Mục lục

[1. Giới thiệu 2](#_Toc58177526)

[2. Lý thuyết 2](#_Toc58177527)

[2.1 Nodejs là gì? 2](#_Toc58177528)

[2.2 NPM 3](#_Toc58177529)

[2.3 Express là gì? 5](#_Toc58177530)

[2.4 ReactJS là gì? 12](#_Toc58177531)

[2.4.1 Component 12](#_Toc58177532)

[2.4.2 JSX — Javascript Syntax Extension 13](#_Toc58177533)

[2.4.3 Multiple components 13](#_Toc58177534)

[2.4.4 Props & State là gì? 14](#_Toc58177535)

[2.5 Redux là gì? 19](#_Toc58177536)

[2.5.1 Tại sao phải có redux 19](#_Toc58177537)

[2.5.2 Hiểu cách redux làm việc 21](#_Toc58177538)

[2.5.3 Nguyên lý vận hành 24](#_Toc58177539)

[3. Xây dựng dự án 24](#_Toc58177540)

[3.1 Backend 24](#_Toc58177541)

[3.1.1 Thiết kế cơ sở dữ liệu 24](#_Toc58177542)

[3.1.2 Sơ đồ diagram 29](#_Toc58177543)

[3.1.3 Công nghệ sử dụng 29](#_Toc58177544)

1. Giới thiệu

Bối cảnh: Hiện nay đại học Thủy Lợi thực hiện điểm danh sinh viên trên giảng đường hoàn toàn thủ công, gây mất thời gian giảng dạy, thời gian nghỉ giữa giờ.

Bài toán đặt ra: xây dựng hệ thống điểm danh tự động để không cần điểm danh thủ công

Hướng giải quyết: Sinh viên sẽ bật bluetooth trên điện thoại của mình, đặt tên bluetooth theo mã sinh viên. Giảng viên đến giờ vào lớp sẽ quét bluetooth tất cả các thiết bị xung quanh, và dựa vào mã sinh viên để điểm danh.

Ưu điểm: Hoàn toàn tự động, nhanh chóng, có thể ghi lại chính xác thời gian điểm danh nên ghi lại được cả các trường hợp đi muộn. Phát hiện được điểm danh hộ nếu 1 máy điểm danh 2 lần với 2 mã sinh viên khác nhau

Nhược điểm: sinh viên phải có điện thoại có bluetooth, nếu không sẽ phải điểm danh thủ công.

Công nghệ sử dụng: Hệ thống sẽ gồm 3 phần:

* Backend restful api: viết bằng nodejs-express, cơ sở dữ liệu mysql
* Frontend web để quản lý giảng viên, sinh viên, giờ học, lịch học… viết bằng reactjs
* Android app cho giảng viên thực hiện điểm danh (sinh viên chỉ cần phát bluetooth, không cần cài app), viết bằng java

1. Lý thuyết
   1. Nodejs là gì?

* Nodejs là một **nền tảng** (Platform) phát triển độc lập được xây dựng ở trên Javascript Runtime của Chrome mà chúng ta có thể xây dựng được các ứng dụng mạng một cách nhanh chóng và dễ dàng mở rộng.
* Nodejs được xây dựng và phát triển từ năm 2009, bảo trợ bởi công ty Joyent, trụ sở tại California, Hoa Kỳ. Dù sao thì chúng ta cũng nên biết qua một chút chút lịch sử của thứ mà chúng ta đang học một chút chứ nhỉ? =))
* Phần Core bên dưới của Nodejs được viết hầu hết bằng C++ nên cho tốc độ xử lý và hiệu năng khá cao.
* Nodejs tạo ra được các ứng dụng có tốc độ xử lý nhanh, realtime thời gian thực.
* Nodejs áp dụng cho các sản phẩm có lượng truy cập lớn, cần mở rộng nhanh, cần đổi mới công nghệ, hoặc tạo ra các dự án Startup nhanh nhất có thể.
  1. NPM

NMP là viết tắt của Node package manager là một công cụ tạo và quản lý các thư viện lập trình Javascript cho [Node.js](https://nodejs.org/). Trong cộng đồng Javascript, các lập trình viên chia sẻ hàng trăm nghìn các thư viện với các đoạn code đã thực hiện sẵn một chức năng nào đó. Nó giúp cho các dự án mới tránh phải viết lại các thành phần cơ bản, các thư viện lập trình hay thậm chí cả các [framework](https://topdev.vn/blog/framework-la-gi/).

Với npm , công việc sẽ đơn giản đi rất nhiều, chúng giúp bạn thực hiện việc quản lý đơn giản hơn rất nhiều. Các thư viện đều có sẵn trên npm, bạn chạy một dòng lệnh để tải về và dễ dàng include chúng hơn.

Mỗi đoạn code này có thể phụ thuộc vào rất nhiều các mã nguồn mở khác, thật may mắn khi các công cụ quản lý thư viện ra đời, nếu không sẽ mất rất nhiều công sức trong việc quản lý các thư viện này.

Cộng đồng sử dụng npm rất lớn, hàng nghìn các thư viện được phát hành, hỗ trợ Javascript ES6, React, Express, Grunt, Duo… Hiện nay cũng đã xuất hiện thêm Yarn một công cụ tương tự npm, được Facebook phát triển với nhiều tính năng vượt trội có khả năng sẽ thay thế npm.

Nếu như bạn từng code Php thì sẽ biết [Composer](https://topdev.vn/blog/composer-la-gi/) là công cụ quản lý thư viện của nó, tương tự như NPM là công cụ quản lý thư viện Javascript.

**Cài đặt NPM**:

npm có sẵn khi bạn tải NodeJS về. Để kiểm tra xem trên hệ thống của bạn đã được cài npm chưa chúng ta sử dụng lệnh npm -v, nếu một phiên bản hiện ra thì hệ thống của bạn đã được cài đặt npm.

Vì **NPM** là một phần mềm cài đặt trên máy tính của bạn nên bạn có thể sử dụng nó để cài đặt các thư viện **Javascript** từ trên **Internet**. Để cài đặt một thư viện nào đó, chỉ cần mở cửa sổ **Terminal (hoặc CMD)** và thực thi lệnh giống dưới đây:

npm install package-name

VD như mình thử tải Vuejs về sử dụng sẽ dùng lệnh:

npm install vue

Khi đó muốn sử dụng Vue.js chúng ta chỉ cần sử dụng lệnh require():

var Vue = require('vue');

**Cài đặt global và cài đặt local**

**Có hai cách để cài đặt một gói bằng npm:**

* **Local:** sẽ tạo ra thư mục node\_modules nếu chưa có trong project hoặc nếu có rồi nó sẽ lấy code của gói cần cài đặt đưa vào đây, tức chỉ hiện diện trong thư mục của project hiện tại. Khi cần sử dụng bạn có thể sử dụng lệnh require().
* **Global:** sẽ lưu trữ code của gói trong các file hệ thống cố định trong máy, chỉ có thể dùng các package này thông qua các hàm CLI (Command Line Interface) ví dụ như gulp. Không thể dùng package thông qua require().

**Mặc định thì các package khi cài đặt đều sẽ là cài trên project của bạn.**

Các package thư viện đưa vào project của bạn có thể liên tục có update mới. Thực hiện npm update để thực hiện cập nhật tất cả các gói liên quan. Nếu bạn chỉ muốn cập nhật một gói cụ thể có thể sử dụng cú pháp:

npm update package\_name

**Package.json**

Để quản lý các gói cài đặt cục bộ bằng npm thì cách tốt nhất là thông qua file package.json, chính là file nằm trong thư mục gốc của project. File [JSON](https://topdev.vn/blog/json-la-gi/) này chứa các nội dung:

* Các gói thư viện lập trình mà project sử dụng.
* Cho phép xác định phiên bản chính xác của các gói thư viện lập trình được sử dụng.
* Các gói bạn xây dựng có thể chia sẻ dễ dàng với các lập trình viên khác trên toàn cầu thông qua npm.

Lệnh npm init –yes sẽ tạo ra file package.json mẫu.

npm init --yes

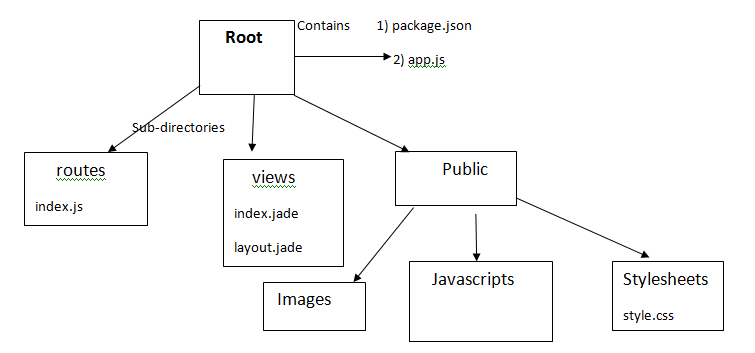
Có một số các thuộc tính trong package.json, chúng ta cùng điểm qua:

* + - **name**: tên gói thư viện
    - **version**: phiên bản gói
    - **description**: phần mô tả về gói thư viện
    - **homepage**: trang chủ của gói
    - **author**: tác giả
    - **contributors**: tên người đóng góp cho package
    - **dependencies**: danh sách các gói phụ thuộc, tự động được cài theo.
    - **repository**: loại repository và url của package, thông thường là git (Xem thêm [Git là gì](https://topdev.vn/blog/git-la-gi/)?)
    - **main**: index.js
    - **keywords**: các từ khóa

Nếu muốn thêm các entry vào thuộc tính dependencies khi cài đặt gói sử dụng thêm flag –save, còn với thuộc tính devDependencies thì sử dụng flag –save-dev.

* 1. Express là gì?

**Expressjs** là một framework được xây dựng trên nền tảng của **Nodejs**. Nó cung cấp các tính năng mạnh mẽ để phát triển web hoặc mobile. **Expressjs** hỗ trợ các method HTTP và midleware tạo ra API vô cùng mạnh mẽ và dễ sử dụng.



*Tổng hợp một số chức năng chính của Expressjs như sau:*

* Thiết lập các lớp trung gian để trả về các HTTP request.
* Define router cho phép sử dụng với các hành động khác nhau dựa trên phương thức HTTP và URL.
* Cho phép trả về các trang HTML dựa vào các tham số.

Dưới đây là một ví dụng để bạn sẽ hiểu được chức năng của mỗi phần trong 1 ứng dụng Express cơ bản.

**Bước 1: thuê Manager (require statements)**

Nếu muốn nhà hàng hoạt động hiệu quả, bạn sẽ cần 1 người giúp nhân viên làm việc hiệu quả ở mức tối đa. [Express](http://blog.topdev.vn/) sẽ là “nhân vật” Manager như thế.

