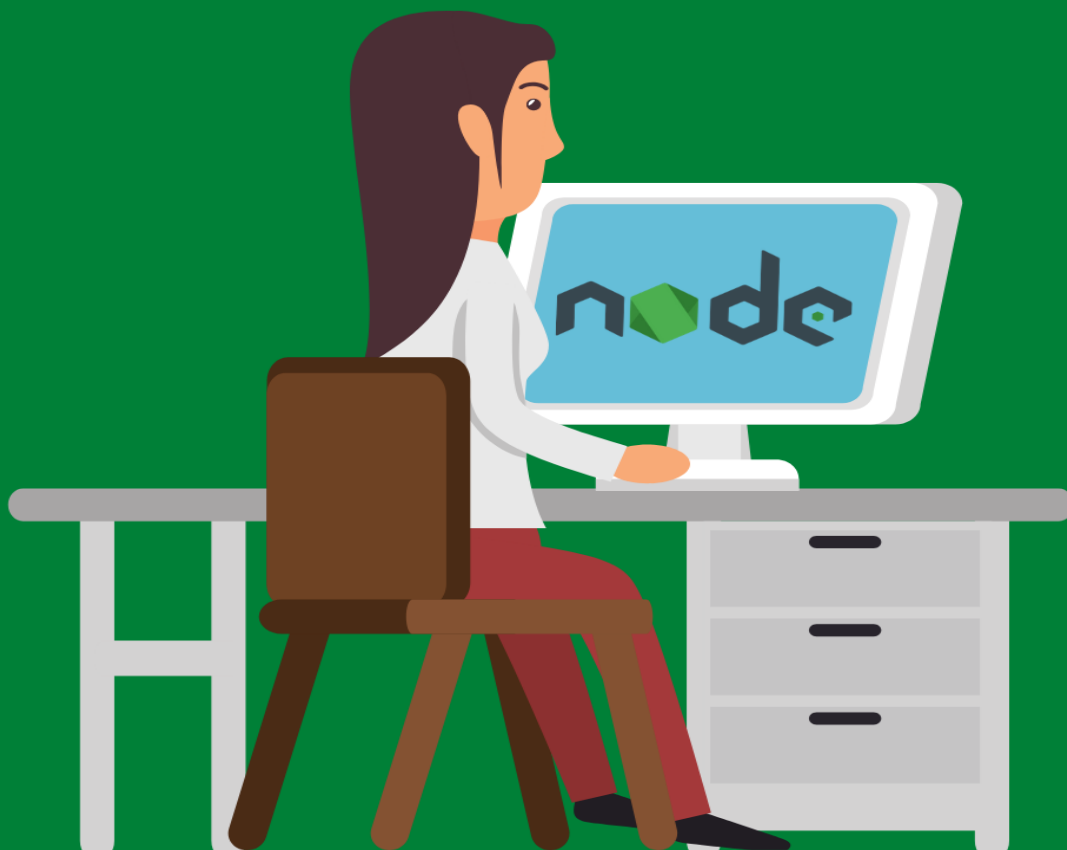


TỰ XÂY DỰNG ỨNG DỤNG TỪ A-Z

LẬP TRÌNH NODE.JS THẬT ĐƠN GIẢN

BY VNTALKING.COM



MỤC LỤC

| | |
|--|-----------|
| Lời nói đầu..... | 5 |
| Cách học đúng cách | 5 |
| Yêu cầu trình độ..... | 6 |
| Liên hệ với tác giả..... | 6 |
| Phần 1: Giới thiệu | 7 |
| Giới thiệu ứng dụng blog sẽ xây dựng | 7 |
| Node.js là gì?..... | 8 |
| Cài đặt Node.js | 9 |
| Tạo server Nodejs đầu tiên | 10 |
| Hiểu hơn về request và response | 12 |
| Phần 2: Giới thiệu về NPM và Express..... | 15 |
| Cài đặt Custom Package với NPM..... | 15 |
| Giới thiệu Express..... | 18 |
| Tổng kết..... | 24 |
| Phần 3: Bước đầu tạo web app với Express | 25 |
| Tải web template | 25 |
| Tự động khởi động server mỗi khi thay đổi mã nguồn | 26 |
| Npm start..... | 27 |
| Tạo thư mục public chứa tệp static | 27 |
| Tạo Page routes | 30 |
| Tổng kết..... | 30 |
| Phần 4: Templating Engine | 31 |
| Giới thiệu Template engine | 31 |
| Layout | 33 |
| Tổng kết..... | 35 |
| Phần 5: Giới thiệu MongoDB | 36 |
| Kiến trúc của MongoDB..... | 37 |
| Cài đặt MongoDB | 37 |
| Kết nối và quản lý MongoDB với Robo 3T..... | 39 |

| | |
|---|-----------|
| Cài đặt Mongoose | 40 |
| Kết nối MongoDB từ Node.js | 40 |
| Định nghĩa Model..... | 40 |
| Tạo các action CRUD với Mongoose model | 41 |
| Tổng kết..... | 43 |
| Phần 6: Ứng dụng MongoDB vào dự án | 44 |
| Lưu dữ liệu bài Post vào Database | 47 |
| Hiển thị danh sách các bài Post..... | 48 |
| Hiển thị dữ liệu động với Template engine | 49 |
| Hiển thị nội dung một Post | 51 |
| Thêm Fields và Schema | 52 |
| Tổng kết..... | 53 |
| Phần 7: Tạo tính năng upload ảnh với Express | 54 |
| Tổng kết..... | 58 |
| Phần 8: Tìm hiểu Express Middleware..... | 59 |
| Middleware tùy chỉnh..... | 59 |
| Tạo và đăng ký Validation middleware | 60 |
| Tổng kết..... | 61 |
| Phần 9: Refactoring theo mô hình MVC..... | 62 |
| Giới thiệu mô hình MVC | 62 |
| Tiến hành Refactoring..... | 63 |
| Tổng kết..... | 66 |
| Phần 10: Tạo tính năng đăng ký thành viên..... | 67 |
| User Model | 69 |
| Controller xử lý đăng ký user | 70 |
| Mã hóa mật khẩu | 71 |
| Mongoose Validation | 73 |
| Tạo tính năng đăng nhập | 74 |
| Tổng kết..... | 77 |
| Phần 11: Xác thực với Express Sessions | 78 |
| Implementing User Sessions..... | 79 |

| | |
|--|-----------|
| Protect một Pages nào đó với Authentication Middleware | 81 |
| User Logout..... | 83 |
| Tạo trang 404..... | 84 |
| Tổng kết..... | 86 |
| Phần 12: Triển khai web app lên server thật..... | 87 |
| Cài đặt server | 87 |
| Đưa sourcecode Node.js lên VPS | 89 |
| Quản lý ứng dụng Node.js bằng PM2 | 90 |
| Kết nối domain vào vps | 91 |
| Cấu hình Nginx Reverse Proxy Server | 91 |
| Chào tạm biệt..... | 92 |
| Tài liệu tham khảo | 93 |

Lời nói đầu

Node.js đang trở thành một xu hướng của giới lập trình back-end. Có rất nhiều ứng dụng lớn của các đại gia đang sử dụng Nodejs. Có thể kể tên như: Paypal, Netflix, LinkedIn...

