- NodeJS là JS runtime, ta có thể dùng NodeJS để chạy JS bên ngoài browser.

- NodeJS sử dụng V8 và V8 là tên của JS engine xây dựng bởi GG - Biên dịch JS code thành machine code.

- NodeJS về cơ bản sử dụng code của V8 - codebase (V8 is written in C++) và thêm vào 1 số tính năng khác.

- Với NodeJS bạn không chỉ viết code chạy trên server mà bạn còn tạo ra được server từ nó

- Event loop sẽ tự động chạy bởi NodeJS khi chương trình chạy. Về cơ bản Event loop chỉ xử lý những callback kết thúc nhanh.

- Các operations tốn nhiều tgian xử lý sẽ được đưa đến **Worker Pool** - quản lý bởi NodeJS

- Worker Pool sẽ phụ trách cho những tiến trình thực thi tốn tgian. Có thể nói nó hoàn toàn tách ra khỏi JS code và chạy ở 1 thread khác.

- Những operations tốn nhiều tgian thực thi như file operation thì NodeJS sẽ bỏ qua giai đoạn đó ở 1 thời điểm nhất định và điều đó đồng nghĩa nếu có quá nhiều callbacks thì Event loop sẽ tiếp tục lặp và hoãn lại các callbacks này để thực thi chúng ở vòng lặp tiếp theo

- Khái niệm quan trọng của NodeJS là sẽ gọi Event loop (tiến trình loop do NodeJS quản lý) nó sẽ chạy liên tục miễn là vẫn còn các event listener.

- Toàn bộ tiến trình node về cơ bản sử dụng 1 thread ở trên máy tính

- Các fields của req cần nhớ : **url, method, headers.**

**CẢI THIỆN CÁCH LẬP TRÌNH VÀ DEBUGGING**

**41.**Trong module này sẽ học debug ứng dụng Nodejs, 1 số tips & tricks giúp viết ứng dụng NodeJS 1 cách hiệu quả (Fixing Errors, Developing Efficiently).

**42.** Học về thực thi project bằng câu lệnh npm: **npm init, npm start, npm run \*name\*** . Vì khi đưa project cho ng khác họ không cần phải tìm file script chạy chính (entry point) mà chỉ cần gõ câu lệnh:

**- npm install (cài những thứ cần thiết trong package.json)**

**- npm start**

**- npm run \*name\* (chạy câu lệnh mình tự config)**

start, init,.. là những key đặc biệt không dc tự ý thay đổi.

**43.** Cài 3th party bằng npm: **[npm install \*name 3th party\*]**

**--save** : cài như là production dependency. (cài vào proj khi lên server thật).

**--save-dev**: chỉ thêm để hỗ trợ cho việc development. (Cài để phục vụ trong lúc dev).

**-g** : Ko cài vào proj mà cài thẳng global vào máy tính và có thể sử dụng bất kỳ đâu.

**44. Global features**: Có sẵn để dùng ko cần cài hoặc import - process,...

**Core NodeJS module**: import vào khi cần dùng ko cần cài.

**3th party**: cài thông qua lệnh npm install và import vào thì mới dùng dc.

**45.** **Nodemon**: Nếu chỉ cài 3th local ở trong project thì khi gõ lệnh ở cmd sẽ ko được, vì cmd chỉ tìm ở global thôi.

Còn npm script sẽ tìm lệnh ở local.

**46.** Terminal chỉ thực thi các global package **KO** thực thi local dependencies.

**47.** Tìm và sửa lỗi code - các loại lỗi:

- Syntax Errors

- Runtime Errors

- Logical Errors

**48. Syntax Errors.**

**49. Runtime Errors.**

**50. Logical Errors.**

**51.** Sử dụng **Debugger**

**52.** Đặt breakpoint, Viết câu lệnh thực thi ở debug console mà ko ảnh hưởng tới code.

Config cho Debugger tự động restart ở **launch.json** khi thay đổi code.

**"restart": true,**

**"runtimeExecutable": "nodemon",**

**"console": "integratedTerminal"**

**53.** Tài liệu về Debugger

**54.** Có thể chỉnh sửa value của 1 biến ngay tại Debugger.

**55.** Review:

- npm

- 3th party

**LÀM VIỆC VỚI EXPRESS JS**

app.listen(port) ~~ const server = http.createServer(app);

               server.listen(port);

**app.listen(port) tương đương với 2 câu lệnh createServer**

**Express làm điều đó cho mình**

**app.listen(port) sẽ trả về node server**

57. Như đã thấy thì code server với 1 mình NodeJS thì phải viết **rất nhiều code** chỉ để xử lý những việc cơ bản (nhỏ) như lấy body từ request,… Cho nên ta phải nhờ đến **framework** **Express JS.**

58. **Express JS** giúp handle những việc ko cần thiết mà NodeJS phải code khá nhiều để xử lý.