Cũng giống như bất kì package NPM nào khác, bạn cần phải npm install module express, sau đó sử dụng lệnh **require**để load module.

const express = reqire(‘express’);

const app = express();

**Bước 2: đưa ra các quyết định quản lý ở nhà hàng (middleware)**

**Theo bạn, đâu là hoạt động thường diễn ra ở mọi nhà hàng? Đó sẽ là 3 hoạt động sau:**

* + - Mời khách đến ghế ngồi
    - Lấy order
    - Tính tiền

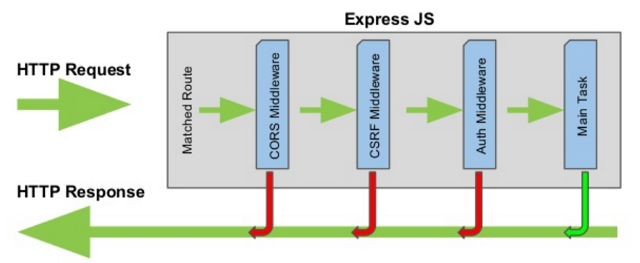
**Với mỗi hoạt động như vậy, sẽ có 1 loạt thứ phải chạy trước khi thực hiện được hoạt động đó. Ví dụ, trước khi mời khách vào ngồi, bạn cần biết:**

* + - Họ có đang mặc sơ mi và mang giày không? Nếu không thì sẽ không được ngồi.
    - Nếu họ muốn ngồi ở quán bar thì đã đủ 21 tuổi chưa? (nếu bạn đang ở Mỹ)

Tương tự với code, bạn sẽ cần phải xác nhận rằng các requests đảm bảo 1 chuẩn mực nào đó trước khi tiếp tục. Ví dụ, nếu ai đó cố gắng đăng nhập vào site thì:

* Họ có tài khoản chưa?
* Họ nhập password đúng chưa?

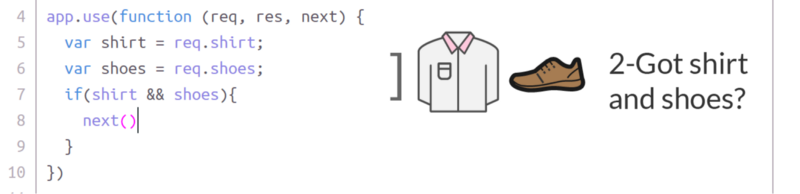
Đây là lý do concept **middleware** đã xuất hiện. Các hàm Middleware cho phép chúng ta thực hiện hành động trước request bất kì và modify nó trước khi gửi lại response.



Trong nhà hàng của bạn, cần hàng loạt các nguyên tắc để đưa ra quyết định xem liệu bạn có nên mời khách vào ghế hay không. Nguyên tắc ở đây là: họ có mang áo sơ mi và giày không?

Đầu tiên, hãy bắt đầu với [app.use()](http://expressjs.com/en/api.html#app.use). Đây là những rules đơn giản cần phải áp dụng để thực hiện các bước tiếp theo. Những rule này không phải GET, POST, PUT hay DELETE.

Ở line 4, có 1 hàm ẩn danh với các thông số req, res và next. Xét trong code block này, bạn chỉ đang kiểm tra request (req) để xem liệu có áo sơ mi & giày hay không.



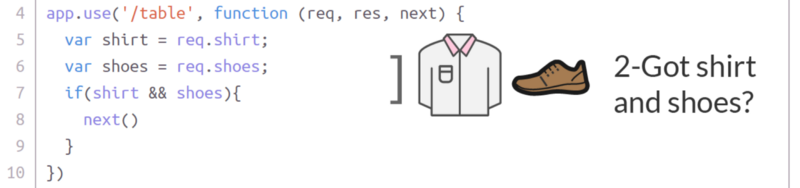
Cuối cùng, bạn cũng cần phải sử dụng hàm next() vì ở đây bạn chỉ đang xác nhận trang phục. Các bước tiếp theo, bạn sẽ cho phép khách được thực sự vào bàn ăn.

Ở line 5 & 6, bạn sẽ kiểm tra liệu các khách hàng đó có mặc áo và mang giày không.

Và ở line 7–9, bạn chỉ tiếp tục nếu các vị khách có cả 2 là mặc áo & mang giày.

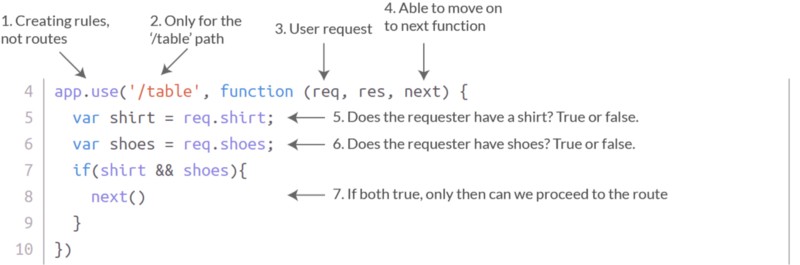
Code block ở trên bị thiếu 1 thứ quan trọng là **path**. Đây là string riêng đi chung với request. Và bởi vì thiếu path, nó sẽ chạy theo mỗi single request.

Bạn có thể tưởng tượng được không? Khi khách vào nhà hàng, gọi món, tính tiền, nhân viên sẽ phải nhìn họ trên dưới để đảm bảo là họ có mặc đồ! Nghe thật vô lý & trái với nguyên tắc kinh doanh.



Vì vậy, chúng ta đã thay đổi line 4 trong ví dụ ở trên. Bây giờ, chúng ta chỉ chạy đoạn code này khi 1 user truy vấn cùng route ‘/table’.

Chi tiết có thể xem hình bên dưới:



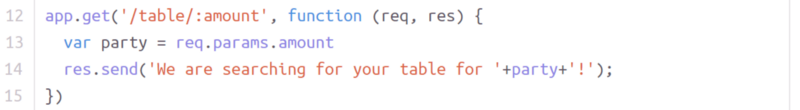
**Bước 3: thực hiện các routines (routing)**

Tiếp tục với ví dụ về việc mời vào chỗ ngồi. Cho đến lúc này, chúng ta chỉ biết được cách để xác thực liệu ai đó có nên được ngồi vào ghế hay không. Nhưng tôi thực sự không biết cách nào để đưa họ vào bàn và mời họ ngồi.

Chính lúc này, **routes** đã xuất hiện. Routes cho phép chúng ta soạn thảo (script) các hành động dựa trên **path**. Các options sẽ là GET, POST, PUT & DELETE, nhưng hiện tại chúng ta sẽ tập trung vào GET và POST.

Trong bối cảnh nhà hàng, chúng ta cần phải tạo 1 request GET để chọn 1 bàn ăn cụ thể và mời khách ngồi. GETs không modify hay thêm vào database của bạn. Chúng chỉ lấy thông tin dựa trên các thông số cụ thể.

Trong trường hợp này, bạn sẽ cần tạo 1 quy trình để sắp xếp chỗ ngồi cho 1 buổi tiệc với 2 khách. Số 2 xuất phát từ **request**của khách hàng.



Đúng vậy, đây chỉ là hành động gửi đi 1 tin nhắn vào lúc cuối. Nó vẫn chưa thực sự tìm được 1 bàn cụ thể để mời khách ngồi. Tôi sẽ cần phải tìm 1 array cho 1 bàn ăn còn trống và xử lý tình huống bên lề – tình huống nằm ngoài tầm giải quyết của tutorial này.

Ở line 12, chúng ta define quy trình này để tìm kiếm bàn ăn khi 1 vị khách **requests**cùng **route** ‘/table’. Cũng giống như ví dụ về middleware ở trên, chúng ta có sẵn các thông số request & response, có 1 **parameter**mà trong ví dụ này là hai.

Trên thực tế, mọi thứ đằng sau function declaration ở line 12 về mặt kỹ thuật là **middleware** vì nó modify user request. Bạn sẽ thấy điều đó trong biểu đồ ở dưới.

Ở line 13, chúng ta tiếp cận 1 số lượng người trong bữa tiệc từ các **parameters**của request object. Nó không được declare ở bất kì đâu vì request đến từ user và chúng ta không có dòng code frontend nào. Vì vậy nếu đây là 1 ứng dụng thực sự thì request sẽ trông như thế này:

req = {

params: {

amount: 2;

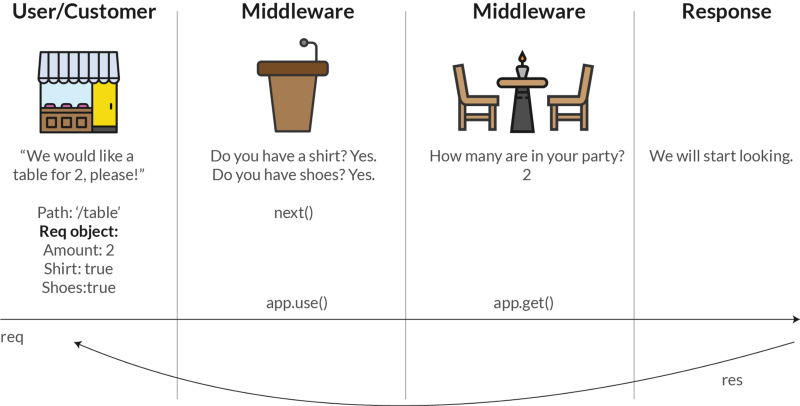
}

}

ở line 13, biến party tiếp cận **property** của **object** params trong **request**ở trên.

Cuối cùng ở line 14, chúng ta gửi 1 **response** trở lại với khách hàng: chúng ta đang tìm kiếm bàn ăn có kích cỡ vừa phải.

Đây là biểu đồ tóm gọn lại mọi thứ:



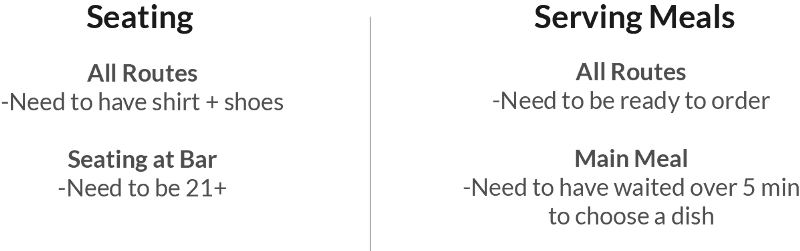
**Bước 3.5: khai thác tối đa nhà hàng (router)**

Bây giờ, bạn có thể truy dấu vết của toàn bộ đường đi từ request đến response. Nhưng khi app của bạn tăng về kích cỡ, bạn sẽ không muốn code những nguyên tắc này độc lập cho mỗi route. Bạn sẽ nhận ra 1 vài routes chia sẻ cùng các rules, vì vậy cần phải tìm cách để ứng dụng 1 set các rules vào nhiều routes.

Nói riêng việc mời khách vào chỗ ngồi, bạn có thể mời khách vào quầy bar hoặc vào bàn ăn. Chúng đều có những nguyên tắc chung như áo sơ mi + giày, nhưng muốn ngồi ở quầy bar thì các thành viên của bữa tiệc phải đủ 21 tuổi.