Mục tiêu của cuốn sách này đó là giúp các bạn bước vào thế giới của Node.js một cách vững chắc nhất. Tức là bạn sẽ hiểu rõ được bản chất, cách xây dựng ứng dụng Nodejs một cách bài bản nhất. Nếu không quá khi gọi là "vũ trụ Node.js". "Vũ trụ Node.js" bắt nguồn từ viên gạch Javascript.

Để khám phá "Vũ trụ Node.js" một cách trơn tru, cuốn sách này sẽ giúp các bạn tìm hiểu "tam trụ" cơ bản của Node.js, đó là Node.js, ExpressJS, và MongoDB.

Mỗi phần trong cuốn sách này sẽ được trình bày thẳng vào vấn đề, kiến thức trọng tâm để tránh mất thời gian vàng ngọc của bạn.

Sau khi bạn đọc xong cuốn sách này, bạn sẽ đủ kỹ năng để tự mình xây dựng một web app bằng Nodejs và triển khai nó trên Internet.

Cách học đúng cách

Cuốn sách này mình chia nhỏ nội dung thành 12 phần, mỗi phần sẽ giới thiệu một chủ đề riêng biệt. Mục đích là để bạn có thể chủ động lịch học, không bị dồn nén quá nhiều.

Với mỗi phần lý thuyết, mình đều có ví dụ minh họa và code luôn vào dự án. Vì vậy, cách học tốt nhất vẫn là vừa học vừa thực hành. Bạn nên **tự mình viết lại từng dòng code** và chạy nó. Đừng copy cả đoạn code trong sách, điều này sẽ làm hạn chế khả năng viết code của bạn, cũng như làm bạn nhiều khi không hiểu vì sao code bị lỗi.

Nhớ nhé, đọc đến đâu, tự viết code đến đó, tự build và kiểm tra đoạn code đó chạy đúng không!

Yêu cầu trình độ

Cuốn sách này mình xây dựng từ những kiến thức nền tảng Node.js từ cơ bản nhất. Nên không cần bạn phải có kiến thức về Node.js.

Tuy nhiên, vì Node.js được xây dựng trên ngôn ngữ Javascript nên sẽ tốt hơn nếu bạn đã có kiến thức căn bản về Javascript. Ngoài ra, bạn cũng cần chút hiểu biết về HTML và CSS để dựng giao diện web.

Liên hệ với tác giả

Nếu có bất kỳ vấn đề gì trong quá trình học, code bị lỗi hoặc không hiểu, các bạn có thể liên hệ với mình qua một trong những hình thức dưới đây:

- Website: <https://vntalking.com>
- Fanpage: <https://facebook.com/vntalking>
- Email: support@vntalking.com
- Github: <https://github.com/vntalking/nodejs-express-mongodb-co-ban>

Node.js là một JavaScript runtime để chạy Javascript phía server. Có rất nhiều công ty đã sử dụng Node.js để xây dựng cho ứng dụng của họ. Có thể kể một số tên tuổi đình đám như: WalMart, LinkedIn, PayPal, YouTube, Yahoo!, Amazon.com, Netflix, eBay và Reddit.

Trong cuốn sách này, chúng ta sẽ học Node.js kết hợp với Express và mongoDB để xây dựng một blog từ đầu, từ con số 0. Trong quá trình đọc và thực hành theo sách, bạn sẽ hiểu và tự mình xây dựng một ứng dụng riêng từ những kỹ thuật trong cuốn sách này.

Chúng ta sẽ cùng nhau tìm hiểu một loạt những kỹ thuật như xác thực người dùng, validate dữ liệu, bất đồng bộ trong Javascript, Express, MongoDB và template engine.v.v...

Giới thiệu ứng dụng blog sẽ xây dựng

Như mình đã giới thiệu ở trên, để việc học có hiệu quả, thay vì chỉ có những đoạn code mẫu ngắn và không liên quan tới nhau, chúng ta sẽ vừa học vừa xây dựng một ứng dụng hoàn chỉnh. Tiêu chí là "*học đến đâu, ứng dụng đến đó*".

Đây là giao diện của ứng dụng:



Hình 1.1: Template giao diện dùng trong sách

Với trang blog này, người dùng có thể đăng ký tài khoản mới. Sau khi đăng ký xong thì họ có thể về trang chủ, đăng nhập vào blog. Thanh navigation bar sẽ hiển thị những menu khác nhau tùy thuộc vào trạng thái người đã đăng nhập rồi hay chưa. Để xây dựng giao diện, chúng ta sẽ sử dụng EJS template engine.

Sau khi người dùng đăng nhập, thanh navigator bar sẽ hiển thị text "LogOut" để họ có thể đăng xuất nếu cần. Ngoài ra, còn thêm một menu "new post" để họ có thể tạo bài viết mới, upload ảnh lên blog. Sau khi họ tạo bài viết xong thì quay trở lại trang chủ, blog sẽ hiển thị danh sách các bài viết đã published.

Thông qua việc thực hành xây dựng ứng dụng blog này, bạn sẽ nắm chắc được kiến thức về Node.js, kết hợp với express và MongoDB.

Node.js là gì?

Trước khi tìm hiểu khái niệm Node.js là gì, bạn cần phải hiểu cơ chế Internet hoạt động như thế nào đã. Khi một người dùng mở trình duyệt, cô ấy vào một website như vntalking.com chẳng hạn. Như vậy là cô ấy đã tạo request tới server, lúc này cô ấy/trình duyệt được coi là một client. Khi client tạo request tới server và server phản hồi lại là nội dung của trang web mà client yêu cầu.

Có một số ngôn ngữ lập trình được thiết kế để viết các ứng dụng cho server như PHP, Ruby, Python, ASP, Java... Nếu trước kia, Javascript được thiết kế để chạy trên các trình duyệt, cung cấp thêm khả năng tương tác của trang web với người dùng. Ví dụ như các menu có hiệu ứng dropdown, hay hiệu ứng tuyết rơi...

Nhưng mọi chuyện đã thay đổi vào năm 2009, khi Node.js ra đời, sử dụng dụng V8 engine làm bộ chạy các mã Javascript. Giờ đây, Javascript đã vượt ra khỏi khuôn khổ của trình duyệt, và cho phép nó chạy trên các server. Như vậy, ngoài các ngôn ngữ dành riêng cho server như PHP, Golang, JAVA... thì Javascript đã trở thành một lựa chọn sáng giá khác.