Nhờ có Express JS nên ta chỉ cần tập trung vào xử lý phần logic ở server.

**Framework** bao gồm những **helper function, tools & rules** giúp bạn xây dựng application nhanh chóng, dễ dàng hơn.

59. Cài Express JS.

60. **app.use():** cho phép ta thêm mới 1 middleware function, khá linh động.

**function** truyền vào **use()** sẽ được thực thi với mọi request gọi lên.

**next** là 1 **function** sẽ được truyền vào bởi **Express**

và **next** phải dc thực thi để cho phép request đi tới middleware tiếp theo.

61. Cách **Middleware** hoạt động, **res.send()** trả về response với dữ liệu bất kỳ: html, json,…

62. Cách thức **ExpressJS** hoạt động ngầm ở dưới: **send()**,…

63. Xử lý các **routes** khác nhau. Request sẽ chạy từ trên xuống khớp với routes nào thì sẽ vào middleware đó.

“/”: match với mọi routes.

“/\*name\*”: match với mọi routes bắt đầu với (/\*name\*).

64. Parse request với body-parser, **req.body.**

65. Giới hạn phạm vi thực thi với POST request => **app.post(), app.get(),…**

66. Sử dụng **Express Router. Router** giống như là **1 express mini** dc gắn với ứng dụng express khác.

**router** ở đây cơ bản hoạt động giống như **app.**

Điều quan trọng là thực ra **router là middleware function.**

**get, post,..** sẽ match chính xác với route, khác với **use.**

67. Trang Error, res.status(404).send(‘’);

68. Lọc paths, **app.use(‘/admin’, adminRouter);**

Các routes bắt đầu với **/admin** sẽ vô file **adminRouter**

**/admin** là phần lọc để chia kênh xử lý.

69. response về html file cho client, chuẩn bị tiếp cận mô hình **MVC**.

**TEMPLATE ENGINE**

77. Những bài trước chỉ trả về client trang HTML tĩnh **(Ko làm trong dự án thực tế)**

Thực tế sẽ trả về dữ liệu **dynamic** kèm với **HTML động**

**MVC**

95. Giới thiệu bắt đầu với MVC.

96. **Models** phụ trách phần thao tác với dữ liệu, logic, tính toán,…

**Views** phụ trách phần giao diện, show ra cho client xem.

**Controller** đóng vai trò trung gian đưa dữ liệu từ **Models** hiển thị ra **Views**.

97,98. Thêm Controllers. Tách riêng ra từng Controller để quản lý riêng Product, User, Category,…

99. Thêm Models,

**SESSION & COOKIE**

Các **req** lên server là độc lập, mỗi lần **request** lên là **tạo một req mới** nên không lưu properties vào **req** dc.

Khắc phục điều này, ta sử dụng **Cookies (lưu ở browser)** bằng cách **setHeader** cho res

**res.setHeader(‘Set-Cookie’, ‘loggedIn=true; Secure; HttpOnly’);**

**Mặc định** browser sẽ gửi **request** lên server kèm theo **Cookies**.

**Sercure**: cookie chỉ dc set khi trang là **HTTPS**

**HttpOnly:** giúp **chặn** truy cập cookie thông qua **javascript** ở client, yếu tố bảo mật cao tránh tấn công từ những trang khác.

Thường sẽ không set trực tiếp Cookie mà sử dụng thư viện hỗ trợ.

**Session** để lưu dữ liệu ở **server** với từng user **riêng biệt**.

Dùng **Cookie** để lưu **hashed ID** của **Session** ở browser.

Tạo **Session Middleware** để session dc sử dụng với từng **incoming request**.

Cài 3th party để sử dụng Session : **npm install express-session**.

Cấu hình session:

app.use(session({

secret: 'ksBojapzPUL6xlasKKImx3RQQmMqBU4S',

resave: false,

saveUninitialized: false,

}))

**Express-session** tự động **set Cookie** ở browser và tự động **đọc Cookie value, Parse Cookie** cho mình.

**Session cookie** sẽ **hết hạn** khi đóng **browser**.

**Session** nhận biết từng kết nối riêng biệt trên server thông qua **hashID** dc lưu trong **Cookie** khi user **request**.