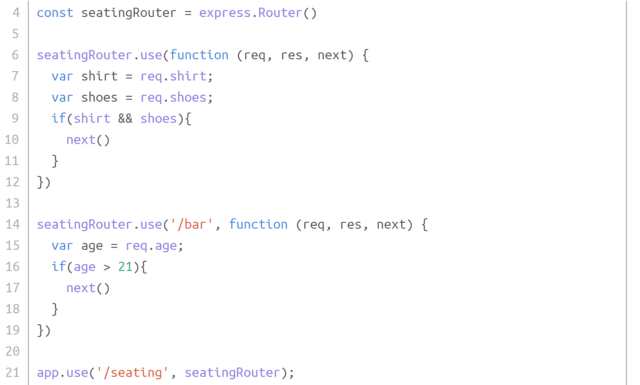
Và, xét trên câu chuyện phục vụ khách hàng, bạn sẽ cần phải dùng đến 1 quy trình khác đi 1 chút như phục vụ món khai vị, món chính và yến tiệc. Tuy nhiên, 3 routes đó đều có nhiều điểm giống nhau.

Lúc này, **router**sẽ giúp bạn bằng cách nhóm các routes để từ đây bạn có thể tạo các nguyên tắc chung.



Chúng ta cần phải tạo middleware để giải quyết hết mỗi trường hợp này. Bạn sẽ chỉ cần xử lý các trường hợp mời chỗ ngồi vì nó sẽ overwrite đoạn code ở trên.

Trích dẫn toàn bộ code ở trên:



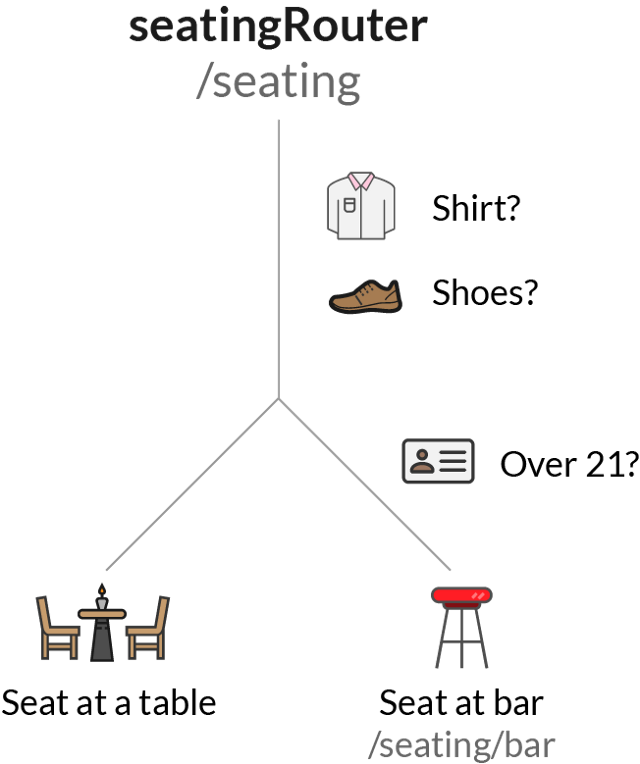
Tôi sẽ giải thích riêng biệt mỗi phần.

Ở line 4, chúng ta declare router.

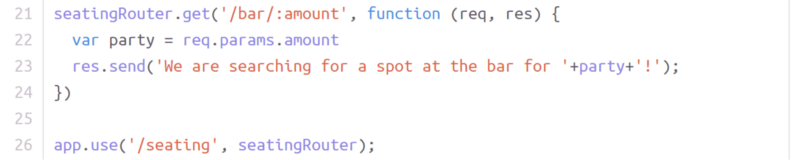
Ở lines 6 & 14, chúng ta hiện có seatingRouter.use() thay thế cho app.use() để chỉ ra rằng **middleware** này chỉ liên quan đến các routes seatingRouter.

Cuối cùng, ở line 21, chúng ta thêm middleware, cho thấy rằng mỗi seatingRouter route bắt đầu với ‘/seating’. Vì vậy, nếu ai đó yêu cầu 1 ghế ngồi ở quầy bar, toàn bộ path sẽ là ‘/seating/bar.’ Nghe có vẻ lệch quy trình nhưng bạn có thể đang mong là path sẽ được define khi bạn tạo router ở line 4.

Bình thường thôi, hãy xem form biểu đồ sau:



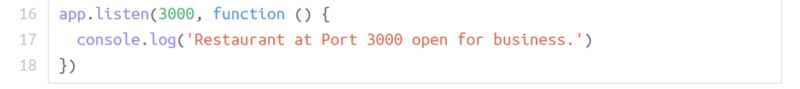
Và, khi thêm 1 route GET, nó sẽ ở trên statement cuối – nơi bạn chỉ định các routes đến router.



**Bước 4: bắt đầu kinh doanh (ports)**

Bạn đã thuê manager, xác định những gì phải làm trước khi chấp nhận customer requests, và xác định những gì phải làm với các customer request cụ thể khi chúng xuất hiện. Bây giờ, chúng ta chỉ cần xác định chính xác địa chỉ cho địa điểm diễn ra tất cả các bước này.

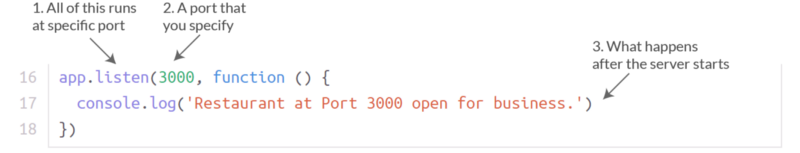
Server của bạn có các **ports,**tương tự như địa chỉ của chính nhà hàng. Vì server có thể xử lý nhiều loại nhà hàng (hoặc các script server-side) cùng 1 lúc, nên bạn cần phải cho server biết nơi mà mỗi script nên chạy.



Trong ví dụ ở trên, port là 3000 và được định vị trong máy tính của bạn. Vì vậy, nếu bạn gõ:

<https://localhost:3000/>

vào browser của mình, và bạn đang chạy app Node, server sẽ biết cách chạy script cụ thể. Trong trường hợp này, ngay khi nhập URL, bạn sẽ nhập message vào console và có thể sử dụng **các routes**bất kì. Nếu bản thân nhà hàng là toàn bộ ứng dụng của bạn thì hiện tại nó đã bắt đầu kinh doanh ở địa chỉ 3000.



* 1. ReactJS là gì?

React.JS là một thư viện Javascript dùng để xây dựng giao diện người dùng. Với cá nhân tôi cũng như nhận xét chung của cộng đồng về ReactJS thì nó nhanh, dễ học và vui.

Tiếp theo chúng ta sẽ bắt đầu đến với những khái niệm.

* + 1. Component

React được xây dựng xung quanh các component, chứ không dùng template như các framework khác. Bạn có thể tạo ra một component bằng các gọi phương thức createClass của đối tượng React, điểm bắt đầu khi tiếp cận với thư viện này.

Ví dụ.

var Button = React.createClass({

render: function(){

return (

<input type="submit" />

);

}

});

Phương thức createClass nhận vào một tham số, là đối tượng mô tả đặc tính của component. Đối tượng này bao gồm tất cả các phương thức để hình thành nên component. Phương thức quan trọng nhất là render, phương thức này được trigger khi component đã sẵn sàng để được render lên trên page.

Trong hàm đó, bạn ẽ trả về một mô tả cho việc bạn muốn React render cái gì lên trên page. Như trong ví dụ ở trên, đơn giản tôi muốn render một button.

Chú ý: Hàm render chính là mô tả cụ thể của UI tại bất cứ thời điểm nào. Vì thế nếu dữ liệu thay đổi, React sẽ take care việc update UI với dữ liệu tương ứng. Các bạn có thể hiểu đơn giản là, khi dữ liệu thay đổi, React sẽ tự động gọi hàm render để update lại UI.

### **2.4.2 JSX — Javascript Syntax Extension**

Đây đơn giản là một syntax extension của Javascript. Với nó bạn có thể viết Javascript với những tag giống như XML (XML-like). Về bản chất, các tag thực sự là những lời gọi hàm, sẽ được chuyển đổi trong React code và end up dưới dạng HTML và Javascript trong cây DOM.

Nhưng với những gì bạn biết ở hiện tại, chỉ cần hiểu đơn giản nó giống như là HTML/XML với một số khả năng khác.

* + 1. Multiple components

Nếu bạn muốn lồng nhiều component vào nhau, bạn sẽ làm điều này trong lệnh return của phương thức render.

Ví dụ.

var Form = React.createClass({

render: function(){

return (

<div>

<input type="submit" onClick={this.props.onUserClick} />

<h3>You have pressed the button {this.props.counter} times!</h3>

</div>

);

}

});

var App = React.createClass({

getInitialState: function(){

return {

counter: 0

}

},

onUserClick: function(){

var newCount = this.state.counter += 1;

this.setState({

counter: newCount

});

},

render: function(){

return (

<div>

<h1> Welcome to the counter app!</h1>

<Form counter={this.state.counter} onUserClick={this.onUserClick} />

</div>

);

}

});

React.render(<App />, document.getElementById("app"));

Phía trên, tôi đang lồng Form component vào trong App component. Đây là một dạng quan hệ cha con (parent-child) mà bạn có thể dễ dàng nhận thấy trong HTML

Phương thức React.render() như các bạn thấy ở trên nhằm mục đích kickstart việc render, và render thừ root component, trong trường hợp trên là App vào trong DOM với container cụ thể là element có id là app

* + 1. Props & State là gì?

Có hai kiểu của data trong React đó là props và state. Sự khác biệt giữa hai kiểu thì hơi khó khăn để hiểu ngay từ ban đầu, ít nhất là về mặt khái niêm. Nhưng một khi bạn bắt đầu code, bạn sẽ nhanh chóng tách biệt được hai loại.

Điểm mấu chốt của sự khác nhau là state thì private và chỉ có thể được thay đổi bên trong bản thân component. Props thì mang tính external, và không bị kiểm soát bởi bản thân component. Nó được truyền từ component cao hơn theo phân cấp, hay có thể hiểu đơn giản là truyền từ component cha xuống component con, cái mà điều khiển dữ liệu trước khi truyền xuống.

Vì thế trong khi một component không thể thay đổi props của nó một cách trức tiếp (điều này có thể làm một cách gián tiếp nhưng hãy để nó vào những phần sau), thì nó có thể tự thay đổi state của bản thân.

**Props**

Nào cùng cài đặt app của chúng ta đã làm trong bài trước với một ít dữ liệu, sử dụng props nhé. Đầu tiên chúng ta cần lấy dữ liệu từ một nơi nào đó. Đó có thể là Ajax call để lấy một số dữ liệu từ API, tuy nhiên chúng ta sẽ hard code nó như một variable.

var text = "Click the button";

Cách để đưa props vào một component nhìn rất giống cách mà chúng ta khai báo attribute cho một HTML element.