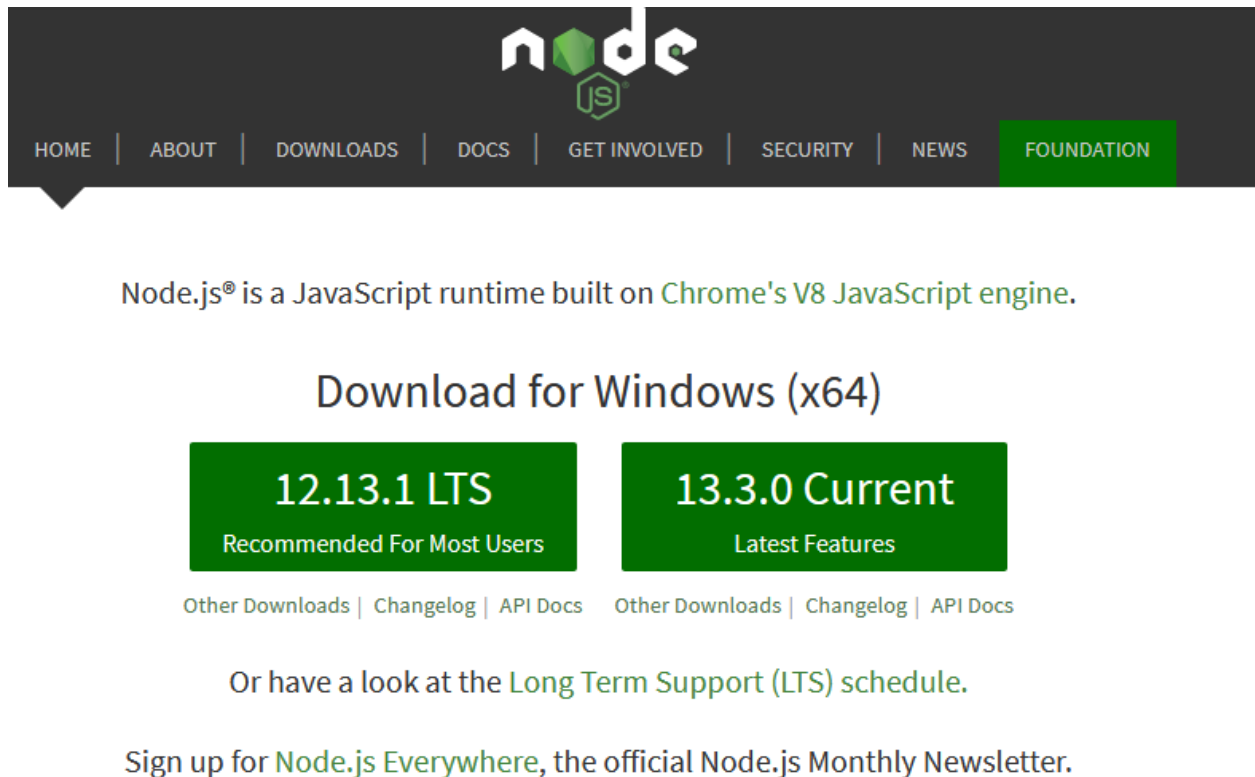
Những lợi ích mà Node.js mang lại cho bạn:

- Đầu tiên, V8 JavaScript engine là một Javascript engine mạnh mẽ, được sử dụng trong trình duyệt Chrome của Google. Điều này sẽ làm ứng dụng của bạn có tốc độ rất nhanh.
- Node.js khuyến khích viết mã kiểu asynchronous (bất đồng bộ) để cải thiện tốc độ ứng dụng, tránh được những vấn đề phát sinh của việc sử dụng đa luồng.
- Thứ 3 đó là Javascript là một ngôn ngữ rất phổ biến, do vậy bạn sẽ thừa hưởng rất nhiều thư viện hay ho mà lại miễn phí.

- Cuối cùng là do Node.js cũng sử dụng javascript, do vậy bạn sẽ tận dụng được những kiến thức đã có từ trước, khi bạn viết ứng dụng dùng Javascript trên trình duyệt. Giờ đây, thay vì phải tìm hiểu thêm một ngôn ngữ mới, bạn chỉ cần biết một mình Javascript là đủ full stack rồi.

Cài đặt Node.js

Để cài đặt Node.js, bạn vào trang chủ nodejs.org (hình 1.2) và tải phiên bản tương ứng với hệ điều hành trên máy tính của bạn.



Hình 1.2: Download node.js

Việc cài đặt diễn ra cũng đơn giản. Nếu máy bạn dùng window thì cài đặt như mọi phần mềm khác thôi. Bạn có thể tham khảo chi tiết các cài đặt chi tiết [tại đây](#).

Sau khi cài xong, bạn có thể kiểm tra version bằng lệnh sau:

```
node -v
```

Ngoài ra, khi cài đặt Node.js, bạn sẽ được khuyến mãi cả NPM nữa.

```
npm -v
```

Kết quả hiện ra như hình 1.3 là được.

```
Microsoft Windows [Version 6.1.7600]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\MinhVu>node -v
v8.11.4

C:\Users\MinhVu>npm -v
5.6.0
```

Hình 1.3: kiểm tra phiên bản node.js

Tạo server Nodejs đầu tiên

Chúng ta sẽ tạo một server đơn giản nhất bằng Node.js để các bạn dễ hiểu hơn về cách một client tạo request tới server và server phản hồi lại request đó như thế nào.

Bạn có thể sử dụng bất kỳ trình soạn thảo code để viết code. Như mình thì vẫn đề xuất các bạn sử dụng Visual Studio Code. Giờ mình sẽ tạo một thư mục và dùng Visual Studio Code (từ giờ mình sẽ viết tắt là VS) để mở thư mục ấy. Các bạn tạo mới một file Javascript, đặt tên là index.js. Dưới đây là đoạn code để tạo một http server đơn giản:

```
const http = require('http')

const server = http.createServer((req, res) => {
  console.log(req.url)
  res.end('VNTALKING: Xin chào Node.js')
})
server.listen(3000)
```

Giải thích code

```
const http = require('http')
```

Hàm require có tác dụng là *import* một module vào tệp bạn đang xử lý. Hàm này trong Node.js giống như hàm import hay include trong các ngôn ngữ khác. Tham số đầu vào của hàm require là tên của module được định nghĩa bằng từ khóa export và trả về package đó.

Quay lại đoạn code trên, mình require một module có tên là *http* và gán nó cho biến cũng có tên là http (mình đặt trùng để cho dễ nhận biết biến này là package nào thôi, chứ bạn có thể đặt biến bất kỳ tên nào bạn muốn).

http là một module được tích hợp sẵn trong Node.js, dùng để cung cấp các phương thức tương tác với server như GET, POST, UPDATE...

```
const server = http.createServer(...)
```

Đoạn code này giúp chúng ta khởi tạo một server. Hàm này có tham số là một hàm số với hai tham số: request và response.

```
const server = http.createServer((req, res) => {  
  console.log(req.url)  
  res.end('VNTALKING: Xin chào Node.js')  
})
```

Thực ra hàm được truyền vào `createServer()` cũng không phải là một khái niệm mới mẻ đâu. Đó chính là callback. Hàm callback này sẽ được gọi khi việc khởi tạo server hoàn thành. Trong đó có hai tham số, req (request) là đối tượng mà nó nhận được từ client (trình duyệt chẳng hạn), còn res (response) là đối tượng mà server sẽ trả về cho trình duyệt. Chúng ta có thể làm bất cứ điều gì với hai đối tượng này. Như trong ví dụ trên, mình đơn giản chỉ là in ra log đối tượng req và trả lại trình duyệt một dòng thông báo: "VNTALKING: Xin chào Node.js".

```
server.listen(3000)
```

Mỗi một server đều phải lắng một port bất kỳ trong dải từ 1 -> 65535. Bạn có thể tùy chọn thoải mái. Tất nhiên nên trừ những port thông dụng đang được sử dụng bởi hệ thống như: 80, 22, 23, 25... Nếu bạn hỏi port là gì thì mình giải thích đơn giản là port là một gateway kết nối ra ngoài hoặc giữa các ứng dụng trên server được sử dụng bởi một ứng dụng cụ thể. Nếu nhiều ứng dụng cùng chạy trên server thì mỗi ứng dụng sẽ sử dụng một port khác nhau.