**Session** thường dùng để **authen user**, ngoài ra có thể store các data khác **thuộc về user** tùy ý mình.

Sử dụng **MongoDB** để lưu **session** với package: \***npm install connect-mongodb-session\*.**

**Session** sẽ **tự động tạo mới** khi **session data** bị thay đổi ,chỉ cần gọi **req.session** và chấm tới property muốn thêm :

*req*.session.user = *result*;

**Cookie** bị xóa khi tắt trình duyệt dc gọi là **Session Cookie**.

**Cookie** khi tắt trình duyệt mà ko bị xóa mà chỉ bị xóa khi hết hạn dc gọi là **Permanent Cookie.**

**npm install bcryptjs**  package dùng để mã hóa.

**CSRF Attack: giả mạo yêu cầu liên trang web (**cross-site request forgery), **session** bị lấy cắp để giả mạo user.

Phòng tránh bằng cách chặn ko cho **session** hoạt động trên các page giả mạo.

Ngăn chặn CSRF: **npm install csurf .** Hỗ trợ tạo **CSRF Token.**

Nhúng **Token** này vào view và với mỗi request sẽ kèm theo **Token** để server check xem có hợp lệ ko. **Token** mới sẽ dc tạo khi render trang mới.

Các **request** với method **KO PHẢI** get thì package **csurf** sẽ tìm **CSRF Token** trong **request body**.

**req.csrfToken()** hàm tạo **Token** cung cấp bởi package. Lấy ở trong **req**

**Express** cung cấp thuộc tính **locals** ở **res**. Cho phép ta set các biến local được truyền ra **view**,

Nó chỉ tồn tại ở views được render.

Thông báo **Flash**: **npm install connect-flash**

**req.flash([key, value])** hàm dc thêm vào req bởi package.

Thông báo **Flash** sẽ tồn tại trong **session** cho đến khi mình sử dụng. Sau khi dùng thì nó sẽ bị **xóa** khỏi **session**.

**SEND MAILS**

<https://sendgrid.com/> : hỗ trợ gửi mail free – 100 mail / ngày

Package hỗ trợ gửi mail kết hợp với sendgrid: **npm install nodemailer**

**npm install nodemailer-sendgrid-transport**

**crypto**: Tạo chuỗi random đã mã hóa tồn tại duy nhất để làm **token**.

**VALIDATION**

Package hỗ trợ validation: **npm install express-validator**

Package này chứa các sub-package để validate: **check**, **validationResult**,…

**Custom Validator** sẽ nhận giá trị trả về là **True, False, new Error, Promise**

Nếu trả về là **Promise** thì Validator sẽ **CHỜ** cho **Promise** trả về kết quả

* Nếu kết quả ko có gì sẽ dc xem là thành công.
* Nếu kết quả trả về **Reject** thì **Validator** sẽ nhận lại **1 Error Message**.

const error = **new** *Error*(*err*);

error.httpStatusCode = 500;

return rext(*err*);

Khi ta gọi **next(err)** và truyền vào **Error** thì **Express** sẽ biết đã có lỗi xảy ra và nó sẽ **BỎ QUA** tất cả các **middleware** khác và **ĐI THẲNG** vào hàm **middleware** xử lý **Error.**

**Throw error** sẽ ngưng mọi xử lý và được gọi thẳng đến **Error Handler** của Express

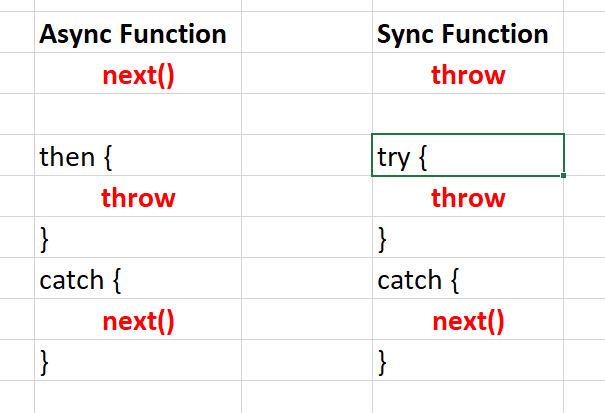
**Throw** ở mọi nơi trừ **Async function, catch()**