<App text={text} />

Lý do chúng ta sử dụng cặp ngoặc nhọn là vì chúng ta cần nói cho JSX biết rằng đó là một Javascript expression.

Một khi App component được cài đặt như thế này, nó có thể truy xuất vào biến text mà ta đã khai báo ở trên thông qua lời gọi this.props.text. Tuy nhiên, nó không thể trực tiếp thay đổi dữ liệu. Từ góc nhìn của component, props của nó là bất biến (immutable). Nó chỉ là thông tin được cài đặt cho component.

Đây là ví dụ.

var text = "Click the button";

var Form = React.createClass({

render: function(){

return (

<div>

<h3>{this.props.text}</h3>

<input type="submit" />

</div>

);

}

});

var App = React.createClass({

render: function(){

return (

<div>

<h1> Welcome to my app!</h1>

<Form text={this.props.text}/>

</div>

);

}

});

React.render(<App text={text}/>, document.getElementById("app"));

Như các bạn thấy, props được truyền vào trong App component trong phương thức React.render(). Sau đó App component có thể truy xuất biến text thông qua lời gọi this.props.text. Nó cũng có thể truyền dữ liệu xuống component con của nó như chúng ta thấy cách mà Form component được App component cài đặt props trong ví dụ.

Khi dữ liệu đến được Form component, chúng ta thấy đây là điểm kết thúc, dữ liệu sẽ được render ra thẻ h3 như trên.

Đây là cách mà dữ liệu được luân chuyển trong React thông qua props.

**State**

Một cách khác để storing dữ liệu trong React là state. Không giống như props, bất biến dưỡi góc nhìn của component thì state có thể thay đổi (mutable).

Vì thế nếu bạn muốn dữ liệu trong ứng dụng thay đổi, ví dụ như dựa trên tương tác người dùng, thì dữ liệu phải được lưu trữ trong component state.

State là private và được quản lý bởi chỉ duy nhất một component, nó không thể truyền xuống cho component con. Nếu bạn muốn truyền xuống cho component con thì bạn phải truyền nó như là một props.

Để cài đặt state, đơn giản chúng ta cài đặt hàm getInitialState() vào component, và trả về bất cứ gì bạn muốn cài đặt trong state của component đó.

Để thay đổi state, đơn giản ta gọi hàm this.setState(), và truyền vào state mới như là một tham số.

Ví dụ.

var App = React.createClass({

getInitialState: function(){

return {

active: true

}

},

handleClick: function(){

this.setState({

active: !this.state.active

});

},

render: function(){

var buttonSwitch = this.state.active ? "On" : "Off";

return (

<div>

<p>Click the button!</p>

<input type="submit" onClick={this.handleClick} />

<p>{buttonSwitch}</p>

</div>

);

}

});

React.render(<App />, document.getElementById("app"));

Đoạn code trên cũng cho bạn làm quen với hệ thống event trong React, rất đơn giản. Chúng ta hook một event listener vào trong button, ở trên là onClick. Khi nó được trigger, chúng ta gọi hàm handleClick, cái mà đã được cài đặt trước đó, và luôn sẵn sàng được gọi thông qua từ khóa this.

Trong hàm handleClick, chúng ta gọi this.setState(), cái mà sẽ thay đổi trạng thái của component.

Chú ý: React event được wrap để chạy trên tất cả các browser, có nghĩa là React giúp bạn đảm bảo event của bạn chạy được trên tất cả các trình duyện.

Chúng ta nên giữ state ở đâu?

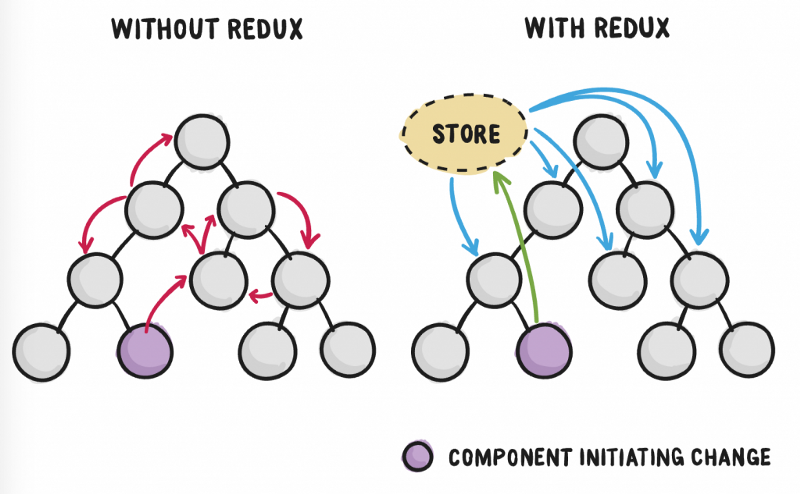
Bạn nên cố gắng giữ số lượng các stateful component ít nhất có thể, và thậm chí giữ tối thiểu lượng dữ liệu trong state. Nếu component cấp dưới cần truy xuất dữ liệu từ state, thì hãy truyền nó thông qua props.

Lưu ý: Stateful component thì luôn luôn là higher level, trong khi Stateless component thường là lower level trong hệ thống phân cấp.

Để hình dung việc state được giữ ở đâu, bạn hãy hỏi bản thân một vài câu hỏi, những câu hỏi này được lấy từ React docs:

* Xác định mỗi component mà render thông tin gì đó dựa trên state.
* Tìm một component mà nó chủ sở hữu chung của các component khác (một component nằm bên trên tất cả các component khác trong hệ thống phân cấp thì cần có state)
* Hoặc là những component là chủ sở hữu chung hoặc là những component nằm trên hệ thống phân cấp sẽ nên giữ state.
* Nếu bạn không thể tìm ra component nào phù hợp, hãy tạo một component mới đơn giản giữ nhiệm vụ lưu trữ state và đặt nó đâu đó nằm bên trên các component là chủ sở hữu chung trong hệ thống phân cấp.
  1. Redux là gì?

**Redux** là một predictable state management tool cho các ứng dụng Javascript. Nó giúp bạn viết các ứng dụng hoạt động một cách nhất quán, chạy trong các môi trường khác nhau (client, server, and native) và dễ dàng để test. [**Redux**](https://topdev.vn/blog/?s=redux) ra đời lấy cảm hứng từ tư tưởng của ngôn ngữ **Elm** và kiến trúc **Flux**của Facebook. Do vậy Redux thường dùng kết hợp với React.



* + 1. Tại sao phải có redux

Do yêu cầu cho các ứng dụng single-page sử dụng Javascript ngày càng trở lên phức tạp thì code của chúng ta phải quản lý nhiều state hơn.

State có thể bao gồm là data trả về từ phía Server và được cached lại hay như dữ liệu được tạo ra và thao tác ở phía client mà chưa được đẩy lên phía server. UI state cũng trở lên phức tạp vì chúng ta cần quản lý việc active Routes, selected tabs, spinners, điều khiển phân trang …vv.

Với Redux, state của ứng dụng được giữ trong một nơi gọi là store và mỗi componentđều có thể access bất kỳ state nào mà chúng muốn từ chúng store này.

Hầu hết các lib như React, Angular, etc được built theo một cách sao cho các components đến việc quản lý nội bộ các state của chúng mà không cần bất kỳ một thư viện or tool nào từ bên ngoài.

Nó sẽ hoạt động tốt với các ứng dụng có ít components nhưng khi ứng dụng trở lên lớn hơn thì việc quản lý states được chia sẻ qua các components sẽ biến thành các công việc lặt nhặt.

Trong một app nơi data được chia sẻ thông qua các components, rất dễ nhầm lẫn để chúng ta có thể thực sự biết nơi mà một state đang live. Một sự lý tưởng là data trong một component nên live trong chỉ một component. Vì vậy việc share data thông qua các components anh em sẽ trở nên khó khăn hơn.

Ví dụ, trong react để share data thông qua các components anh em, một state phải live trong component cha. Một method để update chính state này sẽ được cung cấp bởi chính component cha này và pass như props đến các components con.

Đây là một ví dụ:

class App extends React.Component {

constructor(props) {

super(props);

this.state = { userStatus: "NOT LOGGED IN"}

this.setStatus = this.setStatus.bind(this);

}

setStatus(username, password) {

const newUsers = users;

newUsers.map(user => {

if (user.username == username && user.password === password) {

this.setState({

userStatus: "LOGGED IN"

})

}

});

}

render() {

return (

<div>

<Status status={this.state.userStatus} />

<Login handleSubmit={this.setStatus} />

</div>

);

}

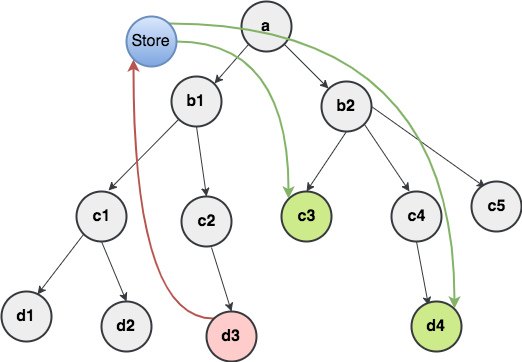
});

Giờ chúng ta hãy tưởng tượng rằng nếu một state phải được chia sẻ giữa các component cách khá xa nhau trong một tree components và state này phải được passed từ một component đến một component khác cho đến khi nó đến được nơi mà nó được gọi.

Cơ bản là, state mà chúng ta đang nói đến phải được nhấc lên một component cha gần nhất và tiếp nữa cho đến khi nó đến được cái component tổ tiên chứa tất cả các components nó cần cái state này sau đó pass cái state này xuống @@. Điều này sẽ khiến state trở nên khó hơn trong việc maintain và less predictable.

Điều này khiến cho bộ phận quản lý state trong app trở lên bừa bộn cũng như app trở lên vô cùng phức tạp. Đó là lý do tại sao chúng ta cần một state management tool như Redux.

* + 1. Hiểu cách redux làm việc



Cái cách mà Redux hoạt động là khá đơn giản. Nó có 1 store lưu trữ toàn bộ state của app. Mỗi component có thể access trực tiếp đến state được lưu trữ thay vì phải send drop down props từ component này đến component khác.

Có 3 thành phần của Redux: Actions, Store, Reducers.

**Actions**

Actions đơn giản là các events. Chúng là cách mà chúng ta send data từ app đến Redux store. Những data này có thể là từ sự tương tác của user vs app, API calls hoặc cũng có thể là từ form submission.

