

**HỌC VIỆN CÔNG NGHỆ BƯU CHÍNH VIỄN THÔNG**  
**KHOA CÔNG NGHỆ THÔNG TIN**

---



**BÁO CÁO THỰC TẬP CƠ SỞ**

**ĐỀ TÀI: PHÁT TRIỂN WEB ĐẶT LỊCH HẸN TRỰC TUYẾN  
CHO PHÒNG KHÁM HOẶC BỆNH VIỆN**

<b>Tên sinh viên</b>	<b>Trần Nhật Minh</b>
<b>Mã sinh viên</b>	<b>B22DCVT350</b>
<b>Lớp</b>	<b>E22CQCN02-B</b>

**Giáo viên phụ trách: Kim Ngọc Bách**

**Hà Nội 2025**

# TÌM HIỂU VỀ NODE.JS VÀ EXPRESS FRAMEWORK

## I. Tìm hiểu về Node.js

### 1. NodeJS là gì?

NodeJS là một mã nguồn được xây dựng dựa trên nền tảng Javascript V8 Engine, nó được sử dụng để xây dựng các ứng dụng web như các trang video clip, các forum và đặc biệt là trang mạng xã hội phạm vi hẹp. NodeJS là một mã nguồn mở được sử dụng rộng rãi bởi hàng ngàn lập trình viên trên toàn thế giới. NodeJS có thể chạy trên nhiều nền tảng hệ điều hành khác nhau từ Window cho tới Linux, OS X nên đó cũng là một lợi thế. NodeJS cung cấp các thư viện phong phú ở dạng Javascript Module khác nhau giúp đơn giản hóa việc lập trình và giảm thời gian ở mức thấp nhất.

### 2. Các đặc tính của NodeJS

- Không đồng bộ: Tất cả các API của NodeJS đều không đồng bộ (none-blocking), nó chủ yếu dựa trên nền của NodeJS Server và chờ đợi Server trả dữ liệu về. Việc di chuyển máy chủ đến các API tiếp theo sau khi gọi và cơ chế thông báo các sự kiện của Node.js giúp máy chủ để có được một phản ứng từ các cuộc gọi API trước (Realtime).
- Chạy rất nhanh: NodeJS được xây dựng dựa vào nền tảng V8 Javascript Engine nên việc thực thi chương trình rất nhanh.
- Đơn luồng nhưng khả năng mở rộng cao: Node.js sử dụng một mô hình luồng duy nhất với sự kiện lập. cơ chế tổ chức sự kiện giúp các máy chủ để đáp ứng một cách không ngăn chặn và làm cho máy chủ cao khả năng mở rộng như trái ngược với các máy chủ truyền thống mà tạo ra hạn chế để xử lý yêu cầu. Node.js sử dụng một chương trình đơn luồng và các chương trình tương tự có thể cung cấp dịch vụ cho một số lượng lớn hơn nhiều so với yêu cầu máy chủ truyền thống như Apache HTTP Server.

- Không đệm:

NodeJS không đệm bất kì một dữ liệu nào và các ứng dụng này chủ yếu là đầu ra dữ liệu.

- Có giấy phép: NodeJS đã được cấp giấy phép bởi MIT License.

### 3. NodeJS hoạt động thế nào

Ý tưởng chính của Node js là sử dụng non-blocking, hướng sự vào ra dữ liệu thông qua các tác vụ thời gian thực một cách nhanh chóng. Bởi vì, Node js có khả năng mở rộng nhanh chóng, khả năng xử lý một số lượng lớn các kết nối đồng thời bằng thông lượng cao. Nếu như các ứng dụng web truyền thống, các request tạo ra một luồng xử lý yêu cầu mới và chiếm RAM của hệ thống thì việc tài nguyên của hệ thống sẽ được sử dụng không hiệu quả. Chính vì lẽ đó giải pháp mà Node js đưa ra là sử dụng luồng đơn (Single-Threaded), kết hợp với non-blocking I/O để thực thi các request, cho phép hỗ trợ hàng chục ngàn kết nối đồng thời.

