mỗi file là một object. Dùng map lặp qua 1 lượt để lấy các trường cần thiết. sau đó lưu vào state.

Tương tự như TH1, trước khi lưu vào state thì cũng phải mã hóa dưới dạng form data. Ở hàm bắt sự kiện submit form, ta cũng sử dụng hàm: convertJsonObjectToFormData để xử lý dữ liệu

let formData = convertJsonObjectToFormData(dataToSubmit);

Với dataToSubmit bao gồm file muốn upload và các trường dữ liệu khác.

Vì **recommendFiles** trong dataToSubmit là một mảng nên để append thêm các file này, ta sử dụng vòng lặp như sau

recommendFiles.forEach(obj => {

       formData.append('recommendFiles', obj.fileUpload)

})

Sau đó tiến hành submit tới server : this.props.createRecommendProcure(formData);

### Mã nguồn server

##### Hướng dẫn chung

Server sử dụng thư viện **multer** để hỗ trợ xử lý files mà client gửi tới. Cách xử lý đã được viết chuẩn hóa trong **middleware**, đó là hàm **uploadFile** (nằm trong file **server/middleware/index.js**).

Hàm nhận vào 2 tham số, arrData là mảng dữ liệu chứa tên khóa (tên thuộc tính lưu trữ file trong form-data khi submit từ client) và đường dẫn sẽ lưu file, type là chế độ xử lý files

exports.uploadFile = (arrData, type) => {

    const staticPath = ["/avatars"];

    var name, arrFile;

    // Tạo folder chứa file khi chưa có folder

    const checkExistUploads = async (portal) => {

        if (portal !== undefined)

            return await arrData.forEach((x) => {

                if (staticPath.indexOf(x.path) !== -1) {

                    let dir2 = `./upload/${x.path}/${portal}`;

                    if (!fs.existsSync(dir2)) {

                        fs.mkdirSync(dir2, {

                            recursive: true,

                        });

                        fs.appendFile(dir2 + "/README.txt", "", (err) => {

                            if (err) throw err;

                        });

                    }

                } else {

                    let dir = `./upload/private/${portal}${x.path}`;

                    if (!fs.existsSync(dir)) {

                        fs.mkdirSync(dir, {

                            recursive: true,

                        });

                        fs.appendFile(dir + "/README.txt", "", (err) => {

                            if (err) throw err;

                        });

                    }

                }

            });

    };

    const getFile = multer({

        storage: multer.diskStorage({

            destination: (req, file, cb) => {

                checkExistUploads(req.portal);

                if (type === "single" || type === "array") {

                    if (staticPath.indexOf(arrData[0].path) !== -1) {

                        cb(null, `./upload${arrData[0].path}/${req.portal}`);

                    } else {

                        cb(

                            null,

                            `./upload/private/${req.portal}${arrData[0].path}`

                        );

                    }

                } else if (type === "fields") {

                    for (let n in arrData) {

                        if (file.fieldname === arrData[n].name) {

                            if (staticPath.indexOf(arrData[n].path) !== -1) {

                                cb(

                                    null,

                                    `./upload${arrData[n].path}/${req.portal}`

                                );

                            } else {

                                cb(

                                    null,

                                    `./upload/private/${req.portal}${arrData[n].path}`

                                );

                            }

                            break;

                        }

                    }

                }

            },

            filename: function (req, file, cb) {

                let extend = file.originalname.split(".");

                let oldNameFile = extend.splice(0, extend.length - 1);

                oldNameFile = oldNameFile.join(".");

                let hash =

                    `${req.user.\_id}\_${Date.now()}\_` +

                    CryptoJS.MD5(oldNameFile).toString();

                cb(null, `${hash}.${extend[extend.length - 1]}`);

            },

        }),

    });

    switch (type) {

        case "single":

            name = arrData[0].name;

            return getFile.single(name);

        case "array":

            name = arrData[0].name;

            return getFile.array(name, 20);

        case "fields":

            arrFile = arrData.map((x) => {

                return {

                    name: x.name,

                    maxCount: 20,

                };

            });

            return getFile.fields(arrFile);

        default:

            break;

    }

};

Hàm này lưu lại tệp ở 3 dạng: (1) **single -** chỉ có một trường file trong form-data submit lên và trường file đó chỉ chứa 1 file (trường hợp 1 như bên client), (2) **array** - chỉ có một trường file trong form-data submit lên nhưng trường file đó chứa nhiều file (trường hợp 2 như bên client), và cuối dùng là (3) **fields** - có nhiều trường file trong form-data submit lên.