Actions được gửi bằng cách sử dụng store.dispatch() method, chúng phải có một type property để biểu lộ loại action để thực hiện. Chúng cũng phải có một payload chứa thông tin. Actions được tạo thông qua một action creator. Ví dụ:

const setLoginStatus = (name, password) => {

return {

type: "LOGIN",

payload: {

username: "foo",

password: "bar"

}

}

}

**Reducers**

Reducers là các function nguyên thủy chúng lấy state hiện tại của app, thực hiện một action và trả về một state mới. Những states này được lưu như những objects và chúng định rõ cách state của một ứng dụng thay đổi trong việc phản hồi một action được gửi đến store.

Đây là một ví dụ về cách mà Reducers hoạt động trong Redux:

const LoginComponent = (state = initialState, action) => {

switch (action.type) {

case "LOGIN":

return state.map(user => {

if (user.username !== action.username) {

return user;

}

if (user.password == action.password) {

return {

...user,

login\_status: "LOGGED IN"

}

}

});

default:

return state;

}

};

**Store**

Store lưu trạng thái ứng dụng và nó là duy nhất trong bất kỳ một ứng dụng Redux nào. Bạn có thể access các state được lưu, update state, và đăng ký or hủy đăng ký các listeners thông qua helper methods.

Tạo một store cho một login app:

const store = createStore(LoginComponent);

Các actions thực hiện trên một state luôn luôn trả về một state mới. Vì vậy, state này là đơn giản và dễ đoán. Bây giờ, chúng ta đã biết hơn một chúng về Redux, hãy trở lại với ví dụ Login component và xem cách cách mà Redux có thể giúp chúng ta được gì.

class App extends React.Component {

render() {

return (

<div>

<Status user={this.props.user.name}/>

<Login login={this.props.setLoginStatus}/>

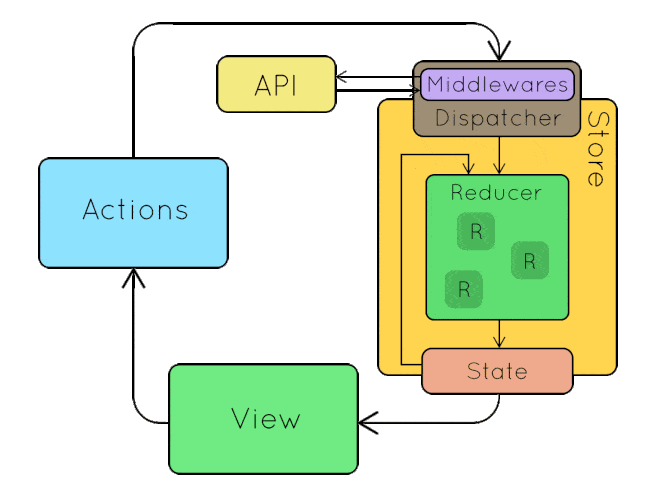
</div>

)

}

}

* + 1. Nguyên lý vận hành



1. Xây dựng dự án
   1. Backend
      1. Thiết kế cơ sở dữ liệu

Hệ thống xây dựng dựa trên cơ sở dữ liệu mysql gồm các bảng sau:

* **Bảng lớp**: id, mã lớp, tên lớp

CREATE TABLE `class` (

`id` int NOT NULL AUTO\_INCREMENT,

`class\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`class\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `class\_code` (`class\_code`)

)

* **Bảng sinh viên:** id, mã sinh viên, họ tên, ngày sinh, giới tính, mã lớp

CREATE TABLE `student` (

`id` int NOT NULL AUTO\_INCREMENT,

`student\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`student\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`date\_of\_birth` date DEFAULT NULL,

`gender` varchar(10) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`class\_id` int DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `student\_code` (`student\_code`),

KEY `class\_id` (`class\_id`),

CONSTRAINT `student\_ibfk\_1` FOREIGN KEY (`class\_id`) REFERENCES `class` (`id`)

)

* **Bảng giảng viên**: id, mã giảng viên, họ tên, ngày sinh, giới tính

CREATE TABLE `teacher` (

`id` int NOT NULL AUTO\_INCREMENT,

`teacher\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`teacher\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`date\_of\_birth` date DEFAULT NULL,

`gender` varchar(10) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `teacher\_code` (`teacher\_code`)

)

* **Bảng học kỳ:** id, mã học kỳ, tên học kỳ

CREATE TABLE `semester` (

`id` int NOT NULL AUTO\_INCREMENT,

`semester\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`semester\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `semester\_code` (`semester\_code`)

)

* **Bảng năm học**: id, tên năm học, mã năm học

CREATE TABLE `year` (

`id` int NOT NULL AUTO\_INCREMENT,

`year\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`year\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `year\_code` (`year\_code`)

)

* **Bảng môn học**: id, tên môn học, mã môn học

CREATE TABLE `subject` (

`id` int NOT NULL AUTO\_INCREMENT,

`subject\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`subject\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `subject\_code` (`subject\_code`)

)

* Mỗi môn học sẽ có nhiều lớp học phần khác nhau, ta có **bảng lớp học phần:** id, tên lớp học phần, mã lớp học phần, id môn học, id năm học, id học kỳ

CREATE TABLE `section\_class` (

`id` int NOT NULL AUTO\_INCREMENT,

`section\_class\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`section\_class\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`id\_subject` int DEFAULT NULL,

`id\_year` int DEFAULT NULL,

`id\_semester` int DEFAULT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `section\_class\_code` (`section\_class\_code`),

KEY `id\_subject` (`id\_subject`),

KEY `id\_year` (`id\_year`),

KEY `id\_semester` (`id\_semester`),

CONSTRAINT `section\_class\_ibfk\_1` FOREIGN KEY (`id\_subject`) REFERENCES `subject` (`id`),

CONSTRAINT `section\_class\_ibfk\_2` FOREIGN KEY (`id\_year`) REFERENCES `year` (`id`),

CONSTRAINT `section\_class\_ibfk\_3` FOREIGN KEY (`id\_semester`) REFERENCES `semester` (`id`)

)

* Để xác định sinh viên nào đăng ký học những lớp học phần nào, ta có **bảng học**: id sinh viên, id lớp học phần

CREATE TABLE `study` (

`id\_student` int NOT NULL,

`id\_section\_class` int NOT NULL,

PRIMARY KEY (`id\_student`,`id\_section\_class`),

KEY `id\_section\_class` (`id\_section\_class`),

CONSTRAINT `study\_ibfk\_1` FOREIGN KEY (`id\_student`) REFERENCES `student` (`id`),

CONSTRAINT `study\_ibfk\_2` FOREIGN KEY (`id\_section\_class`) REFERENCES `section\_class` (`id`)

)

* Để xác định giảng viên nào dạy những lớp học phần nào, ta có **bảng dạy**: id giảng viên, id lớp học phần

CREATE TABLE `teach` (

`id\_teacher` int NOT NULL,

`id\_section\_class` int NOT NULL,

PRIMARY KEY (`id\_teacher`,`id\_section\_class`),

KEY `id\_section\_class` (`id\_section\_class`),

CONSTRAINT `teach\_ibfk\_1` FOREIGN KEY (`id\_teacher`) REFERENCES `teacher` (`id`),

CONSTRAINT `teach\_ibfk\_2` FOREIGN KEY (`id\_section\_class`) REFERENCES `section\_class` (`id`)

)

* Xác định thời khóa biểu cho mỗi lớp học phần qua **bảng thời khóa biểu**: id, id lớp học phần, thời gian bắt đầu buổi học, thời gian kết thúc buổi học

CREATE TABLE `schedule` (

`id` int NOT NULL AUTO\_INCREMENT,

`id\_section\_class` int DEFAULT NULL,

`start\_time` datetime DEFAULT NULL,

`end\_time` datetime DEFAULT NULL,

PRIMARY KEY (`id`),

KEY `id\_section\_class` (`id\_section\_class`),

CONSTRAINT `schedule\_ibfk\_1` FOREIGN KEY (`id\_section\_class`) REFERENCES `section\_class` (`id`)

)

* Để xác định sinh viên nào đã điểm danh tại thời điểm nào, của lớp học phần nào, ta có **bảng điểm danh**: id thời khóa biểu, id sinh viên, id giảng viên, thời điểm điểm danh, thiết bị dùng để điểm danh

CREATE TABLE `attendance` (

`id\_schedule` int NOT NULL,

`id\_student` int NOT NULL,

`id\_teacher` int DEFAULT NULL,

`timestamp` datetime DEFAULT CURRENT\_TIMESTAMP,

`device` varchar(255) COLLATE utf8\_unicode\_ci DEFAULT NULL,

PRIMARY KEY (`id\_schedule`,`id\_student`),

KEY `id\_student` (`id\_student`),

KEY `id\_teacher` (`id\_teacher`),

CONSTRAINT `attendance\_ibfk\_1` FOREIGN KEY (`id\_schedule`) REFERENCES `schedule` (`id`),

CONSTRAINT `attendance\_ibfk\_2` FOREIGN KEY (`id\_student`) REFERENCES `student` (`id`),

CONSTRAINT `attendance\_ibfk\_3` FOREIGN KEY (`id\_teacher`) REFERENCES `teacher` (`id`)

)

* Mỗi người sẽ có 1 tài khoản đăng nhập hệ thống, ta có **bảng người dùng**: id, tên tài khoản, mật khẩu, id quyền

CREATE TABLE `user` (

`id` int NOT NULL AUTO\_INCREMENT,

`username` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci NOT NULL,

`password` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci DEFAULT NULL,

`role` int NOT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `username` (`username`),

KEY `user\_ibfk\_1` (`role`),

CONSTRAINT `user\_ibfk\_1` FOREIGN KEY (`role`) REFERENCES `role` (`id`)

)

* Để xác định người dùng có những quyền gì, ta có **bảng quyền**: id, tên quyền, mã quyền

CREATE TABLE `role` (

`id` int NOT NULL AUTO\_INCREMENT,

`role\_name` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci NOT NULL,

`role\_code` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci NOT NULL,

PRIMARY KEY (`id`),

UNIQUE KEY `role\_code\_UNIQUE` (`role\_code`)

)

* Để quản lý phiên đăng nhập người dùng, ta có **bảng access\_token**:

CREATE TABLE `access\_token` (

`id` int NOT NULL AUTO\_INCREMENT,

`username` varchar(255) CHARACTER SET utf8 COLLATE utf8\_unicode\_ci NOT NULL,

`created\_date` datetime DEFAULT CURRENT\_TIMESTAMP,

`token` text CHARACTER SET utf8 COLLATE utf8\_unicode\_ci NOT NULL,

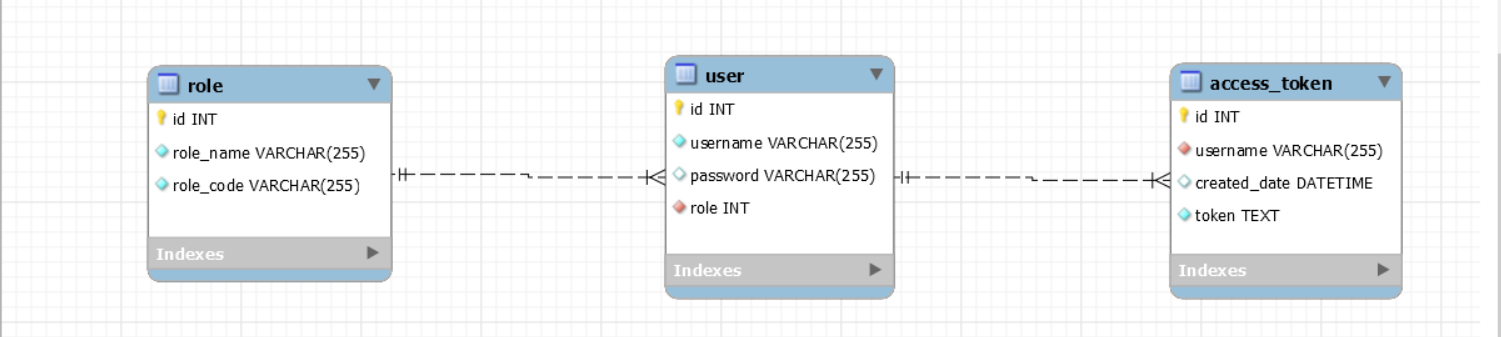
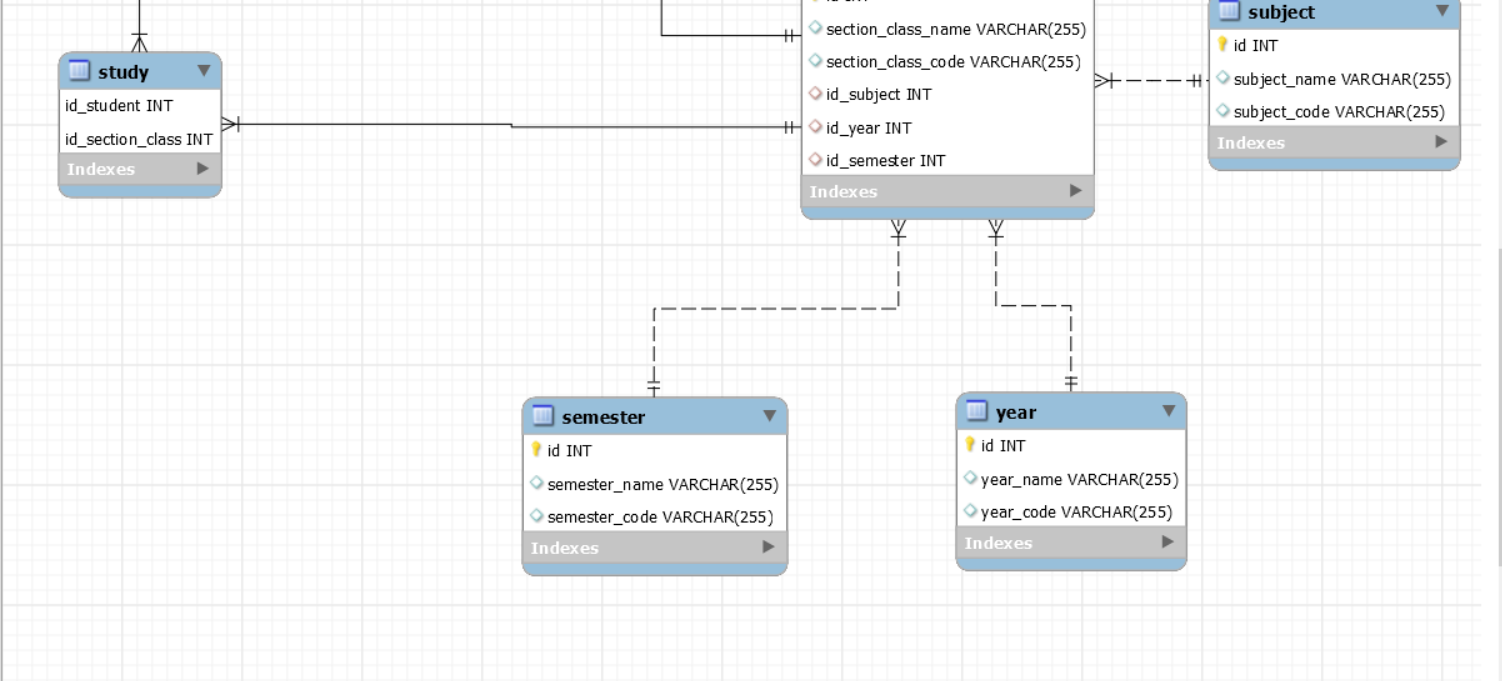
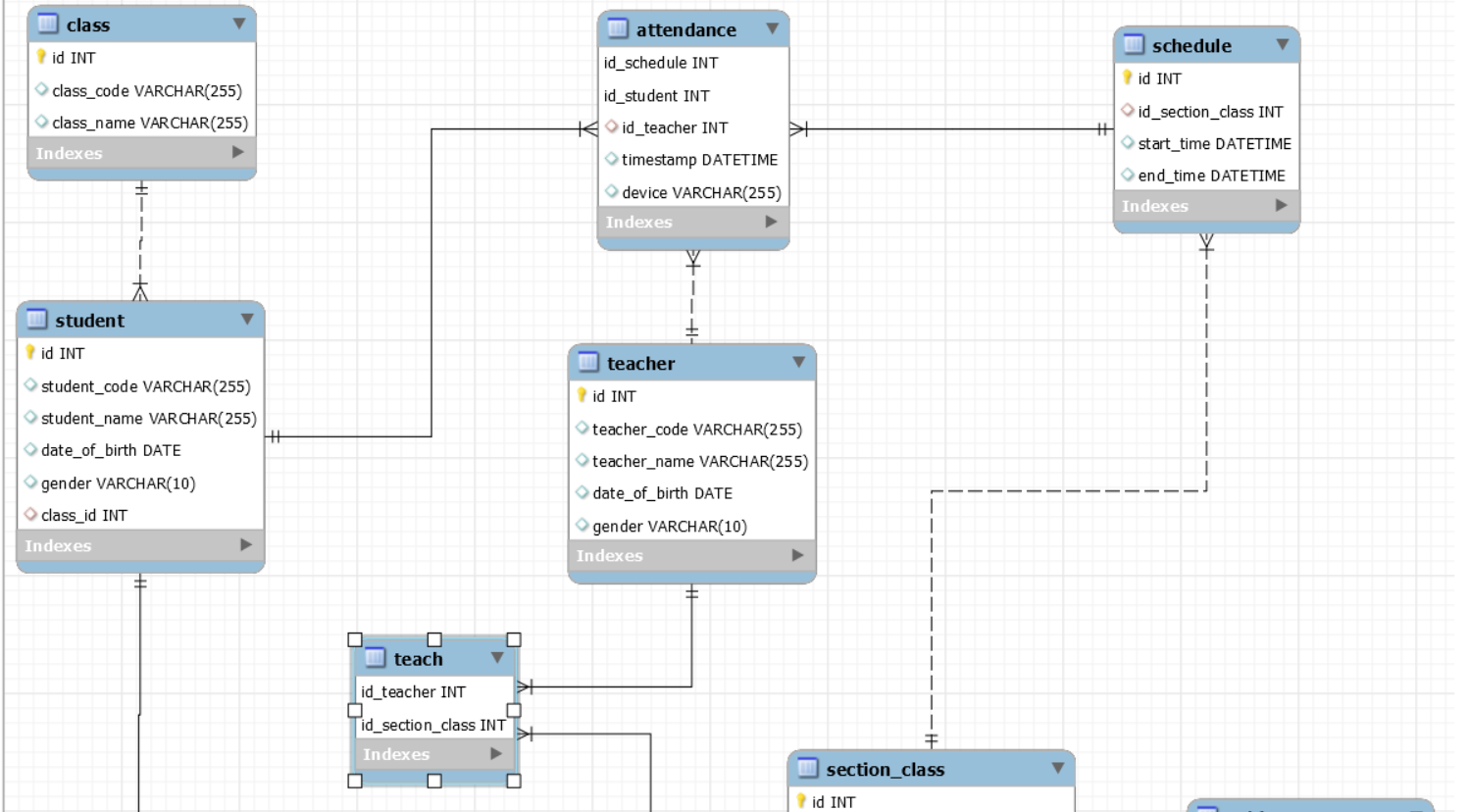
PRIMARY KEY (`id`),

KEY `f\_username\_idx` (`username`),

CONSTRAINT `f\_username` FOREIGN KEY (`username`) REFERENCES `user` (`username`)

)

* + 1. Sơ đồ diagram



### Công nghệ sử dụng

Ngôn ngữ lập trình: javascript

Nền tảng: nodejs

Framework: express

* + 1. Các vấn đề cần chú ý
       1. Cross-origin resource sharing

**CORS** là một cơ chế cho phép nhiều tài nguyên khác nhau (fonts, Javascript, v.v…) của một trang web có thể được truy vấn từ domain khác với domain của trang đó. **CORS** là viết tắt của từ Cross-origin resource sharing.

**no ‘access-control-allow-origin’ header is present on the requested resource**

Đây chính là lỗi CORS policy mà bất cứ lập trình viên nào cũng sẽ gặp phải. Khi call API tới server mà không có header Access-Control-Allow-Origin hoặc giá trị của nó không hợp lệ thì sẽ phát sinh lỗi này và không lấy được dữ liệu từ API. Cách khắc phục lỗi trên là phải config enable **CORS** lên để phía client có thể gọi được dữ liệu.

**Tại sao chúng ta cần CORS**

CORS được sinh ra là vì [same-origin policy](https://www.w3.org/Security/wiki/Same_Origin_Policy), là một chính sách liên quan đến bảo mật được cài đặt vào toàn bộ các trình duyệt hiện nay. Chính sách này ngăn chặn việc truy cập tài nguyên của các domain khác một cách vô tội vạ.

Ta có ví dụ một kịch bản như sau:

* Bạn truy cập một trang web có mã độc. Trang web đó sử dụng Javascript để truy cập tin nhắn Facebook của bạn ở địa chỉ https://facebook.com/messages.
* Nếu bạn đã đăng nhập Facebook từ trước rồi. Nếu không có same-origin policy, trang web độc hại kia có thể thoải mái lấy dữ liệu của bạn và bất cứ điều gì chúng muốn.

Same-origin policy chính là để ngăn chặn những kịch bản như trên để bảo vệ người dùng, giúp an toàn hơn khi lướt web.

$.get('https://facebook.com/messages')

Access to XMLHttpRequest at 'https://facebook.com/messages' from

origin 'xxx' has been blocked by CORS policy: No

'Access-Control-Allow-Origin' header is present on the requested

resource.

Truy cập URL trên từ bất kỳ domain nào ngoài facebook.com bạn cũng sẽ nhận được lỗi như vậy. Đó chính là nhờ same-origin policy.

