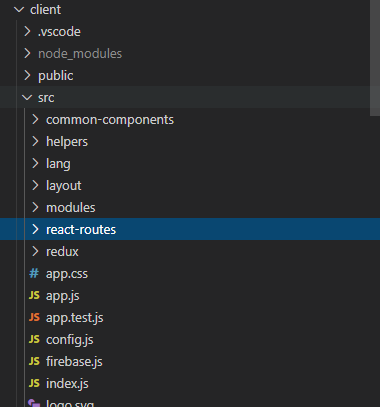
# Cấu trúc project

## Cấu trúc 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.

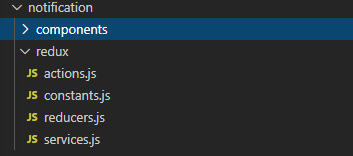
Xem mục 5.12 để biết cách làm việc chuẩn tắc với song ngữ.

### 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}

/>

### React Lazy

Ứng dụng ReactJS là 1 dạng ứng dụng web single page. Mặc định, tất cả các component trogn website được load ngay từ đầu, dù ban đầu hầu như tất cả chưa được dùng đến. Điều này khiến hiệu năng ứng dụng giảm. Để tối ưu, có thể sử dụng Lazy, giúp React chỉ load những component trong route được truy cập.

Để sử dụng Lazy, đầu tiên import các component theo cú pháp dưới (chú ý: các component phải được export dạng default)

export default connect(mapState, null)(withTranslate(Home));

Sau đó, bọc các component trong thẻ <Suspense>

Thêm thuộc tính fallback (truyền vào thuộc tính này 1 component Layout, React sẽ render component Layout này khi chưa load xong component chính từ server)

import React, { Component, Suspense, lazy } from "react";

const Home = lazy(() => import("../modules/home/components"))

const NotFound = lazy(() => import("../modules/not-found/components"))

const Document = lazy(() => import("../modules/document/components/user"))

return (

<*Suspense* *fallback*={<*Layout*/>}>

<*Switch*>

<*PrivateRoute*

*layout*={Layout}

*component*={Home}

/>

<*PrivateRoute*

*layout*={Layout}

*component*={NotFound}

/>

<*PrivateRoute*

*layout*={Layout}

*component*={Document}

/>

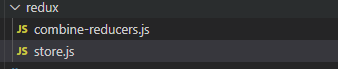
</*Switch*>

</*Suspense*>

)

### 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(

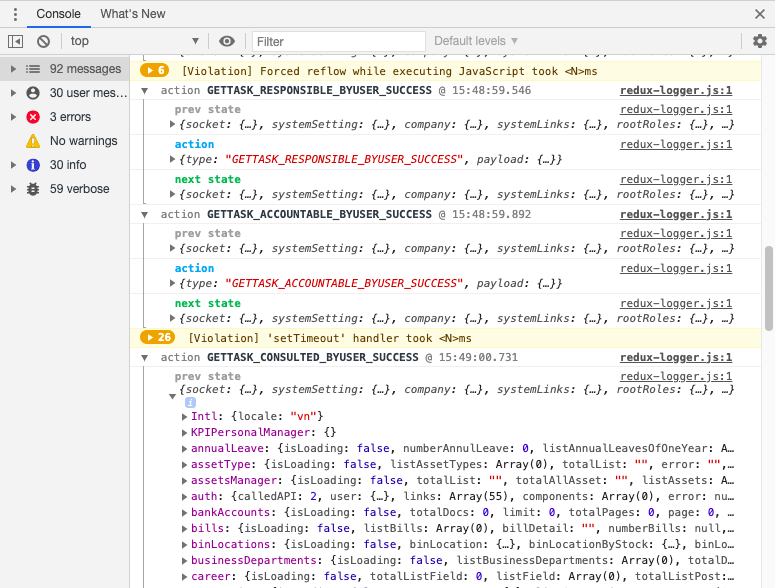
rootReducer,

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)

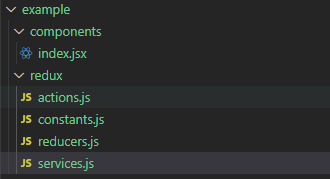
## Cách tạo mới một module

### 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",

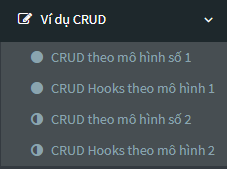
example: "ví dụ tạo mới 1 trang",

….,}

**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"}

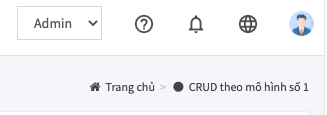
    layout={Layout}

    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';

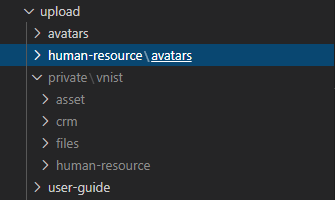
=> Tạo mới thành công common-component

## 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,

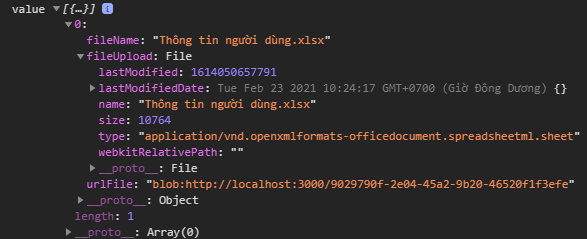
            fileUpload: files[0].fileUpload,

        })

    }

}

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',

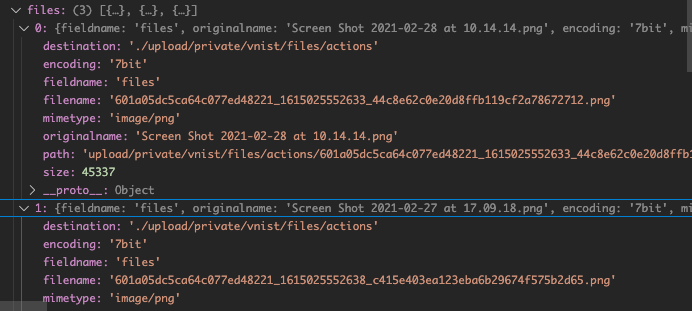
path: '/human-resource/healthInsuranceAttachment'

}

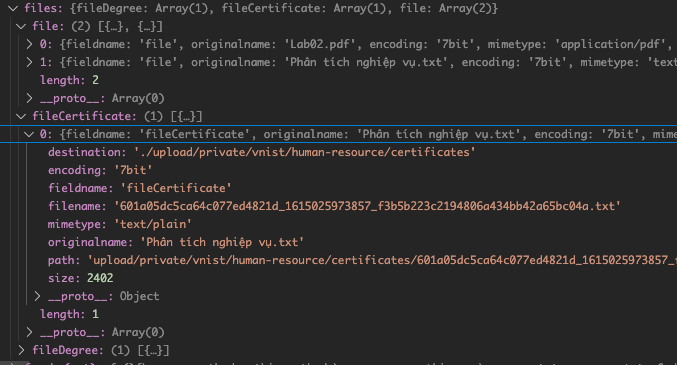
]

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.



## 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.