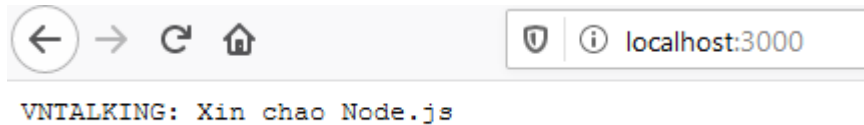
Trong ví dụ này, ứng dụng sẽ sử dụng port 3000 và mọi request tới port 3000 đều sẽ trả về dòng thông báo "VNTALKING: Xin chào Node.js "

Cách chạy ứng dụng

Để thực thi ứng dụng, chúng ta mở cửa sổ lệnh (trong ubuntu gọi là terminal, còn trong window gọi command Prompt), di chuyển con trỏ tới thư mục mã nguồn

```
Microsoft Windows [Version 6.1.7600]  
Copyright (c) 2009 Microsoft Corporation. All rights reserved.  
E:\github\nodejs-express-mongodb-co-ban>node index.js
```

Giờ bạn mở trình duyệt và truy cập vào địa chỉ `http://localhost:3000` để xem kết quả.



Bởi vì chúng ta đang chạy ứng dụng trên chính máy tính của mình nên chỉ có thể truy cập qua localhost. Phần cuối cuốn sách, mình sẽ hướng dẫn các bạn triển khai ứng dụng để publish lên internet, và bất kỳ ai, bất kỳ đâu cũng có thể truy cập được.

Hiểu hơn về request và response

Ứng dụng của chúng ta hiện tại mới chỉ phản hồi dòng chữ "VNTALKING: Xin chào Node.js" cho client, cho dù đằng sau URL:localhost:3000/xxx là gì đi nữa. Để có những xử lý phản hồi khác nhau cho những URL khác nhau, chúng ta cần có thể xử lý đơn giản như sau:

```
const server = http.createServer((req, res) => {
  if (req.url === '/about')
    res.end('The about page')
  else if (req.url === '/contact')
    res.end('The contact page')
  else if (req.url === '/')
    res.end('The home page')
  else {
    res.writeHead(404)
    res.end('page not found')
  }
})

server.listen(3000)
```

Giờ bạn chạy lại server và thử trên trình duyệt những URL sau:

- <http://localhost:3000>
- <http://localhost:3000/about>
- <http://localhost:3000/contact>

Tuy nhiên, đây chỉ là mình làm ví dụ đơn giản để bạn hiểu hơn về request và response thôi. Còn sau này, với dự án thực tế thì người ta không dùng kiểu *if-else* thế này đâu. Lúc đó bạn sẽ gặp khái niệm Router. Sử dụng router để điều hướng trang web.

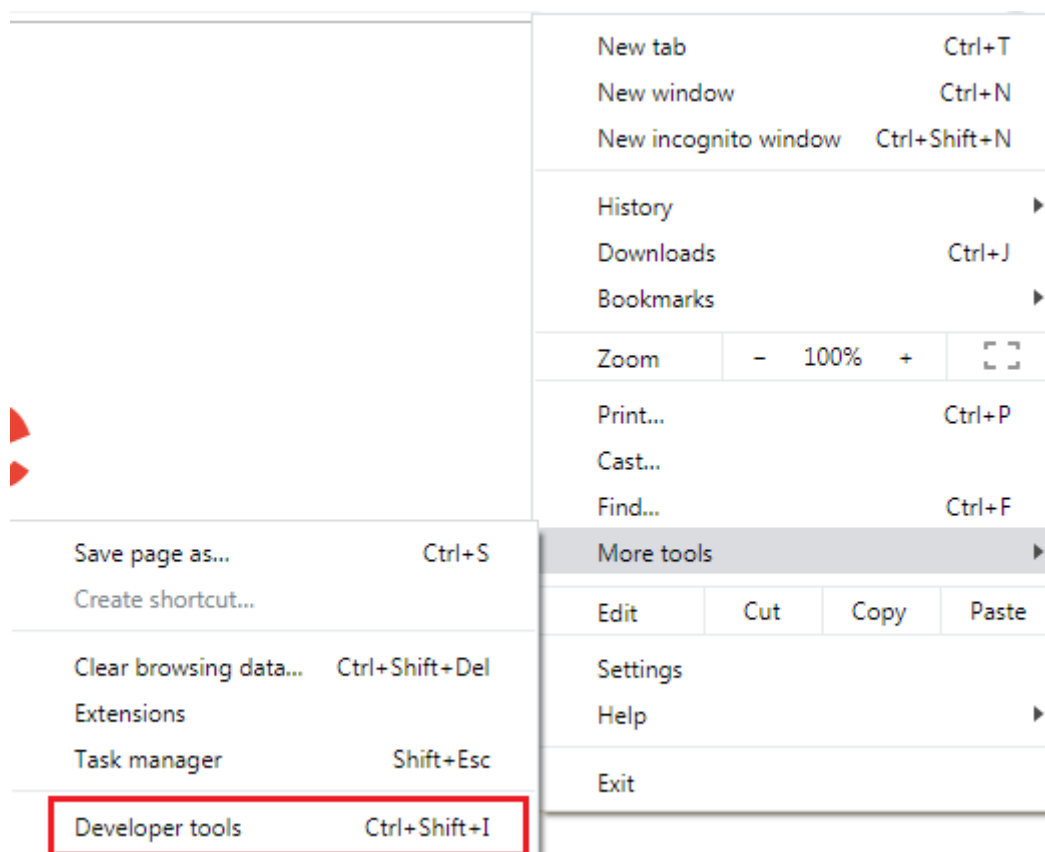
Tất nhiên, chúng ta sẽ tìm hiểu router ở phần sau của cuốn sách.