Với chế độ single và array, arrData chỉ có 1 phần tử (các phần tử còn lại sẽ bị bỏ qua).

##### Chế độ single

Ứng với **trường hợp upload một file ở** mục 5.8.2.1, xử lý phía server như sau. Trong file user.route.js, đoạn code sau cài đặt api chỉnh sửa thông tin người dùng, trước khi chạy vào controller để xử lý thì phải chạy qua 1 middleware uploadFile

router.patch("/profile/:id/change-information", auth, authCUIP,

uploadFile([{name:'avatar', path:'/avatars'}], 'single'),

AuthController.changeInformation);

Tham số thứ nhất của hàm UploadFile là mảng gồm 1 object: name là ‘avatar’ phải trùng với **key** ta **append** vào form-data khi submit ở client, và **path**: là đường dẫn tới thư mục sẽ lưu file. Tham số thứ 2 là chế độ **single**.

Sau khi chạy qua middle UploadFile, file sẽ được lưu lại ở thư mục chuẩn theo quy định ở trên server. Sau đó các trường giá trị khác như email, name sẽ vẫn nằm trong **req.body**. Còn file avatar thì sẽ được multer xử lý và lưu ở trong **req.file**.Trong controller sẽ lấy ra đường dẫn file: req.file.path và tiến hành lưu giá trị này vào db.

##### Chế độ array

Ứng với **trường hợp upload một file ở** mục 5.8.2.2, xử lý phía server như sau

Trong file **purchase-request.route.js** quy định, api thêm mới (method POST) phải chạy qua middleware **uploadFile**

router.post('/purchase-request', auth,

uploadFile([{ name: 'recommendFiles', path: '/asset/purchase-request/files' }], 'array'), RecommendProcureController.createPurchaseRequest);

Ví dụ này khác với ví dụ trước là tham số thứ 2 của middleware upload file sẽ là ‘array’

Sau khi qua uploadFile đến tới controller, các trường dữ liệu khác sẽ nằm trong req.body. Danh sách tệp tin đính kèm sẽ lưu trong **req.files** (có s nghĩa là có nhiều tệp tin). Sau đó controller xử lý tiến hành lưu vào db.

##### Chế độ fields

Trường hợp tổng quát nhất là cho tham số thứ 2 của middleware uploadFile là ‘**fields’**. Khi đó, form-data ở client gửi lên có nhiều trường chứa file, mỗi trường có thể là một mảng các file. Ví dụ, tạo mới employee, các trường ảnh avatar, bằng đại học, giấy khai sinh, cmnd, hồ sơ xin việc, hợp đồng làm việc, … đều là các trường file độc lập nhau

Ví dụ trong server/modules/human-resource/profile/profile.route.js

const data = [{

name: 'fileAvatar',

path: '/human-resource/avatars'

},

{

name: 'fileDegree',

path: '/human-resource/degrees'

},

{

name: 'fileCertificate',

path: '/human-resource/certificates'

},

{

name: 'fileCareer',

path: '/human-resource/career'

},

{

name: 'fileMajor',

path: '/human-resource/major'

},

{

name: 'fileContract',

path: '/human-resource/contracts'

},

{

name: 'file',

path: '/human-resource/files'