Một số lệnh cơ bản:

1.Lệnh in ra màn hình: `console.log('Hello, world!');`

2. Đọc biến môi trường:

```
console.log(process.env);  
console.log(process.env.PATH);
```

3.Lấy tham số dòng lệnh:

```
console.log('Các tham số:', process.argv);
```

#### 4. Đọc và ghi file bằng module tạo file file.js:

```
file.js x
Node JS > file.js > ...
1  const fs = require('fs');
2
3  // Ghi nội dung vào file 'example.txt'
4  fs.writeFileSync('example.txt', 'Nội dung ban đầu', 'utf8');
5
6  // Đọc lại nội dung file
7  const data = fs.readFileSync('example.txt', 'utf8');
8  console.log('Nội dung file:', data);
9
```

sử dụng lệnh: node file.js vào terminal:

```
PS E:\Hoc hanh\Hoc Frontend\Node JS> node file.js
Nội dung file: Nội dung ban đầu
```

#### 5. Tạo một server HTTP đơn giản

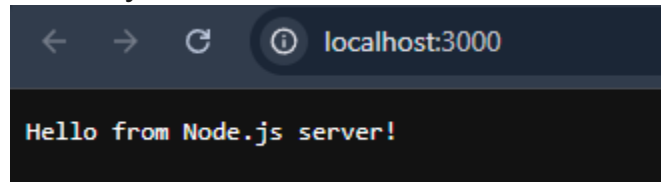
tạo file server.js:

```
Node JS > server.js > ...
1  const http = require('http');
2
3  // Tạo server
4  const server = http.createServer((req, res) => {
5    res.writeHead(200, {'Content-Type': 'text/plain'});
6    res.end('Hello from Node.js server!');
7  });
8
9  // Lắng nghe trên cổng 3000
10 server.listen(3000, () => {
11   console.log('Server đang chạy tại http://localhost:3000');
12 });
13
```

sử dụng lệnh node server.js vào terminal:

```
PS E:\Hoc hanh\Hoc Frontend\Node JS> node server.js
Server đang chạy tại http://localhost:3000
```

khi chạy localhost:3000



## 4.NPM: The Node Package Manager

Khi thảo luận về Node js thì một điều chắc chắn không nên bỏ qua là xây dựng package quản lý sử dụng các cộng cụ NPM mà mặc định với mọi cài đặt Node js. Ý tưởng của mô-đun NPM là khá tương tự như Ruby-Gems: một tập hợp các hàm có sẵn có thể sử dụng được, thành phần tái sử dụng, tập hợp các cài đặt dễ dàng thông qua kho lưu trữ trực tuyến với các phiên bản quản lý khác nhau.

Danh sách các mô-đun có thể tìm trên web NPM package hoặc có thể truy cập bằng cách sử dụng công cụ NPM CLI sẽ tự động cài đặt với Node js.

Một số các module NPM phổ biến nhất hiện nay là:

+[expressjs.com/](https://expressjs.com/) - Express.js, một Sinatra-inspired web framework khá phát triển của Node.js, chứa rất nhiều các ứng dụng chuẩn của Node.js ngày nay.

+connect - Connect là một mở rộng của HTTP server framework cho Node.js, cung cấp một bộ sưu tập của hiệu suất cao "plugins" được biết đến như là trung gian; phục vụ như một nền tảng cơ sở cho Express

+[socket.io](https://socket.io/) and sockjs - Hai thành phần Server-side websockets components nổi tiếng nhất hiện nay.

+Jade - Một trong những engines mẫu, lấy cảm hứng từ HAML, một phần mặc định trong Express.js.

+mongo and mongojs - MongoDB hàm bao để cung cấp các API cho cơ sở dữ liệu đối tượng trong MongoDB Node.js

+redis - thư viện Redis client.

+coffee-script - CoffeeScript trình biên dịch cho phép developers viết các chương trình Node.js của họ dùng Coffee.

+underscore (lodash, lazy) - Thư viện tiện ích phổ biến nhất trong JavaScript, package được sử dụng với Node.js, cũng như hai đối tác của mình, hứa hẹn hiệu suất tốt hơn bằng cách lấy một cách tiếp cận thực hiện hơi khác nhau.