Mình muốn rõ hơn về đoạn code này:

```
res.writeHead(404)
```

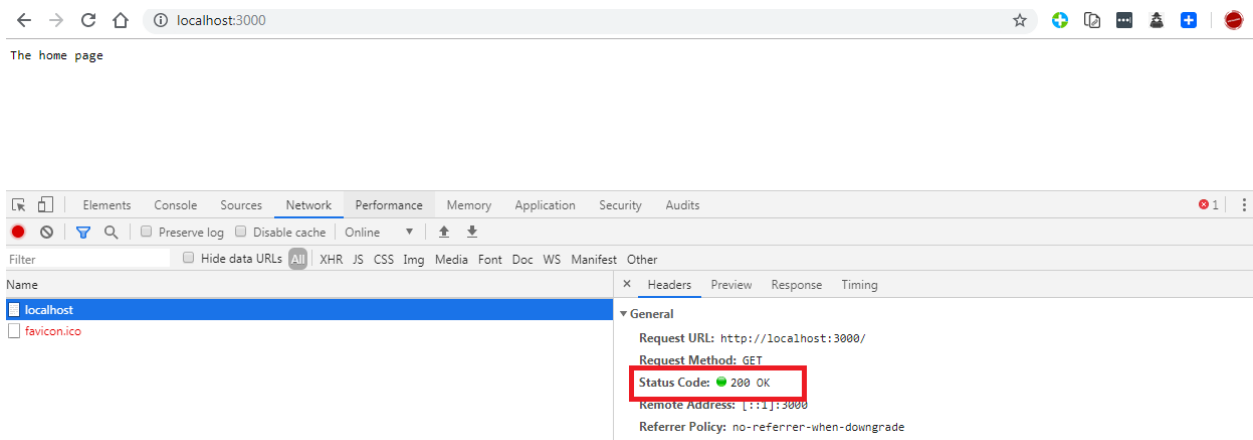
Khi có một request tới server và server phản hồi lại request đó. Với request valid, server sẽ phản hồi mã 200-OK. Còn nếu gặp lỗi thì tùy vào kiểu lỗi mà trả về mã lỗi tương ứng. Mã lỗi được quy định theo chuẩn của HTTP. Các bạn có thể tham khảo [ở đây](#).

Các bạn có thể kiểm tra mã trả về từ server bằng cách: Mở trình duyệt (Chrome chẳng hạn), tìm đến Developer tools như hình vẽ. Sau đó chọn tab Network



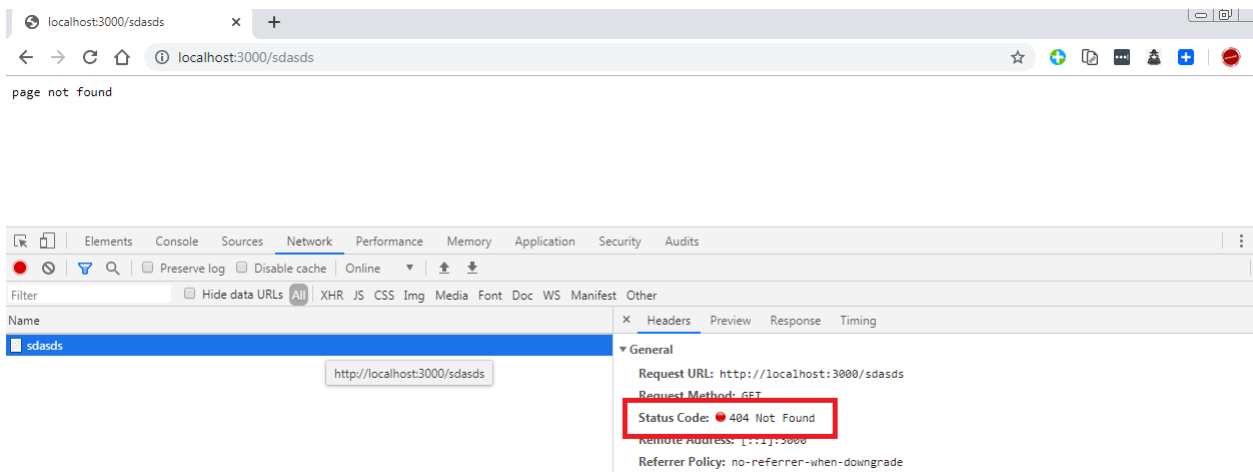
Hình 1.4: Developer tools trong trình duyệt Chrome

Sau đó bạn truy cập vào một URL chẳng hạn: `http://localhost:3000`



Hình 1.5: Kiểm tra status code 200 - OK

Còn nếu bạn truy cập một URL không tồn tại, như đoạn code trên thì sẽ trả về lỗi 404



Hình 1.6: Status 404 - Not Found

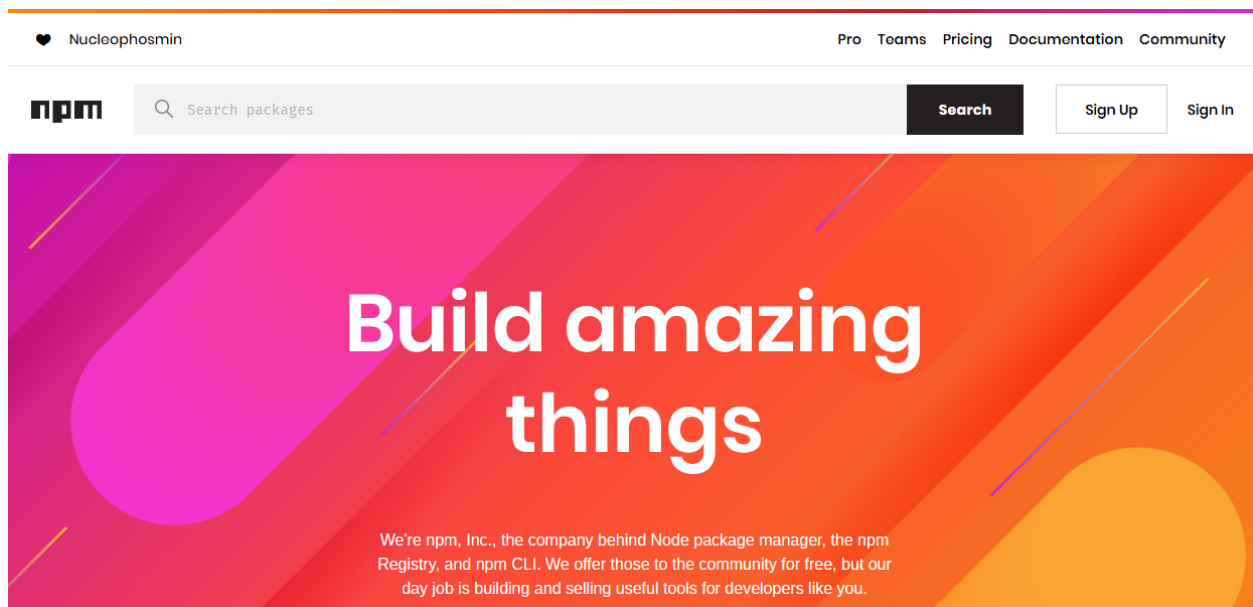
Tóm lại, với đoạn code trên, chúng ta đang viết một hàm Javascript đơn lẻ để lắng nghe những request từ client như browser, mobile hay bất kể client nào khác sử dụng API. Chúng ta gọi hàm này là một request handler (xử lý request). Cứ khi có request đến, hàm này sẽ tìm kiếm và xử lý rồi trả kết quả về cho client.

Tất cả ứng dụng Node.js đều hoạt động theo cơ chế như vậy. Một single request handler để xử lý tất cả các request và phản hồi các request đó. Với các ứng dụng nhỏ thì điều này có vẻ đơn giản. Nhưng với ứng dụng lớn, phải xử lý rất nhiều request, chưa kể để có kết quả phản hồi thì cũng mất nhiều thời gian như: render HTML, upload/download ảnh... Để giải quyết những vấn đề này chúng ta cần sự trợ giúp của công cụ. Phần tiếp theo của cuốn sách, chúng ta sẽ cùng nhau xem ExpressJS giải quyết vấn đề này như thế nào.