},

{

name: 'healthInsuranceAttachment',

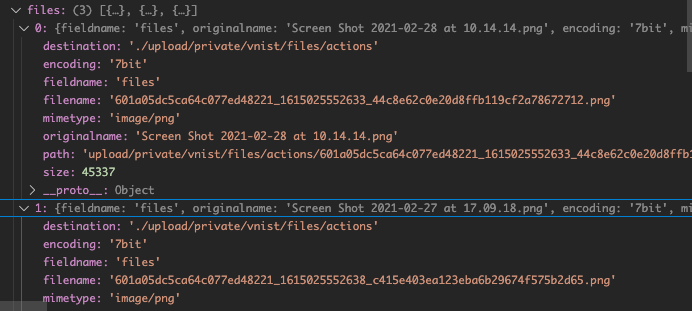
path: '/human-resource/healthInsuranceAttachment'

}

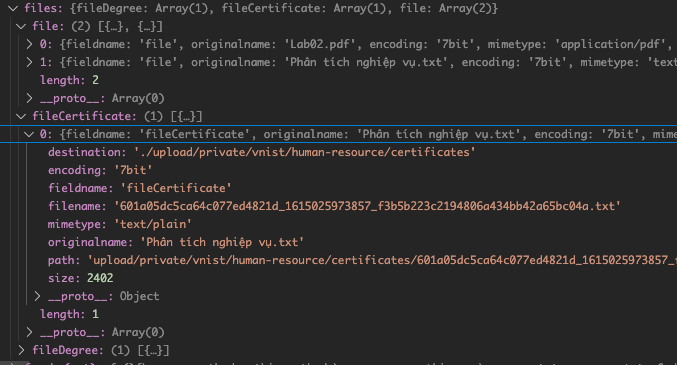
]

router.post('/employees', auth, uploadFile(data, 'fields'), EmployeeController.createEmployee);

Với ‘**array’,** **req.files** có dạng mảng các file



Với **‘fields’,** **req.files** có dạng là một đối tượng, với các thuộc tính là tên các thuộc tính gửi lên trong form-data từ client (như trong ví dụ là các thuộc tính file, fileCertificate, và fileDegree). Giá trị mỗi thuộc tính lại là một mảng các file.



## Cách dùng biến redux lưu kết quả từ nhiều lời gọi service

Bài toán đặt ra: Khi gọi một service, kết quả trả về từ service được lưu trong 1 biến redux. Các component trong cùng một trang đều gọi service đó nhưng với tham số khác nhau. Cần lưu kết quả trả về từ service cho từng component trong biến redux

Giải pháp:

Ví dụ với class component

Ví dụ với function component

VD đọc trong qlcv/client/src/modules/kpi/organizational-unit/creation/redux/reducers.js, COPY\_KPIUNIT\_REQUEST

## Đợi đủ dữ liệu mới render component

Bài toán đặt ra: Một component gọi 3 service. Đợi kết quả của cả 3 service component mới bắt đầu render?

Giải pháp:

Ví dụ với class component

Ví dụ với function component

## Cách code chuẩn tắc

Code chuẩn tắc hiểu đơn giản là một tập các quy tắc chuẩn khi lập trình một ngôn ngữ cụ thể. Những quy tắc này bao gồm: cách tổ chức file, thụt lề, hướng dẫn comment, khai báo biến, quy tắc viết câu lệnh, khoảng trắng, quy tắc đặt tên, nguyên tắc lập trình phần mềm, …

Khi tham gia một dự án lớn bắt buộc phải có một chuẩn code rõ ràng giúp team có thể đọc và sửa code cho nhau.

Đối với project hiện tại, khi tham gia vào dự án phải đảm bảo các yếu tố sau đây.

### Viết code theo đoạn

Khi viết code cần có sự rõ ràng. Viết code cũng như viết văn, nếu viết liền tù tì thì sẽ khiến người đọc cảm thấy khó theo dõi, khó đọc, mệt, không có khoảng nghỉ. Do đó các hàm cần cách nhau 2-3 dòng trống. Trong một hàm, các dòng lệnh liên quan tới nhau phải được nhóm riêng thành một đoạn code. Các đoạn code cách nhau 1-2 dòng trống.