+forever - Có lẽ là tiện ích phổ biến nhất để đảm bảo rằng một kịch bản nút cho chạy liên tục. Giữ quá trình Node.js của bạn lên trong sản xuất đối mặt với bất kỳ thất bại không ngờ tới.

## II. Tìm hiểu về Express Framework

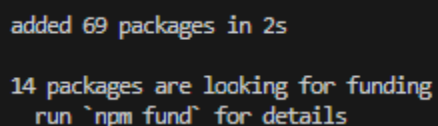
### 1. Express Framework là gì

Express 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. Express hỗ trợ các method HTTP và middleware tạo ra API vô cùng mạnh mẽ và dễ sử dụng.

### 2. Các tính năng nổi bật của Express

- ▷ Thiết lập router cho phép sử dụng với các hành động khác nhau dựa trên phương thức HTTP và URL.
- ▷ Hỗ trợ xây dựng theo mô hình MVC
- ▷ Cho phép định nghĩa middleware giúp tổ chức và tái sử dụng code
- ▷ Hỗ trợ RESTful API

Để sử dụng express ta cần tải gói package express.  
nhập lệnh: `npm install express` vào terminal



```
added 69 packages in 2s
14 packages are looking for funding
run `npm fund` for details
```

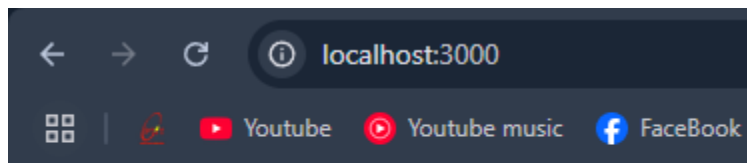
sau khi tải xong sẽ có dòng lệnh này

## 2.1. Khởi tạo 1 server Express:

1. Khởi tạo thư mục (Create directory)
2. Khởi tạo file index.js (Create index.js file)
3. Khởi tạo npm (Initialize npm)
4. Cài đặt gói Express (Install the Express package)
5. Viết ứng dụng server trong file index.js (Write server application in index.js)
6. Bắt đầu server (Start server)

```
server.js x
Node JS > server.js > app.listen() callback
1  const express = require('express');
2  const app = express();
3  app.(parameter) res: Response<any, Record<string, any>, number>
4    res.send('Hello World!')
5  });
6  const port = 3000;
7
8  app.listen(port, () => {
9    console.log(`listening on port ${port}!`)
10 });
11
```

```
PS E:\Hoc hanh\Hoc Frontend\Node JS> node server.js
listening on port 3000!
```



Hello World!

## 2.2.Routing

Routing/định tuyến đề cập đến cách các endpoint(URI) của ứng dụng phản hồi các yêu cầu từ client.

Ta xác định định tuyến bằng các phương thức của đối tượng trong ứng dụng Express tương ứng với các phương thức HTTP. Ví dụ: `app.get()` để xử lý yêu cầu GET và `app.post` để xử lý yêu cầu POST. Tương tự với các phương thức khác như PUT, PATH, DELETE,... Ta cũng có thể sử dụng `app.all()` để xử lý tất cả các phương thức HTTP và `app.use` để chỉ định middleware cần thiết (ta sẽ nói về middleware sau).

Các phương thức định tuyến này chỉ định một hàm callback (đôi khi được gọi là "handler function"), chúng được gọi khi ứng dụng nhận được yêu cầu đến tuyến được chỉ định (endpoint) và phương thức HTTP. Nói cách khác, nó sẽ "lắng nghe" các yêu cầu ứng với (các) tuyến và (các) phương thức được chỉ định, và khi thấy trùng, ứng dụng sẽ gọi hàm callback được chỉ định.

ví dụ:

```
var express = require('express')
var app = express()
```

```
app.get('/', function (req, res) {
  res.send('hello world')
})
```

Về hai tham số là req và res ta sẽ tìm hiểu như sau:

## 2.3.Request

Request viết tắt là req là các yêu cầu từ phía client đến server. Như ở ví dụ trên thì request có vẻ không cần thiết, nhưng thực tế ta sẽ cần rất nhiều thông tin từ request, như body từ phương thức POST, thông tin từ đường dẫn, các yêu cầu xác thực,... Vì thế có rất rất nhiều loại request, nhưng ở đây ta chỉ cần quan tâm đến những loại thường dùng là:

### **req.body**

Lấy phần thông tin từ body của request. Sau phiên bản Express.4x để lấy thông tin từ phương thức POST của HTTP. Ta phải sử dụng body-parser.