Giới thiệu về NPM và Express

Trong đoạn code minh họa của phần 1, chúng ta hoàn toàn sử dụng những package build-in của Node.js, ví dụ: http, fs... Trong khi đó, có rất nhiều packages được phát triển bởi các nhà phát triển bên thứ ba mà bạn có thể sử dụng cho dự án của bạn. Những packages này được lưu trữ trên website npmjs.com (hình 2.1). Trang web npmjs.com có thể hiểu như một market các packages, nơi mà bạn có thể tha hồ lựa chọn package nào phù hợp với dự án để sử dụng.

Còn NPM (Node Package Manager) là phần mềm được cài đặt cùng với Node.js, nó được dùng để quản lý các packages mà bạn download từ npmjs.com.



Hình 2.1: giao diện trang web npmjs.com

Cài đặt Custom Package với NPM

Để cài đặt một package từ npmjs.com, bạn tìm kiếm package từ hộp search, sau khi chọn được một package có vẻ ưng ý. Bạn chọn package đó và nhìn các thông tin một package như hình bên dưới.

The screenshot shows the npm page for the package 'easy-password-gen'. At the top, it indicates version 2.0.0, public status, and a publication date of 2 months ago. Navigation links include 'Readme', 'Explore' (with a BETA tag), '1 Dependency', '0 Dependents', and '2 Versions'. The package name 'easy-password-gen' is prominently displayed. Below it, a row of badges shows 'npm v2.0.0', 'downloads 17/month', 'readme style standard', 'license MIT', 'PRs welcome', and a 'Generate a random password' button. A 'Table of Contents' lists 'Usage', 'Install', 'Contribute', and 'License'. On the right, an 'Install' section features a red-bordered box with the command `> npm i easy-password-gen`, pointed to by a red arrow. Below this is a 'Weekly Downloads' line graph showing a peak at 5. A table lists package details: Version 2.0.0, License MIT, Unpacked Size 12.8 kB, and Total Files 17.

| Version | License |
|---------|---------|
| 2.0.0 | MIT |

| Unpacked Size | Total Files |
|---------------|-------------|
| 12.8 kB | 17 |

Hình 2.2: Trang thông tin một module trên npmjs.com

Ở đây nhìn vào thông tin cách cài đặt, họ có ghi rõ lệnh cài đặt package này bằng npm: Ví dụ lệnh cài đặt package easy-password-gen như hình 2.2

```
npm i easy-password-gen
```

Trong cuốn sách này, mình giới thiệu với các bạn một package quan trọng và cũng rất phổ biến trong thế giới Node.js, đó là ExpressJS. Bởi vì cú pháp API của Node.js có thể dài dòng, khó hiểu và giới hạn về tính năng, nên người ta bắt đầu nghĩ tới một framework giúp họ đơn giản hóa quá trình viết ứng dụng.

ExpressJS là một Framework nhỏ, nhưng linh hoạt được xây dựng trên nền tảng của Nodejs. Nó cung cấp các tính năng mạnh mẽ để phát triển ứng dụng web. Chúng ta sẽ thấy ExpressJS làm đơn giản hóa các API của Node.js, cách tổ chức dự án theo mô hình MVC với middleware và routing. Ngoài ra còn hàng tá những hàm Utils rất hữu ích để xử lý HTTP và render HTML.

Để cài đặt ExpressJS, các bạn gõ:

```
npm install express
```

Trước khi bạn có thể chạy câu lệnh cài đặt express như trên, dự án của bạn cần phải tạo trước *package.json*. Đây là file cấu hình của dự án, lưu trữ tất cả các thông tin về dự án như: tên dự án, version, các packages được sử dụng trong dự án... Nếu bạn chạy lệnh `npm install express` mà không có *package.json* thì sẽ gặp lỗi: **"no such file or directory, open ... package.json"**.

Để generate ra file *package.json*, bạn di chuyển con trỏ vào thư mục dự án, gõ lệnh:

```
npm init
```

Sau đó bạn điền các thông tin như trong hướng dẫn của trình thuật sĩ. Sau khi hoàn thành, bạn gõ "Yes" để xác nhận tạo *package.json*

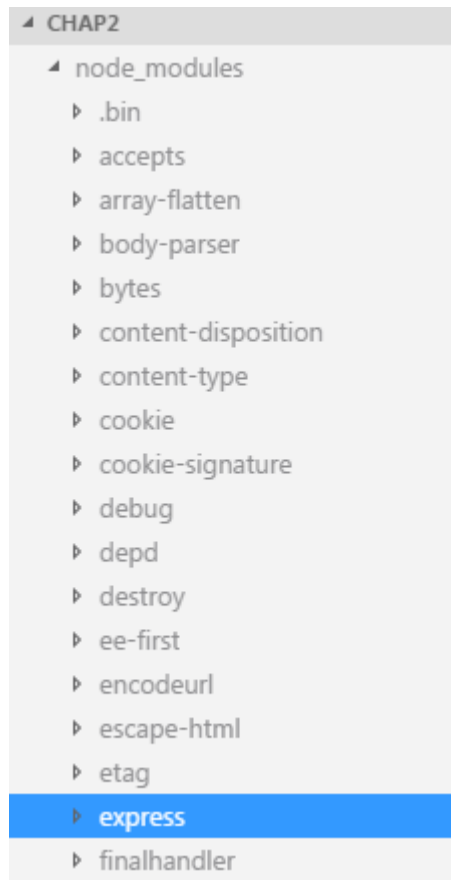
Ví dụ đây là *package.json* mà mình tạo cho dự án cho phần 2 của cuốn sách này.

```
{
  "name": "chap2",
  "version": "1.0.0",
  "description": "Code minh họa cho chap 2",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "VNTALKING.COM",
  "license": "ISC"
}
```

Tiếp theo, bạn gõ lệnh cài đặt express: *npm install express*. Khi việc cài đặt kết thúc, bạn sẽ thấy *package.json* thêm một dependencies là *express* như bên dưới.

```
{
  "name": "chap2",
  "version": "1.0.0",
  "description": "Code minh họa cho chap 2",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "VNTALKING.COM",
  "license": "ISC",
  "dependencies": {
    "express": "^4.17.1"
  }
}
```

Thẻ *dependencies* sẽ chứa tất cả tên của các packages cùng với version lúc cài đặt. Tại thời điểm của cuốn sách này thì version mới nhất của express là 4.17.1. Ngoài ra, toàn bộ mã nguồn của package express sau khi được cài đặt sẽ được lưu tại thư mục *node_modules* (Hình 2.3)



Hình 2.3: nơi chứa mã nguồn expressJS trong dự án

Bạn mở thư mục *node_modules* thì thấy ngoài thư mục có tên *express* thì còn có rất nhiều thư mục khác nữa. Mặc dù lúc trước bạn mới chỉ cài mỗi một mình ExpressJS. Nguyên nhân là do ExpressJS có sử dụng các dependencies khác, nên khi cài đặt ExpressJS thì npm cũng sẽ tải và cài đặt chúng luôn. Để biết được *express* có những dependencies nào thì bạn lại vào package của ExpressJS là sẽ thấy.