### Comment code

Do dùng đa ngôn ngữ, cần sử dụng comment cho các thành phần giao diện trong trang

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

{/\* Tên ví dụ \*/}

<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>

{/\* Mô tả ví dụ \*/}

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

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

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

</div>

</form>

Khi viết hàm, cần comment đầy đủ cho hàm, theo đúng định dạng như sau. Tránh comment hàm sử dụng dấu //

/\*\*

\*

\*/

Ví dụ với hàm compareTime

/\*\*

 \* So sánh hai giá trị ngày nhập vào

 \* @param {\*} time1

 \* @param {\*} time2

 \* @param {\*} type

 \* Kết quả trả về: 0: không thể so sánh,

 - 1: time1 <= time2, 1: time1 > time2

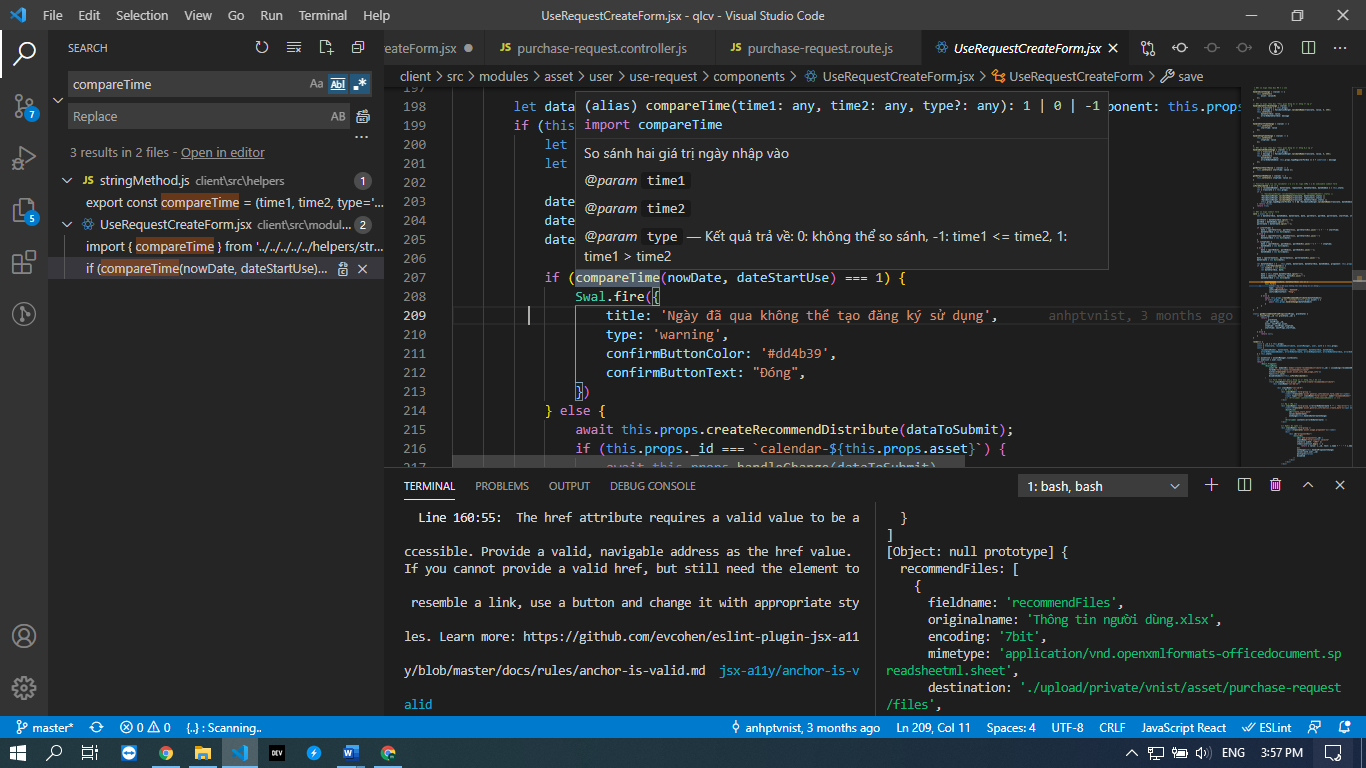
 \*/

export const compareTime = (time1, time2, type='dmy') => {

// xử lý code trong đây

}

Khi sử dụng định dạng comment trên, khi có một nơi nào đó gọi tới hàm compareTime, chỉ cần rê chuột vào tên hàm đang gọi, sẽ hiện lên popup chứa thông tin về hàm:



Ngoài ra các quy tắc khác như đặt tên biến, tên hàm, tên class, khoảng cách, đặt tên serive, …, tham khảo tạimục **3.1**

### Cách dùng biến state, props phía client

Thồng thường ở hàm render hoặc một số hàm sử dụng state và props, khi muốn lấy các giá trị trong props hoặc state, ta thường truy cập trực tiếp như this.props.users, this.state.userName,…

Các viết như này không sai, nhưng gây tới code dài, khó đọc , khó đổi tên biến về sau. Do đó, khi viết, ta cần tránh cách truy cập trực tiếp nói trên. Thay vào đó, cần sử dụng cú pháp Destructuring trong javascript ES6.

Thay vì gọi this.props.user, this.props.task, … ta sẽ viết thành let {user, task} = this.props; tương tự với khi viết với state: let {userName, password} = this.state;

### Một số lưu ý chung

So sánh dùng === thay vì ==. Ba dấu bằng: so sánh cùng kiểu dữ liệu, cùng giá trị. VD: “3” === 3 có kết quả là False. Hai dấu bằng: tự động chuyển đổi kiểu dữ liệu (nếu cần) rồi mới so sánh. VD: “3” == 3 có kết quả là True

Null khác với undefined. VD check điều kiện: if (a==null) là chưa chuẩn. Dùng if (a) sẽ chuẩn và an toàn hơn

Dùng **let/const** thay vì var. Sử dụng var: biến toàn cục, ra khỏi block, biến đó vẫn có tác dụng, dẫn đến hiệu ứng không mong muốn

Chú ý thứ tự các phương thức useState, useEffect, các phương thức handle, validate dữ liệu và các phương thức khác

Chú ý khi truy vấn thuộc tính của một đối tượng, cần kiểm tra đối tượng có là null/undefined. Ví dụ, thay vì viết obj.property, nên viết là obj?.property để tránh bị lỗi.

### Cách populate truy vấn đúng trong mã nguồn server

**Populate()** là một phương thức để liên kết các document, lấy thêm dữ liệu về một đối tượng ở document khá. Tránh lạm dụng populate lấy hết toàn bộ dữ liệu, làm dư thừa dữ liệu, dẫn tới hiệu năng kém. Nghiêm trọng hơn, có thể làm lộ thông tin của các trường quan trọng như pasword trong bảng user. Khi không chỉ định select trường nào, mặc định tất cả các trường sẽ được lấy. Vì vậy, khi dùng populate cần chú ý chỉ select những trường dữ liệu cần thiết.

Ví dụ : document **customer** reference tới document **group** và **status**. Khi thực hiện truy vấn lấy danh sách **customer** lỗi thường gặp khi tiến hành **populate** trên 2 documents này như sau.

 await Customer(connect(DB\_CONNECTION, portal)).find()

.populate('group')

 .populate('status’)

Nếu chỉ cần lấy trường name trong document **group** mà không sử dụng các trường khác, cách làm trên sẽ lấy ra tất cả các trường trong **group**, gây lãng phí tài nguyên và thiếu bảo mật. Cần khắc phục như sau:

 await Customer(connect(DB\_CONNECTION, portal)).find()

.populate({ path: 'group', select: '\_id name' })

 .populate({ path: 'status', select: '\_id name' })

Cách làm trên chỉ lấy ra trường \_id và name của document group.