## req.params

Lấy tham số từ parameter của đường dẫn. Ví dụ như ta có link là: GET /api/users/:name

## req.query

Tương tự như req.params nhưng là lấy query từ link

## req.header

Lấy thông tin từ header của HTTP. Chủ yếu dùng để xác thực người dùng

### 2.4.Response

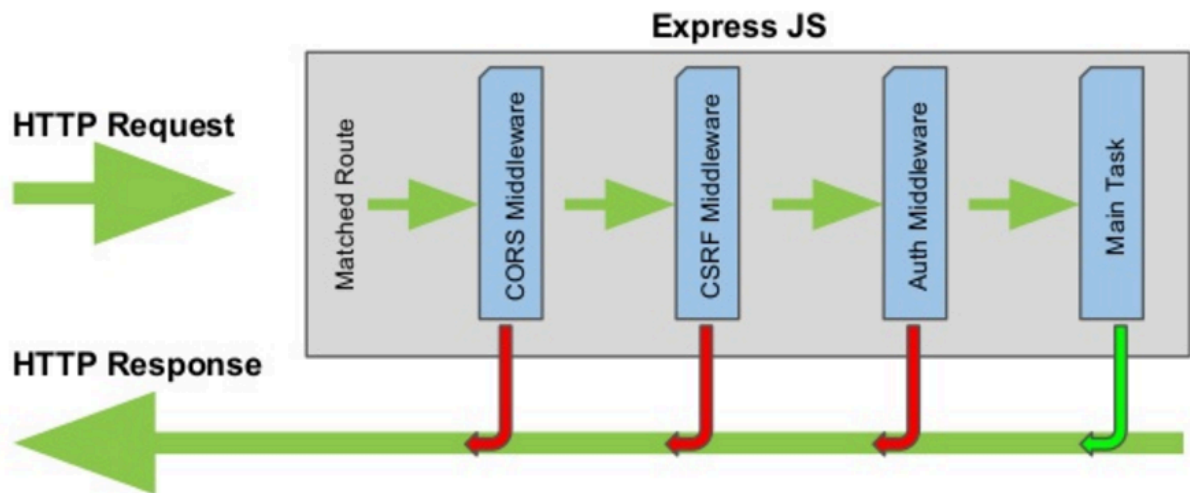
Response viết tắt là res. Là phản hồi từ phía server đến client. Có thể là gửi về dữ liệu mà client yêu cầu, hoặc thực hiện các thao tác mà client mong muốn, như chuyển hướng, render đến template,... Ta có bảng tóm tắt các res thường dùng như sau.

Method	Description
res.download()	Tệp được tải xuống
res.end()	Kết thúc quá trình response
res.json()	Gửi về file json
res.jsonp()	Gửi về file json hỗ trợ JSONP
res.redirect()	Chuyển hướng request
res.render()	Render ra giao diện theo template
res.send()	Gửi về nhiều loại tệp khác nhau

<code>res.sendFile()</code>	Gửi về dưới dạng luồng octet
<code>res.status()</code>	Gửi về trạng thái của HTTP phản hồi

## 2.5. Các loại Middleware

Middleware là các hàm được dùng để tiền xử lý, lọc các request trước khi đưa vào xử lý logic hoặc điều chỉnh các response trước khi gửi về cho người dùng.