Tuy nhiên, bạn không nên sửa bất kì dòng code trong thư mục *node_modules*. Vì đây là mã nguồn gốc của package, bạn sẽ không biết được sẽ phát sinh lỗi gì nếu sửa chúng đâu.

Giới thiệu Express

Ở phần này, mình sẽ giới thiệu nhiều hơn về ExpressJS, tại sao nó lại giúp phát triển ứng dụng Node.js đơn giản và nhanh hơn.

Đầu tiên, chúng ta sẽ import *express* vào ứng dụng bằng cách thêm đoạn code sau vào *index.js*:

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

Để có thể so sánh và kết luận tại sao ExpressJS lại giúp phát triển ứng dụng nhanh hơn, chúng ta sẽ viết lại những gì ở phần 1 cuốn sách, nhưng với sự trợ giúp của ExpressJS.

```
const http = require('http');

const server = http.createServer((request, response) => {
  console.log("xin chào VNTALKING.COM");
})

server.listen(3000)
```

Nhưng với ExpressJS thì bạn chỉ cần như sau:

```
const express = require('express')
const app = express()
app.listen(3000, () =>{
  console.log("App listening on port 3000")
})
```

Bạn thấy đấy, chúng ta không cần phải import thêm bất cứ package nào cả, cũng không cần phải viết đoạn xử lý request hay response nữa. Về bản chất thì ExpressJS lo làm giúp việc import các package cần thiết cho chúng ta rồi, chúng ta không cần phải làm nữa. Tuy nhiên, nếu chỉ đơn giản là rút gọn code thì ExpressJS lại quá bình thường phải không? ExpressJS không đơn giản như vậy đâu.

Xử lý request với Express

ExpressJS cho phép xử lý linh hoạt hơn với các request "GET" hay "POST" từ trình duyệt. Dưới đây là minh họa cách ứng dụng trả về một đối tượng JSON khi nhận được một request:

```
const express = require('express')
const app = express()
app.listen(3000, () =>{
  console.log("App listening on port 3000")
})

app.get("/", (request, response) =>{
  response.json({
    name: "Duong Anh Son",
    website: "VNTALKING.COM"
  })
})
```

Trong đoạn code trên, bạn có thể thấy là ExpressJS cung cấp nhiều lựa chọn hơn để bạn có thể phản hồi lại những request từ trình duyệt. Ví dụ, chúng ta trả về đối tượng JSON cho trình duyệt bằng hàm `response.json()`. Nhờ điều này mà chúng ta dễ dàng xây dựng các REST API với Node.js.

Ngoài ra, chúng ta có thể định nghĩa những routes cụ thể, ví dụ như:

```
app.get("/about", (request, response) =>{
  response.json({
    name: "Duong Anh Son",
    website: "VNTALKING.COM"
  })
})
```

Cái này người ta gọi là Routing. Tức hiểu nôm na là chúng ta quy ước phần xử lý với một URL cụ thể nào đó. Như ở phần 1, để xử lý những URL riêng biệt, chúng ta phải nhờ đến *if-else* trong một hàm xử lý request lớn.

```
const server = http.createServer((req, res) => {
  if (req.url === '/about')
    res.end('The about page')
  else if (req.url === '/contact')
    res.end('The contact page')
  else if (req.url === '/')
    res.end('The home page')
  else {
    res.writeHead(404)
    res.end('page not found')
  }
})
```

Với ExpressJS thì bạn có thể tách riêng biệt cho từng URL riêng biệt. Điều này giúp tăng khả năng module hóa hay khả năng bảo trì dự án.

Với đoạn code thì express có thể tách như sau:

```
app.get("/about", (request, response) =>{
  res.send('The about page')
})

app.get("/contact", (req, res) =>{
  res.send('The contact page')
})

app.get("/contact", (req, res) =>{
```

```

    res.send('The contact page')
  })

  app.get("/", (req, res) =>{
    res.send('The home page')
  })

  app.get('*', function(req, res){
    res.header(404)
    res.send('page not found')
  });

```

Bất đồng bộ với Call Back

Trong những đoạn code trên, bạn đã bắt gặp việc sử dụng các hàm call back. Call Back là một khái niệm vô cùng quan trọng trong Node.js, được sử dụng để xử lý các hàm bất đồng bộ.

Đồng bộ tức là các hàm viết sau sẽ đợi hàm viết trước hoàn thành. Ví dụ như trong PHP thì các hàm sẽ lần lượt thực hiện:

Task 1 -> Task 2 -> Task 3 -> Task 4 -> Completion

Như vậy, nếu task 1 thực hiện quá lâu hoặc bị block vì lý do nào đó thì hàm task sẽ không thể thực hiện. Với Node.js thì hơi khác, nó cho phép các hàm được thực hiện bất đồng bộ, tức là hàm sau không cần phải đợi hàm trước thực hiện xong. Tất cả sẽ được cho vào một queue, cái nào xong trước thì thông báo.

Ví dụ đoạn code sau trong Node.js

```

//Task1
app.get('/', (req, res) => {
  // query database
})
//Task2
app.get('/about', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'about.html'))
})

```

Khi bạn chạy chương trình, cả 2 task đều sẽ được thực hiện cùng một thời điểm. Nếu task 1 mà cần nhiều thời gian, mà task 2 lại đơn giản chỉ gửi một file tĩnh cho trình duyệt thì khả năng task sẽ thực hiện xong trước. Cả 2 task cùng bắt đầu một thời điểm nhưng không có nghĩa chúng sẽ thực hiện đồng thời. Khi một task cần thời gian thì task sẽ được thực hiện. Và khi một task thực hiện xong thì sử dụng call back để thông báo cho chương trình biết.

Về lý thuyết thì chương trình bất đồng bộ sẽ chạy nhanh hơn vì tận dụng được thời gian nhàn rỗi của CPU.

Trả về một file html cho client

Để phản hồi lại client một file html trong ExpressJS, bạn sử dụng `sendFile` api. Cách làm như sau:

```
// called when request to '/about' comes in
app.get('/about', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'about.html'))
})

//called when request to '/contact' comes
app.get('/contact', (req, res) => {
  res.sendFile(path.resolve(__dirname, 'contact.html'))
})
```

Như ví dụ trên, máy chủ node.js sẽ trả về cho client tệp about.html khi client request tới URL: `"/about"`



Trong các ví dụ vừa qua mình sử dụng hàm `app.get(...)` để xử lý các yêu cầu HTTP GET. GET là loại request được quy ước dùng để truy cập vào server để lấy thông tin, tài nguyên, hình ảnh... mà không thay đổi nội dung phía server. Ngoài GET ra, HTTP còn định nghĩa các loại request khác như POST, DELETE, UPDATE. Chúng ta sẽ tìm hiểu chúng dần dần trong các phần sau của cuốn sách.