Một ví dụ khác là khi sử dụng populate cho các đối tượng lồng nhau.

timesheetLogs: [

    {

        creator: {

            type: Schema.Types.ObjectId,

            ref: "User",

        },

}

],

Ví dụ **model task** có trường timesheetLogs là một mảng đối tượng, khi thực hiện lấy thông tin danh sách công việc, để populate trường creator nằm trong timesheetLogs để lấy ra tên người tạo (trường creator) ta làm như sau:

await Task(connect(DB\_CONNECTION, portal)).find()

.populate({ path: "timesheetLogs.creator", select: "name" })

Ở đây, ta sử dụng dấu . để truy cập tới đối tượng creator

## Hướng đẫn dùng song ngữ

Hiện tại project hỗ trợ 2 ngôn ngữ là tiếng Anh và tiếng Việt, lần lượt định nghĩa ở file **en.js** và **vi.js** trong thư mục client/src/lang

Cách dùng song ngữ cho chuẩn tắc như sau:

Mở file **vi.js:** Khi thêm mới một module với các page, cần thêm page vào sidebar. Để hiện thị tên trang theo song ngữ, cần thêm vào mục **menu** gồm key và giá trị là tên trang bằng tiếng Việt.

messages: {

        menu: {

            home: "Trang chủ",

            system\_administration: "Quản trị hệ thống",

            manage\_configuration: "Cấu hình hệ thống",

Nếu trong giao diện các trang trong module muốn sử dụng các text phổ biến như thêm mới, sửa, xóa, … thì cần lấy trong mục **general**. **Không định nghĩa lại song ngữ cho các text này trong module của mình**.

menu: {

…

},

general: {

            add: "Thêm",

            edit: "Sửa",

            delete: "Xóa",

            save: "Lưu",

}

Mỗi module sẽ tạo tương ứng 1 key là tên module. Đối tượng ứng với key đó được chia làm 2 phần: phần đầu chứa các key phục vụ cho hiển thị giao diện và phần sau chứa các key phục vụ cho viẹc hiển thị thông điệp trả về từ server.

Nếu module cần sử dụng các text đặc thù, cần tạo mới key chung là tên module, sau đó định nghĩa các text đặc thù trong key đó. Ví dụ với module **example**:

menu: {…},

general: {…},

manage\_example: {

            title: "Quản lý tài liệu biểu mẫu",

            version: "Tên phiên bản",

            information: "Thông tin",

            different\_versions: "Phiên bản khác",

            amount: "Số lượng",

            name: "Tên tài liệu",

}

Tránh trường hợp module này dùng key translate của module khác, dẫn tới việc bảo trì code khó khăn. Nếu có text nào dùng chung nhiều, cần chuyển vào mục key general.

File en.js cấu hình ngôn ngữ tiếng Anh cũng tương tự.

Sau khi cấu hình xong trong 2 file ngôn ngữ **en.js** và **vi.js,** ở các component sử dụng song ngữ như sau:

Lấy đối tượng translate trong props: const {  translate } = this.props;

Ở những vị trí cần hiển thị text ra giao diện, thay vì viết text bình thường, cần dùng các key translate đã định nghĩa từ trước:

Ví dụ với button tìm kiếm



Do Text là ‘Tìm kiếm’ được sử dụng ở nhiều form khác nhau nên sẽ được định nghĩa ở mục **general** {translate('general.search)}

Còn với các text khác chỉ có trong module example, ta gọi ra như sau: translate('manage\_example.exampleName'), với **'manage\_example**  là tên module và exampleName là key của text đó.

## Cơ sở dữ liệu MongoDB

### Hướng dẫn tạo mới collection

Project sử dụng MongoDB. Để tạo mới một model ta làm như sau

**Bước 1**: Tạo một file có tên example.model.js (tùy theo tên đối tượng dữ liệu)

**Bước 2**: Trong file vừa tạo, thực hiện thêm code như sau để bắt đầu tạo một **Schema**:

Sau khi khai báo **require()** mongoose, dùng phương thức khởi tạo của **Schema** để tạo ra một biến **Schema**

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;

}

Trong trường hợp trên chỉ có 2 trường, **exampleName** và **description** (String).

Sau cùng ta export Schema vừa tạo. Cần kiểm tra tra xem db đã tồn tại collection Example hay chưa. Nếu chưa có thì tạo mới, nếu có rồi thì không phải tạo mới nữa. Hàm db.model dùng để tạo thêm một collection dựa vào Schema đã định trước đó. Tham số thứ nhất là tên collection, tham số thứ 2 là Schema vừa tạo.