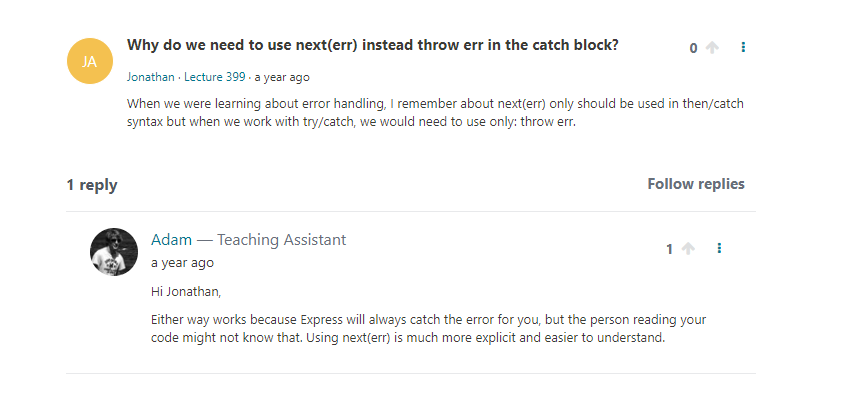
**Sync function** thì dùng **try...catch**.

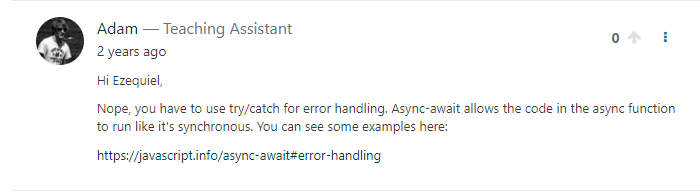
**Async function** thì dùng **Promise** (**then...catch**).

**Async function** khi xảy ra lỗi **bắt buộc** phải gọi **next(err).**

**Next(error)** sẽ được gọi đến **Error Handler** khi để trong **catch().**

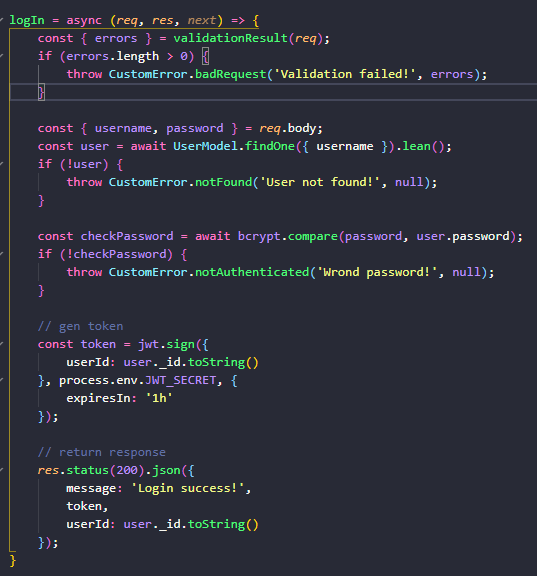






**Async/Await** cho phép code ở trong **Async function** hoạt động như **Sync function**.

**Express** sẽ handle mọi error ở **middleware** có 4 param **(err, req, res, next) => {}.**



**Trái:** code này sử dụng **async/await** thay thế cho **Promise** sẽ ngắn gọn hơn. Kèm theo đó là **try...catch** để bắt lỗi.

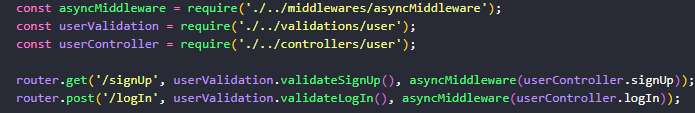
**Phải**: code này lược bỏ bớt **try...catch** vì nếu có 100 routes thì phải **lặp đi lặp lại nhiều lần** try...catch nên ta thêm **1 middleware bao bọc bên ngoài controller** để bắt lỗi cho mình và không phải lồng nhiều try..catch nữa.

*module*.*exports* = *fn* => (*req*, *res*, *next*) => {

    return *Promise*.resolve(fn(*req*, *res*, *next*)).catch(*next*);

};

**Sử dụng**



**UPLOAD & DOWNLOAD**

Package hỗ trợ parse data dạng **File**: **npm install multer**

<form *action*="/admin/save" *method*="POST" *enctype*="multipart/form-data">

Thêm property **enctype** ở thẻ form có chứa iput **File**.