Hình trên mô tả khi một request gửi đến Express sẽ được xử lý qua 5 bước như sau :

- Tìm định tuyến tương ứng với request
- Dùng CORS Middleware để kiểm tra cross-origin Resource sharing của request
- Dùng CSRF Middleware để xác thực CSRF của request, chống fake request
- Dùng Auth Middleware để xác thực request có được truy cập hay không
- Xử lý công việc được yêu cầu bởi request (Main Task)

Bất kỳ bước nào trong các bước 2,3,4 nếu xảy ra lỗi sẽ trả về response thông báo cho người dùng, có thể là lỗi CORS, lỗi CSRF hay lỗi auth tùy thuộc vào request bị dừng ở bước nào.

Hàm middleware thường có dạng như sau:

```
async function name(req, res, next) {  
  await something  
  next();  
}
```

Ta thấy hàm middleware sử dụng 3 tham số `req`, `res` và `next`, hai tham số trước ta đã biết còn `next` được dùng để chuyển tiếp sang hành động kế tiếp. Trong một vài trường hợp ta sẽ sử dụng tham số `err`, đối tượng lỗi. Các chức năng middleware có thể thực hiện các tác vụ sau:

- 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

Trong Express, có 5 kiểu middleware có thể sử dụng :

- Application-level middleware (middleware cấp ứng dụng)
- Router-level middleware (middleware cấp điều hướng - router)
- Error-handling middleware (middleware xử lý lỗi)
- Built-in middleware (middleware sẵn có)
- Third-party middleware (middleware của bên thứ ba)

## Application-level middleware

Ở đây ta xây dựng application-level middleware của ứng dụng bằng cách sử dụng `app.use()` và `app.METHOD` trong đó METHOD là phương thức HTTP của request mà middleware xử lý như (GET, PUT hoặc POST).

Ví dụ dưới đây mô tả một hàm ko khai báo đường dẫn cụ thể, do đó nó sẽ được thực hiện mỗi lần request:

```
const express = require('express')
```

```
const app = express()
```

```
app.use(function (req, res, next) {  
  console.log('Time:', Date.now())  
  next() })
```

Ví dụ dưới đây dùng hàm use đến đường dẫn `/user/:id`. Hàm này sẽ được thực hiện mỗi khi request đến đường dẫn `/user/:id` bất kể phương thức HTTP nào:

```
app.use('/user/:id', function (req, res, next) {  
  console.log('Request Type:', req.method)  
  next()  
})
```

Tiếp theo là một ví dụ cho hàm được thực hiện mỗi khi truy cập đến đường dẫn `/user/:id` bằng phương thức GET:

```
app.get('/user/:id', function (req, res, next) {  
  res.send('USER')  
})
```

Khi muốn gọi một loạt hàm middleware cho một đường dẫn cụ thể, chúng ta có thể thực hiện như ví dụ dưới đây, bằng cách khai báo liên tiếp các tham số là các hàm sau tham số đường dẫn:

```
app.use('/user/:id', function (req, res, next) {  
  console.log('Request URL:', req.originalUrl)  
  next()  
}, function (req, res, next) {  
  console.log('Request Type:', req.method)  
  next()  
})
```

Hoặc ta có thể tách ra thành 2 lần khai báo `app.use`, gọi là multiple routes, tuy nhiên ở các hàm phía trước cần gọi hàm `next()` khi kết thúc mỗi hàm, nếu không như ví dụ dưới đây, route thứ 2 sẽ không bao giờ được thực hiện do hàm thứ 2 trong route thứ nhất không gọi đến hàm `next()`:

```
app.get('/user/:id', function (req, res, next) {  
  console.log('ID:', req.params.id)  
  next()  
}, function (req, res, next) {
```

```

    res.send('User Info')
  })

  // handler for the /user/:id path, which prints the user ID
  app.get('/user/:id', function (req, res, next) {
    res.end(req.params.id)
  })

```

*Lưu ý:* Khi muốn bỏ qua các hàm middleware tiếp theo không thực hiện nữa, chúng ta sẽ sử dụng lệnh `next('route')`, tuy nhiên việc này chỉ tác dụng với các hàm middleware được load thông qua hàm `app.METHOD` hoặc `router.METHOD`.