Database được thiết kế theo mô hình Multi tenant. Ngoài db gốc cho toàn hệ thống, mỗi công ty sẽ có một db riêng. Khi khởi động, server sẽ connect tới db gốc với tên db gốc định nghĩa bởi thuộc tính DB\_NAME trong file **server/.env.**

global.DB\_CONNECTION = mongoose.createConnection(

        `mongodb://${process.env.DB\_HOST}:${process.env.DB\_PORT || "27017"}/${process.env.DB\_NAME}`,

        connectOptions

    );

initModels(DB\_CONNECTION, models);

Sau đó server gọi hàm initModels để tạo Models cho db gốc.

Hàm initModels này có 2 tham số, tham số thứ nhất là db chứa models muốn tạo, tham số thứ hai models là danh sách các models đã được định nghĩa ở thư mục models. Nếu thấy models đã tạo, hàm này chỉ lấy ra luôn model mà không tạo lại để tối ưu hiệu năng.

exports.initModels = (db, models) => {

    for (const [key, model] of Object.entries(models)) {

        if(!db.models[key]) model(db)

    }

}

Các request đều chạy qua middleware auth. Middleware này ngoài việc xử lý xác thực, còn xác định xem db ứng với request là gì, rồi lưu tên db đó vào tham số portal đính kèm vào đối tượng req. Sau đó gọi hàm initModels để tạo model

/\*\*

 \* Xác định db truy vấn cho request

 \*/

req.portal = !req.user.company

    ? process.env.DB\_NAME

    : req.user.company.shortName;

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

Khi thực hiện truy vấn dữ liệu trên hệ thống, cần chỉ rõ truy vấn trên db nào bằng cách lấy ra tham số portal đó từ đối tượng req.

exports.getTaskLog = async (portal, taskId) => {

let task = await Task(connect(DB\_CONNECTION, portal))

.findById(taskId)

.populate("logs.creator");

return task.logs.reverse();

};

### Xóa dữ liệu

Tùy chức năng, nếu muốn người dùng xóa một document và vẫn giữ lại document đó để backup/restore lại về sau, cần dùng trường deleteSoft. Trường này có giá trị true nếu document đã bị xóa, và có giá trị là false nếu document chưa bị xóa.

### Các câu lệnh truy vấn cơ bản

MongoDB là CSDL phi quan hệ. Để join giữa các collection, đơn giản nhất là dùng câu lệnh populate, ngoài ra có thể dùng các câu lệnh aggregate[[1]](#footnote-1) trong MongoDB, đặc biệt là lệnh lookup[[2]](#footnote-2), match[[3]](#footnote-3), unwind[[4]](#footnote-4), replateRoot[[5]](#footnote-5), group[[6]](#footnote-6)

* lookup: để populate trong trường hợp dùng aggregate (do aggregate không dùng được với populate)
* match: tạo điều kiện tìm kiếm với aggregate
* unwind: mở 1mảng, mỗi phần tử thành 1 document
* replaceRoot: sau khi unwind, muốn phần tử bắt đầu là document vừa unwind
* group: chọn 1 trường, nhóm các document đang có với cùng giá trị ở trường đó thành 1 document mới.

### Hướng dẫn thiết kế CSDL

#### Hướng dẫn chung

Lưu ý khi thiết kế

* Combine objects into one document if you will use them together. Otherwise separate them (but make sure there should not be need of joins).
* Duplicate the data (but limited) because disk space is cheap as compare to compute time.
* **Do joins while write, not on read.**
* Optimize your schema for most frequent use cases.
* Do complex aggregation in the schema.

#### Cài đặt quan hệ nhiều nhiều

<https://bezkoder.com/mongodb-many-to-many-mongoose/>

Khi thiết kế các collection, để tăng hiệu năng, cần hạn chế thực hiện thao tác join giữa các collection. Cách làm là không tách thành nhiều collection, để tất cả các thông tin cần thiết trong collection, dù không đảm bảo tính ACID trong CSDL quan hệ.

VD: model Task có lưu tất cả các trường thông tin, như comments, actions, logs, … mà không tách thành collection riêng.