Lấy **File** từ **req** bằng **req.file**.

    <img *src*="/<%-product.imageUrl%>" *alt*="Card image" *style*="width:30%">

Bên **views** khi thêm “/” ở đầu đường dẫn như **src** của thẻ **<image> (Tương tự các thẻ có reference khác)** thì đường dẫn đó sẽ được đính trực tiếp lên **root**:

[**http://localhost:3000/\*...\***](http://localhost:3000/*...*)

Nếu ko có “/” thì nó sẽ nối tiếp vào url dag hiện hành. VD:

* Khi đang ở route admin

[**http://localhost:3000/admin/\*...\***](http://localhost:3000/admin/*...*)

app.use('/upload', express.static(path.join(\_\_dirname, 'upload')));

Khi có **req** với **route** là ‘/upload’ thì các file bên trong folder **upload** sẽ dc xét là đồng cấp với folder **upload**. (Khi gọi đến **/upload** thì giống như đang đứng trong thư mục **upload** nên ko thể truy cập đến /upload/upload/...)

Sử dụng module **fs,path** để **xóa** file dc upload lên hệ thống bằng phương thức **unlink();**

Từ bản NodeJS **14.3** trở đi ta có thể viết **await** bên ngoài **async function**

=> **Top-level await** (trên cùng của script)

**SOCKET IO**

**Package: npm install –save socket.io**

**SocketIO** bản từ 3.0 trở lên phải config thêm **CORS** mới hoạt động dc.

<https://socket.io/docs/v3/handling-cors/>

  const io = require('socket.io')(server, {

            cors: {

                origin: 'http://localhost:3000',

                methods: ['GET', 'POST']

            }

        });

        io.on('connection', *socket* => {

            console.log('Client connected');

        });

**socket** đại diện cho từng thằng **client** kết nối tới.

io.getIO().emit

**emit** sẽ bắn message đến tất cả user đang kết nối.

**broadcast** bắn message đến tất cả user **trừ** thằng bắn ra.

<https://socket.io/docs/v3/emit-cheatsheet/index.html>

**DEPLOY**

**Secure Response Header**: **npm install –save helmet**

Thêm header để chống attack.

**Compression**: **npm install –save compression**.

Nén assets để tải về client nhanh hơn. (css, js, **KO nén images**)

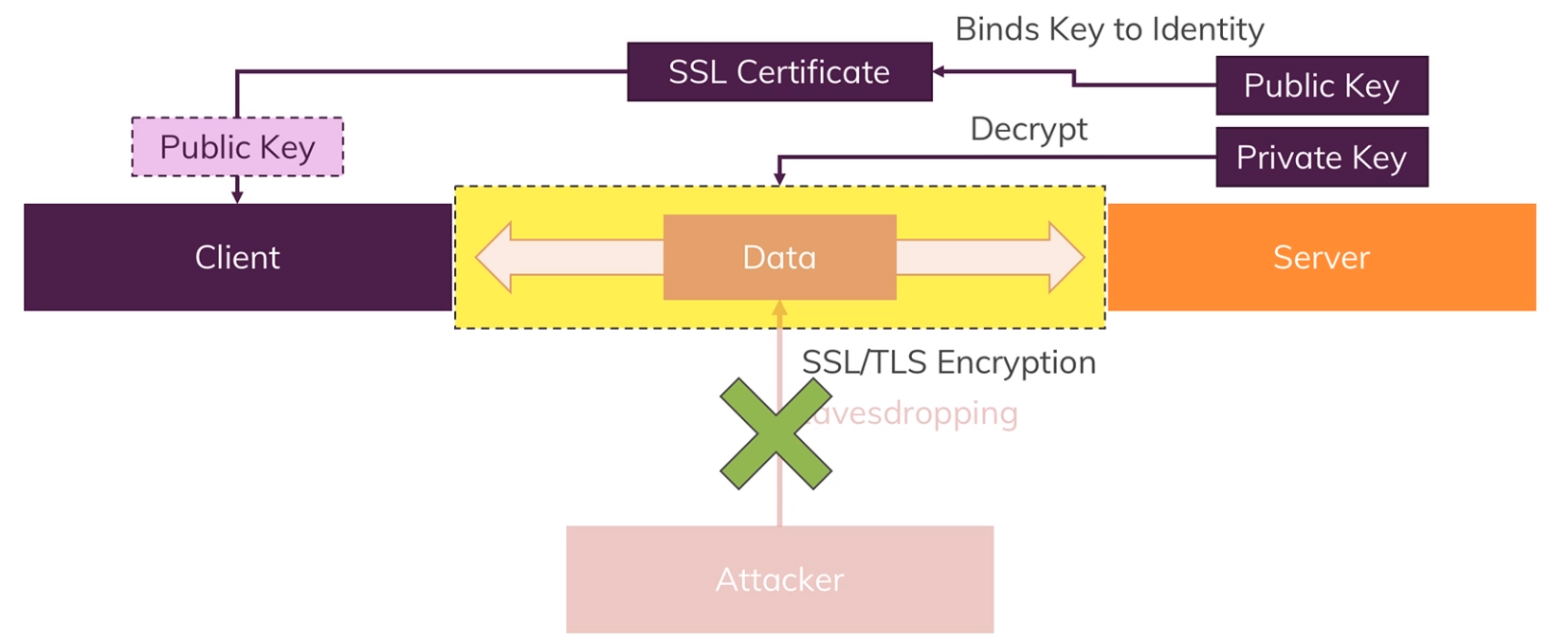
Lưu log bằng **Morgan**:

// 'a' means 'append' to access.log

const accessLogStream = fs.createWriteStream(path.join(\_\_dirname, 'access.log'), { flags: 'a' });

app.use(morgan('combined', { stream: accessLogStream }));

**SSL/TLS** : để bảo vệ dữ liệu gửi từ client.



**Attacker** nghe trộm dữ liệu từ **client -> server**, Với **SSL/TLS encryption** sẽ ngăn chặn dc việc này.

**Cách thức hoạt động:**

**Public Key** và **Private Key** sẽ được server định nghĩa trên **server**. **Private Key** chỉ 1 mình **server** dc biết vì sau sẽ dùng **Private Key** để decrypt data. **Public Key** sẽ dc dùng để **encrypt**.

**SSL Certificate** sẽ kết hợp **Public Key** với **Server** (domain, admin email,...). Mình sẽ set khi tạo Certificate. Sau khi kết hợp hai thứ trên **SSL Certificate** sẽ gửi về cho **client** (browser) và client sẽ nhận biết dc **Public Key** đó thuộc về **Server** nào.

Nếu mình tự tạo Key thì browser sẽ **KO TRUST** vì ko biết thông tin đó có đúng hay ko.

Vì thế nên mình hay nhận **Warnings** là trang web sử dụng SSL nhưng ko an toàn.

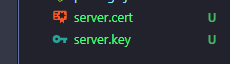
Trên môi trường **Production** mình sẽ sử dụng **SSL Certificate** dc cung cấp bởi cơ quan cấp chứng chỉ dc browser **TRUST**.

**Public Key** dc gửi về cho client thông qua **Certificate** và client có thể **encrypt data** và gửi lên cho server. Sau đó server có thể **decrypt data** đó với **Private Key**. Chỉ có Private Key đó mới có thể decrypt dc.

Tạo ở local 2 file : **server.key, server.cert**

**Server.key:** chứa private key của server.

**Server.cert**: SSL Certificate.



Tạo server bằng https => require(‘https’);

Đọc 2 file này bằng readFileSync(). Hàm này block code thực thi khi đọc xong 2 file này.

Tạo server https

 const server = https.createServer({ key: privateKey, cert: certificate }, app);

        server.listen(process.env.PORT || 8080);

Thông thường dịch vụ **Hosting** sẽ làm mọi việc config **SSL** này cho mình. Mình **KO** cần phải tự tay viết code.

**Nếu mình tự cấu hình server riêng ko xài Hosting service thì code trên có thể hỗ trợ mình config HTTPS.**

**Khi lên môi trường Production phải SET biến môi trường cho NODE**

NODE\_ENV=production

**KHÓA API NÂNG CAO**

Package: **npm install slugify**

Dùng để chuyển chuỗi thành chữ thường và nối với nhau bằng dấu “-“.

// Devcentral Bootcamp          devcentral-bootcamp

Khi các **routes** trùng nhau mà **logic middleware** khác thì có thể viết rút gọn bằng cách **nối** lại với nhau:

router.route('/')

    .get(asyncHandler(getBootcamps))

    .post(asyncHandler(createBootcamp));