Thế nhưng trong thế giới web, lập trình viên thường xuyên phải thực hiện truy vấn đến các domain khác, đặc biệt là khi làm việc với các API.

Đó là lúc chúng ta cần đến **CORS**. **CORS** sử dụng các HTTP header để “thông báo” cho trình duyệt rằng, một ứng dụng web chạy ở origin này (thường là domain này) có thể truy cập được các tài nguyên ở origin khác (domain khác).

Một ứng dụng web sẽ thực hiện truy vấn HTTP cross-origin nếu nó yêu cầu đến các tài nguyên ở origin khác với origin nó đang chạy (khác giao thức, domain, port). Sự khác biệt về giao thức ở đây là khác biệt kiểu như HTTP với FTP chứ không phải HTTP và HTTPS (dù nhiều trình duyệt không cho phép trộn lẫn các tài nguyên truy cập bằng HTTP và HTTPS nhưng đó là vấn đề khác, không liên quan đến CORS).

3.1.4.2 Lưu mật khẩu người dùng

Thông thường, khi tạo tài khoản cho người dùng đăng nhập vào hệ thống, ta sẽ lưu lại thông tin username, password vào database. Khi người dùng đăng nhập, ta kiểm tra username, password mà người dùng vừa nhập vào nếu trùng với thông tin lưu trong database thì xác nhận user đăng nhập thành công.

Nhưng, nếu lưu mật khẩu lại như vậy, sẽ có rủi ro khi database bị người ngoài xem được, toàn bộ mật khẩu của người dùng sẽ bị lộ. Để ngăn chặn điều đó, ta sẽ không lưu trực tiếp mật khẩu, mà chỉ lưu 1 đoạn mã hash của mật khẩu gốc lại. Khi người dùng đăng nhập, ta chỉ cần hash mật khẩu người dùng vừa nhập và kiểm tra với mã hash đã lưu trong database là được. Khả năng có 2 mật khẩu khác nhau mà có chung mã hash là rất thấp, gần như không thể.

Giả sử 1 hacker có được thông tin toàn bộ user trong database, hacker cũng chỉ có thể biết được mã hash của mật khẩu người dùng. Để biết chính xác mật khẩu là gì, hacker bắt buộc phải thực hiện kiểm tra thử từng mật khẩu và hash để kiểm tra (**bruteforce), điều này sẽ rất mất thời gian**

**Tuy nhiên, vẫn còn một vấn đề. Nếu như mật khẩu người dùng bị hacker bruteforce ra được, giả sử là “123456”, hacker hoàn toàn có thể tìm tất cả các user còn lại trong database xem có mã hash nào trùng với mã hash của “123456” không => nếu bruteforce được mật khẩu sẽ biết được mật khẩu của tất cả các user có chung mật khẩu đó.**

**Để ngăn chặn điều này, ta sẽ không chỉ hash mật khẩu người dùng, mà sẽ hash cùng với username luôn, tức là nếu username='Tintk'’và password='’23456’ ta sẽ lưu trong database mã hash('Tintk123456'). Điều này sẽ đảm bảo nếu 2 user có chung mật khẩu thì mã hash lưu trong database cũng sẽ khác nhau => bruteforce mỗi lần cũng chỉ có thể tìm được 1 user**

#### **Xác thực người dùng - jwt**

Như chúng ta đã biết trong **API** thì Client giao tiếp và gửi request (yêu cầu) đến Server thông qua URL trên giao thức HTTP , ví dụ ta có 2 yêu cầu:

// [GET]

@RequestMapping(value = "/users/{userid}", method = RequestMethod.GET)

public User findUserByUserId(@PathVariable("userid") long userId) {

return userRepository.findOne(userId);

}

// [DELETE]

@RequestMapping(value = "/users/{userid}", method = RequestMethod.DELETE)

public String deleteUserByUserId(@PathVariable("userid") long userId) {

userRepository.delete(userId);

return "Successful";

}

* @RequestMapping : dùng để cấu hình URL của request
* method : ở đây mình sử dụng 2 phương thức đó là GET và DELETE
* @PathVariable: đây là tham số truyền vào đi kèm với request , ở đây mình truyền trực tiếp trên đường dẫn nên mình sử dụng PathVariable

Ví dụ ta yêu cầu Server lấy **User** có **Id** là **01** như sau [GET] localhost:8080/users/01 hoặc xóa [DELETE] localhost:8080/users/01. Ở đây nếu chúng ta không sử dụng bất kì phương thức nào để bảo mật API thì tất cả các User khác điều có thể gọi tới các Request này để lấy thông tin hoặc xoá User 01 và Server sẽ thực hiện yêu cầu mà không cần biết yêu cầu này có phải là của User 01 hay không . Điều này rất nguy hiểm , các hacker có thể xóa hết dữ liệu hoặc đánh cắp thông tin người dùng bằng cách truy cập vào các URL này , cho nên ta cần một phương pháp nào đó để Server xác định được yêu cầu đó là của User01 thì Server mới thực hiện , vì vậy ta sẽ sử dụng **JWT.**

**JWT là gì?**

JSON Web Mã (JWT) là một chuẩn mở (RFC 7519) định nghĩa một cách nhỏ gọn và khép kín để truyền một cách an toàn thông tin giữa các bên dưới dạng đối tượng JSON. Thông tin này có thể được xác minh và đáng tin cậy vì nó có chứa chữ ký số. JWTs có thể được ký bằng một thuật toán bí mật (với thuật toán HMAC) hoặc một public / private key sử dụng mã hoá RSA.

Một ví dụ về JWT Token:

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJleHAiOjEzODY4OTkxMzEsImlzcyI6ImppcmE6MTU0ODk1OTUiLCJxc2giOiI4MDYzZmY0Y2ExZTQxZGY3YmM5MGM4YWI2ZDBmNjIwN2Q0OTFjZjZkYWQ3YzY2ZWE3OTdiNDYxNGI3MTkyMmU5IiwiaWF0IjoxMzg2ODk4OTUxfQ.uKqU9dTB6gKwG6jQCuXYAiMNdfNRw98Hw\_IWuA5MaMo

Thoạt trông phức tạp là thế nhưng nếu hiểu, cấu trúc của một JWT chỉ đơn giản như sau:

<base64-encoded header>.<base64-encoded payload>.<base64-encoded signature>

Nói một cách khác, JWT là sự kết hợp (bởi dấu .) một Object Header dưới định dạng JSON được encode base64, một payload object dưới định dạng JSOn được encode base64 và một Signature cho URI cũng được mã hóa base64 nốt.

**Giải thích thêm về 3 thành phần của JWT**

**Header**

Header bao gồm hai phần chính: loại token (mặc định là JWT - Thông tin này cho biết đây là một Token JWT) và thuật toán đã dùng để mã hóa (HMAC SHA256 - HS256 hoặc RSA).

{

"alg": "HS256",

"typ": "JWT"

}

**Payload**

Payload chứa các claims. Claims là một các biểu thức về một thực thể (chẳng hạn user) và một số metadata phụ trợ. Có 3 loại claims thường gặp trong Payload: reserved, public và private claims.

Reserved claims: Đây là một số metadata được định nghĩa trước, trong đó một số metadata là bắt buộc, số còn lại nên tuân theo để JWT hợp lệ và đầy đủ thông tin: iss (issuer), iat (issued-at time) exp (expiration time), sub (subject), aud (audience), jti (Unique Identifier cho JWT, Can be used to prevent the JWT from being replayed. This is helpful for a one time use token.) ... Ví dụ:

{

"iss": "jira:1314039",

"iat": 1300819370,

"exp": 1300819380,

"qsh": "8063ff4ca1e41df7bc90c8ab6d0f6207d491cf6dad7c66ea797b4614b71922e9",

"sub": "batman",

"context": {

"user": {

"userKey": "batman",

"username": "bwayne",

"displayName": "Bruce Wayne"

}

}

}

{

"iss": "scotch.io",

"exp": 1300819380,

"name": "Chris Sevilleja",

"admin": true

}

**Public Claims** - Claims được cộng đồng công nhận và sử dụng rộng rãi.

**Private Claims** - Claims tự định nghĩa (không được trùng với Reserved Claims và Public Claims), được tạo ra để chia sẻ thông tin giữa 2 parties đã thỏa thuận và thống nhất trước đó.

**Signature**

Chữ ký Signature trong JWT là một chuỗi được mã hóa bởi header, payload cùng với một chuỗi bí mật theo nguyên tắc sau:

HMACSHA256(

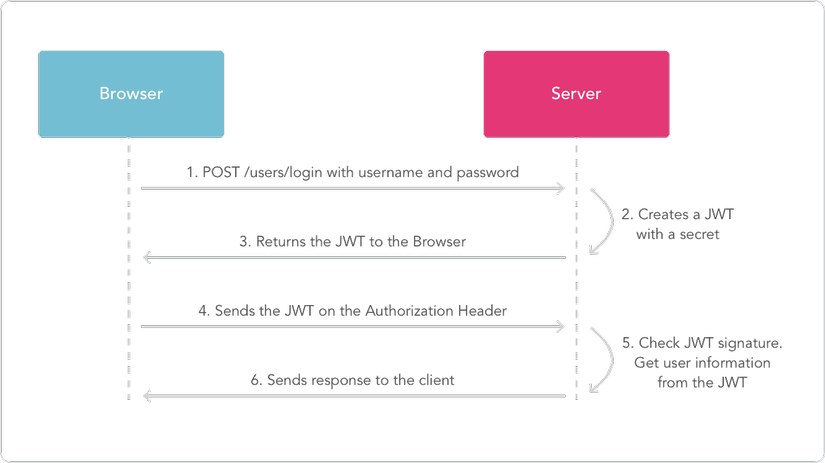
base64UrlEncode(header) + "." +

base64UrlEncode(payload),

secret)

Do bản thân Signature đã bao gồm cả header và payload nên Signature có thể dùng để kiểm tra tính toàn vẹn của dữ liệu khi truyền tải.