Để cài đặt quan hệ nhiều nhiều giữa collection A và collection B, có nhiều phương án

Cách 1: A{b:[]}

Cách 2: B{a:[]}

Cách 3: Kết hợp cách 1, cách 2. Quản lý dữ liệu khó khăn, nhưng có thể có lợi ích về hiệu năng truy vấn

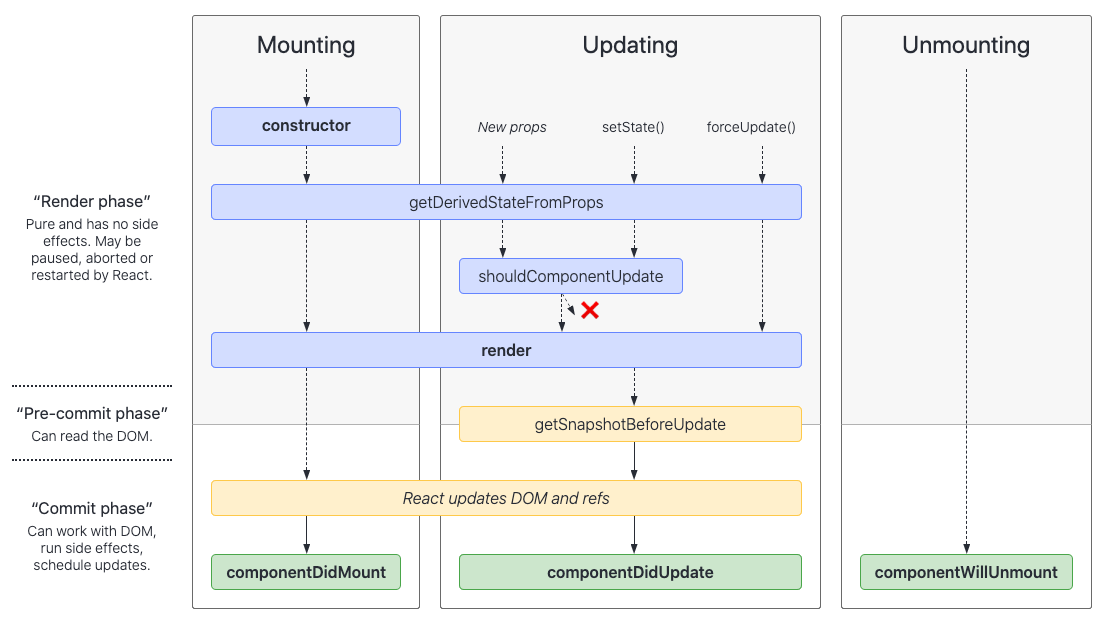
Cách 4: Tạo collection C{a: ObjectId, b: ObjectId}. Ví dụ: xem bảng UserRole

# 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[[7]](#footnote-7)) 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[[8]](#footnote-8) 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 />

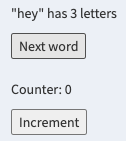
<p>Counter: {count}</p>

<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 Counter1 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}

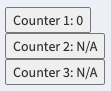
</button>

</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://docs.mongodb.com/manual/reference/method/db.collection.aggregate/ [↑](#footnote-ref-1)
2. https://docs.mongodb.com/manual/reference/operator/aggregation/lookup/ [↑](#footnote-ref-2)
3. https://docs.mongodb.com/manual/reference/operator/aggregation/match/ [↑](#footnote-ref-3)
4. https://docs.mongodb.com/manual/reference/operator/aggregation/unwind/ [↑](#footnote-ref-4)
5. https://docs.mongodb.com/manual/reference/operator/aggregation/replaceRoot/ [↑](#footnote-ref-5)
6. https://docs.mongodb.com/manual/reference/operator/aggregation/group/ [↑](#footnote-ref-6)
7. <https://github.com/facebook/react/blob/v16.8.6/packages/shared/shallowEqual.js> [↑](#footnote-ref-7)
8. <https://reactjs.org/docs/higher-order-components.html> [↑](#footnote-ref-8)