Ví dụ dưới đây mô tả một hàm middleware sẽ kết thúc ngay lập tức khi tham số `id=0`:

```

app.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next route
  if (req.params.id === '0') next('route')
  // otherwise pass the control to the next middleware function in this stack
  else next()
}, function (req, res, next) {
  // render a regular page
  res.render('regular')
})

// handler for the /user/:id path, which renders a special page
app.get('/user/:id', function (req, res, next) {
  res.render('special')
})

```

## Router-level middleware

Các middleware này về chức năng không khác gì so với application-level middleware ở trên, tuy nhiên thay vì dùng biến `app` thì ta dùng `router` để chuyên biệt về việc định tuyến:

```
const router = express.Router()
```

Router-level middleware sử dụng `route.use()` và `router.METHOD()`. Phần code dưới đây mô tả một cách sử dụng router để thiết lập các route cần thiết cho một resource có tên là user:

```
const express = require('express')
const app = express()
const router = express.Router()
```

// a middleware function with no mount path. This code is executed for every request to the router

```
router.use(function (req, res, next) {
  console.log('Time:', Date.now())
  next()
})
```

// a middleware sub-stack shows request info for any type of HTTP request to the /user/:id path

```
router.use('/user/:id', function (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}, function (req, res, next) {
  console.log('Request Type:', req.method)
  next()
})
```

// a middleware sub-stack that handles GET requests to the /user/:id path

```
router.get('/user/:id', function (req, res, next) {
  // if the user ID is 0, skip to the next router
  if (req.params.id === '0') next('route')
  // otherwise pass control to the next middleware function in this stack
  else next()
}, function (req, res, next) {
  // render a regular page
  res.render('regular')
})
```

// handler for the /user/:id path, which renders a special page

```
router.get('/user/:id', function (req, res, next) {
  console.log(req.params.id)
  res.render('special')
})
```

```
// mount the router on the app
app.use('/', router)
```

## Error-handling middleware

Đây là các middleware phục vụ cho việc xử lý lỗi. Một lưu ý là các hàm cho việc này luôn nhận bốn tham số (*err*, *req*, *res*, *next*). Khi muốn khai báo một middleware cho việc xử lý lỗi, cần phải tạo một hàm có 4 tham số đầu vào. Mặc dù ta có thể không cần sử dụng đối tượng *next*, nhưng hàm vẫn cần format với bốn tham số như vậy. Nếu không Express sẽ không thể xác định đó là hàm xử lý lỗi, và sẽ không chạy khi có lỗi xảy ra, chỉ hoạt động giống như các hàm middleware khác.

Đoạn code dưới đây mô tả một hàm xử lý lỗi truyền về cho client lỗi 500 khi có lỗi xảy ra từ server:

```
app.use(function (err, req, res, next) {
  console.error(err.stack)
  res.status(500).send('Something broke!')
})
```

## Built-in middleware

Kể từ phiên bản 4.x, Express đã không còn phụ thuộc vào thư viện [Connect](#). Tất cả các hàm middleware trước đây đều đã được tách ra thành các modules riêng biệt. Điều này cung cấp cách tối ưu hóa và tùy chỉnh ứng dụng Express một cách linh hoạt nhất, giúp tạo ra một ứng dụng Web Application phù hợp với nhu cầu, không bị thừa những thứ không cần thiết.

Express có các hàm built-in middleware sau:

- `express.static`: các file tĩnh như hình ảnh, HTML, ....
- `express.json`: phân tích cú pháp request đến với JSON payload. *Lưu ý*: Chỉ khả dụng từ Express 4.16.0+

- `express.urlencoded`: phân tích cú pháp request đến với URL-encoded payloads. *Lưu ý*: Chỉ khả dụng từ Express 4.16.0+

## Third-party middleware

Sử dụng Third-party sẽ giúp ta thêm các chức năng cho vào web app của mình mà không cần mất công triển khai. Ta sẽ cần cài đặt module thông qua npm, sau đó khai báo sử dụng trong đối tượng `app` nếu dùng ở Application-level, hoặc qua đối tượng `router` nếu dùng ở Router-level