**Cách hoạt động của JWT**



Giải thích theo sơ đồ trên thì :

* **Bước 1 :** Người dùng yêu cầu đăng nhập với Username , Password
* **Bước 2 + 3 :**Server nhận được yêu cầu và kiểm tra Username , Password nếu đúng sẽ gửi cho người dùng một chuỗi JWT
* **Bước 4 :** Người dùng sẽ dùng JWT này kèm theo các yêu cầu kế tiếp
* **Bước 5 + 6 :** Server sẽ nhận yêu cầu và kiểm tra chuổi JWT , nếu chuỗi hợp lệ thì sẽ thực hiện yêu cầu

Hơi rối một chút đúng không , mình sẽ giải thích theo ví dụ ở đầu bài thì ta sẽ có :

* **Bước 1 :** User01 sẽ yêu cầu đăng nhập với Username và Password của mình
* **Bước 2 + 3 :** Server nhận được yêu cầu của User01 và kiểm tra Username , Password nếu đúng sẽ gửi về User01 một chuỗi JWT
* **Bước 4 :** Lúc này User01 yêu cầu [GET] localhost:8080/users/01 kèm theo chuỗi JWT vừa nhận được
* **Bước 5 + 6 :** Server nhận yêu cầu và kiểm tra xem nếu JWT chính xác là chuổi JWT lúc đầu Server đã gửi cho User01 thì sẽ trả về các thông tin của User01

3.1.4.3 SQL injection

**SQL Injection** là một kỹ thuật lợi dụng những lỗ hổng về câu truy vấn của các ứng dụng. Được thực hiện bằng cách chèn thêm một đoạn [SQL](https://topdev.vn/blog/sql-la-gi/) để làm sai lệnh đi câu truy vấn ban đầu, từ đó có thể khai thác dữ liệu từ database. **SQL injection** có thể cho phép những kẻ tấn công thực hiện các thao tác như một người quản trị web, trên cơ sở dữ liệu của ứng dụng.

Ví dụ, trong form đăng nhập, người dùng nhập dữ liệu, trong trường tìm kiếm người dùng nhập văn bản tìm kiếm, trong biểu mẫu lưu dữ liệu, người dùng nhập dữ liệu cần lưu. Tất cả các dữ liệu được chỉ định này đều đi vào cơ sở dữ liệu.

Thay vì nhập dữ liệu đúng, kẻ tấn công lợi dụng lỗ hổng để insert và thực thi các câu lệnh SQL bất hợp pháp để lấy dữ liệu của người dùng… SQL Injection được thực hiện với ngôn ngữ lập trình SQL. SQL (Structured Query Language) được sử dụng để quản lý dữ liệu được lưu trữ trong toàn bộ cơ sở dữ liệu.

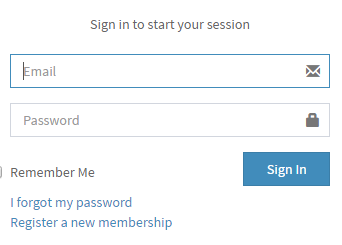
**Sự nguy hiểm của SQL Injection**

* Hack tài khoản cá nhân.
* Ăn cắp hoặc sao chép dữ liệu của trang web hoặc hệ thống.
* Thay đổi dữ liệu nhạy cảm của hệ thống.
* Xóa dữ liệu nhạy cảm và quan trọng của hệ thống.
* Người dùng có thể đăng nhập vào ứng dụng với tư cách người dùng khác, ngay cả với tư cách quản trị viên.
* Người dùng có thể xem thông tin cá nhân thuộc về những người dùng khác, ví dụ chi tiết hồ sơ của người dùng khác, chi tiết giao dịch của họ,…
* Người dùng có thể sửa đổi cấu trúc của cơ sở dữ liệu, thậm chí xóa các bảng trong cơ sở dữ liệu ứng dụng.
* Người dùng có thể kiểm soát máy chủ cơ sở dữ liệu và thực thi lệnh theo ý muốn.

**Ví dụ tấn công**

Việc kiểm tra lỗ hổng này có thể được thực hiện rất dễ dàng. Đôi khi ta chỉ cần nhập ký hiệu ' hoặc " vào các trường được kiểm tra. Nếu nó trả về bất kỳ thông báo bất ngờ hoặc bất thường, thì ta có thể chắc chắn rằng SQL Injection khả thi cho trường đó.

Ví dụ: một Form đăng nhập như sau



Và đoạn code server xử lý của bạn:

if(isset($\_POST['username']) && isset($\_POST['password'])){

$sql = "SELECT \* FROM tbl\_user WHERE username='". $\_POST['username'] . "' AND password = '" .$\_POST['password'] ."'";

}

Nếu như người dùng không nhập bình thường nữa mà chẳng hạn như họ có thêm một dấu nháy ' hoặc " vào thì dòng code của bạn sẽ bị lỗi ngay. Hoặc họ có thể sửa thành một câu truy vấn luôn luôn đúng như sau.

SELECT \* FROM tbl\_user WHERE username = '' OR '1' = '1' and password = '' OR '1' = '1'

Hoặc chèn thêm một câu lệnh truy vấn phía sau:

SELECT \* FROM tbl\_user WHERE username = 'admin' and password = 'admin'; Drop table users;

**Cách giảm thiểu và phòng ngừa SQL Injection**

Luôn kiểm tra kỹ các trường nhập dữ liệu và các bạn cần ràng buộc thật kỹ dữ liệu người dùng nhập vào.

### Luồng chạy chương trình

#### 3.1.5.1 App.js

Server khi bắt đầu khởi chạy sẽ chạy vào file app.js đầu tiên

Tại đây, ta cài đặt 1 số cấu hình cần thiết của express

'use strict';

import express from 'express';

import cors from 'cors';

import morgan from 'morgan';

import body\_parser from 'body-parser';

import config from './config';

import routes from './routes';

import {init as mysql\_init} from './lib/mysql\_connector';

import response from './lib/middlewares/res';

import {error404} from './lib/middlewares/error\_handler';

global['config'] = config;

const app = express();

mysql\_init(config.mysql);

app.disable('x-powered-by');

app.set('trust proxy', true);

app.use(cors({origin: true, credentials: true}));

app.use(morgan(':remote-addr :remote-user :user-agent :method :url HTTP/:http-version :status :res[content-length] - :response-time ms'));

app.use(response());

app.use(body\_parser.json({limit: '50mb'}));

app.use(body\_parser.urlencoded({extended: false, limit: '50mb'}));

app.use('/api/', routes);

app.use(error404());

app.listen(config.http.port, () => {

    console.log(`\nStart server at: ${new Date()}

                HTTP server is listening at: ${config.http.host}:${config.http.port}

                Mysql: ${config.mysql.host}, Database: ${config.mysql.database}

    `);

});

Để giải quyết vấn đề Cross-origin resource ở trên, npm cung cấp thư viện cors, ta gọi luôn tại app.js

import cors from 'cors';

app.use(cors({origin: true, credentials: true}));

Khi người khác gọi api, server sẽ trả response kèm thêm tham số x-power-by là nodejs, tức server được viết bằng nodejs. Để tránh rủi ro, ta không nên cho người dùng biết server được viết bằng gì, express cung cấp tùy chọn sau để tắt x-power-by đi:

app.disable('x-powered-by');

Để dễ dàng trong việc quản lý các response, ta định nghĩa 1 cấu trúc response nhất định, để thống nhất giữa bên backend và frontend khi gọi qua lại

import response from './lib/middlewares/res';

app.use(response());

Ở đây, response ta định nghĩa là 1 object: {status, message, data}

Ta cần response thống nhất cho mọi api, để bên phía frontend khi gọi, sẽ không phải check quá nhều trường hợp khi mà mỗi lần gọi api khác nhau sẽ có response trả về cấu trúc khác nhau => dễ quản lý và mở rộng dự án

Không nên để người dùng truyền 1 lượng lớn dữ liệu lên server 1 lúc, ta giới hạn dung lương data truyền lên là 50mb

app.use(body\_parser.json({limit: '50mb'}));

app.use(body\_parser.urlencoded({extended: false, limit: '50mb'}));

sau khi cài đặt xong các cấu hình cần thiết, mọi api khi gọi đến server sẽ được chạy ở bên router.js

import routes from './routes';

app.use('/api/', routes);

#### 3.1.5.2 Routes

Nhiệm vụ của routes là quyết định xem api nào, sẽ được chạy những hàm nào, ở đâu

Thứ tự chạy sẽ là: khi người dùng gọi 1 api, router sẽ quyết định api được gọi đến hàm nào (được viết trong controller), tại controller dữ liệu đầu vào sẽ được xử lý, kiểm tra tính hợp lệ, và gọi đến business, business sẽ thực hiện các thao tác với cơ sở dữ liệu, xử lý thông tin và trả về cho controller, sau đó controler trả response về cho người dùng

Có 4 api cơ bản là GET, POST, PUT, DELETE ứng với các thao tác lấy dữ liệu, thêm, sửa, xóa (cái này do mình tự quy định để dễ quản lý và mở rộng dự án, còn thực ra làm là do mình code)

Ở đây ta có thêm 1 khái niệm **middleware** trong express

ExpressJS khi hoạt động, về cơ bản sẽ là một loạt các hàm **Middleware** được thực hiện liên tiếp nhau. Sau khi đã thiết lập, các request từ phía người dùng khi gửi lên ExpressJS sẽ thực hiện lần lượt qua các hàm Middleware cho đến khi trả về response cho người dùng. Các hàm này sẽ được quyền truy cập đến các đối tượng đại diện cho Request - **req**, Response - **res**, hàm Middleware tiếp theo - **next**, và đối tượng lỗi - **err** nếu cần thiết.

Một hàm Middleware sau khi hoạt động xong, nếu chưa phải là cuối cùng trong chuỗi các hàm cần thực hiện, sẽ cần gọi lệnh next() để chuyển sang hàm tiếp theo, bằng không xử lý sẽ bị treo tại hàm đó.

Các chức năng mà middleware có thể thực hiện trong ExpressJS sẽ bao gồm :

* Thực hiện bất cứ đoạn code nào
* Thay đổi các đối tượng request và response
* Kết thúc một quá trình request-response
* Gọi hàm middleware tiếp theo trong stack

Hiểu 1 cách đơn giản thì bắt đầu từ lúc người dùng gọi 1 api, server sẽ lần lượt chạy qua nhiều hàm khác nhau để xử lý, thay đổi request, response, mỗi hàm đó được gọi là 1 middleware. Cors nói ở trên cũng là 1 dạng middleware.

Ví dụ, khi gọi api để tạo account mới, đầu tiên phải qua middlaware cors để loại bỏ lỗi cors => middlaware kiểm token của người gọi có hợp lệ không… => middlaware kiểm tra quyền của người gọi api, phải là admin mới được cấp quyền => middleware kiểm tra đầu vào có hợp lệ không => tạo account

Bất kì 1 bước nào của middleware không hợp lệ, lỗi thì thao tác gọi api sẽ không thành công.

Việc chia ra các middleware thực hiện từng nhiệm vụ riêng biệt giúp cho việc quản lý, mở rộng dự án dễ dàng hơn.

Trong dự án này, mọi api sẽ được đi qua middleware kiểm tra token, được kiểm tra bằng thư viện jsonwebtoken của npm, sau đó đi qua middleware kiểm tra quyền truy cập vào route.

#### 3.1.5.3 Controller

Nhiệm vụ chính của controller là kiểm tra dữ liệu đầu vào được gửi lên có hợp lệ hay không, xử lý dữ liệu đầu vào sau đó gửi đến business để thực hiện nhiệm vụ chính.

Để dễ dàng trong việc kiểm tra đầu vào, npm cung cấp thư viện Joi

….