Trả về static resource (image, css, js...) cho client

Với các ứng dụng web, việc phải sử dụng image, css, js là điều gần như bắt buộc để định dạng, trang trí giao diện cho ứng dụng. Nhưng mặc định thì tất cả các tài nguyên trên máy chủ đều được bảo mật, nếu không có quyền thì client không thể truy cập được, hoặc ít nhất phải truy cập qua các router.

Có một giải pháp phổ biến là chúng ta sẽ đưa hết các file static (image, css, js...) vào một thư mục, gọi là public. Thư mục public có thể truy cập từ bất kỳ đâu, ai cũng có thể truy cập được, chỉ biết đường dẫn của nó.

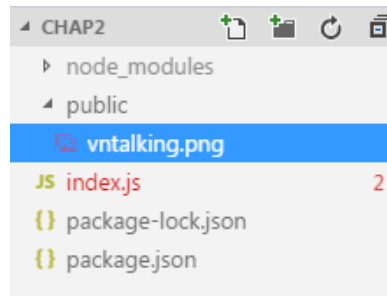
Trong Express, bạn dễ dàng định nghĩa một thư mục public như thế.

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

```
app.use(express.static('public'))

app.listen(3000, () => {
  console.log("App listening on port 3000")
})
```

Giờ làm ví dụ minh họa, mình đưa một tệp ảnh vntalking.png vào thư mục public như dưới đây.



Hình 2.4: Thư mục public dùng để chứa static resource

Giờ bạn có thể truy cập trực tiếp trên trình duyệt `http://localhost:3000/vntalking.png`. Bạn thấy đấy, chúng ta không cần phải định nghĩa router để truy cập vào file ảnh đó.

Bạn có thể sử dụng các tệp trong thư mục public trong các file html mà không phải thông qua router.

```
//about.html
<h1>Đây là màn hình thông tin về VNTALKING.COM</h1>

```

Và thử vào trình duyệt kiểm tra thử.



Hình 2.5: Minh họa truy cập static resource

Tổng kết

Qua phần này, chúng ta đã biết cách cài đặt thêm package từ bên ngoài thông qua npm. Biết về một framework cực kỳ mạnh mẽ ExpressJS, nó giúp chúng ta xử lý các request và phản hồi cho client cực dễ dàng. Ngoài ra, bạn cũng biết rõ hơn về cấu trúc tệp cấu hình dự án *package.json*, khi bạn cài đặt thêm một package thì sẽ có thêm trong thẻ dependencies để dễ quản lý hơn. Cuối cùng, bạn đã biết cách tạo thư mục public để lưu trữ các tệp static như css, js, image... phục vụ cho viết thiết kế giao diện ứng dụng.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap2>

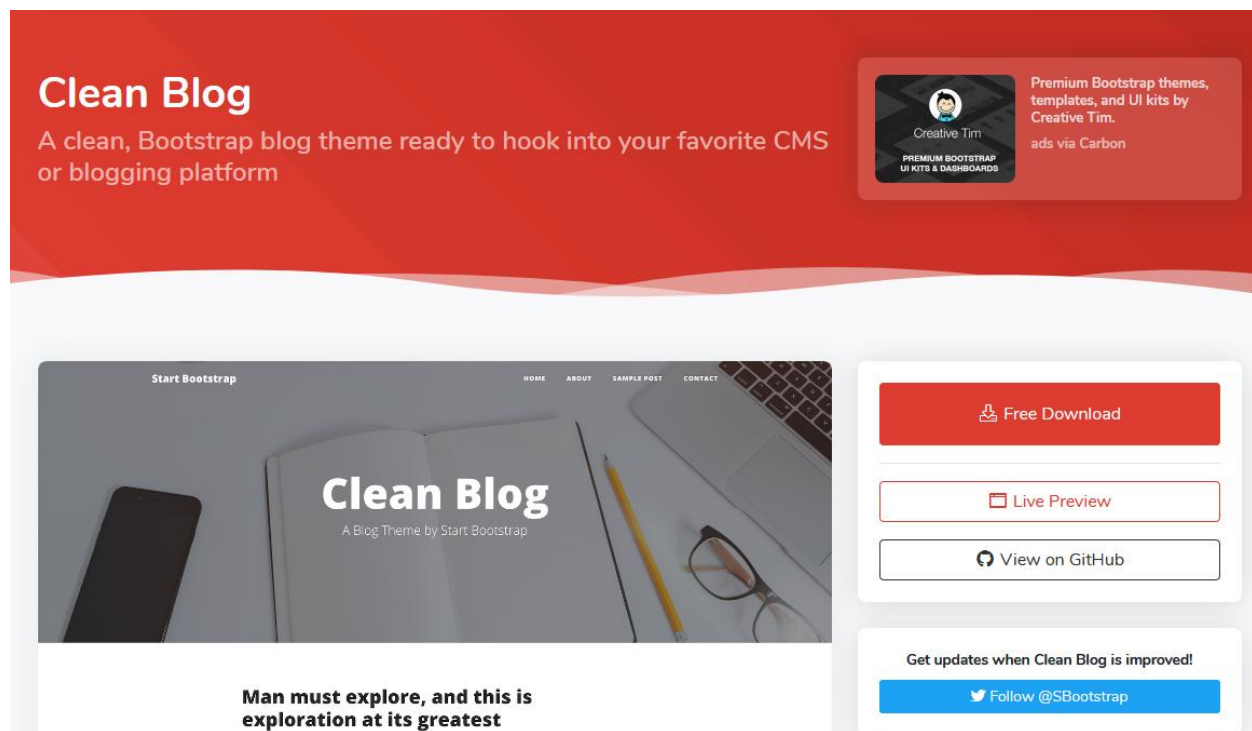
Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Bước đầu tạo web app với Express

Vì mục tiêu của cuốn sách này không tập trung vào hướng dẫn các bạn thiết kế website bằng HTML, CSS. Do vậy, chúng ta sẽ sử dụng một template có sẵn để thực hành Node.js + ExpressJS. Để tìm một template, bạn vào trang startbootstrap.com và lựa chọn một template ưng ý nhất.

Tải web template

Trong cuốn sách này, mình chọn clean blog template - Hình 3.1 (<https://startbootstrap.com/themes/clean-blog/>)



Hình 3.1 Giao diện một web template

Bạn tải miễn phí template này về, và giải nén vào một thư mục trong máy tính.

Bước tiếp theo, chúng ta sẽ khởi tạo dự án Node.js mới bằng câu lệnh: `npm init`

(Bạn cứ trả lời "Yes" cho các câu hỏi mà NPM hỏi để tạo ra *package.json*). Tiếp theo thì mình cài đặt express cho dự án.

```
$ npm install express
```

Ok, giờ bạn tạo file *index.js* trong thư mục gốc của dự án, và viết đoạn code tạo server đơn giản như sau (Nếu bạn quên thì quay lại đọc phần 2 - mình đã giới thiệu cách tạo server đơn giản):

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

const app = new express()
app.listen(4000, () => {
  console.log('App listening on port 4000')
})
```

Để chạy server, bạn gõ lệnh sau vào terminal:

```
$ node index.js
```

Tự động khởi động server mỗi khi thay đổi mã nguồn

Mặc định, để kiểm tra kết quả mỗi khi bạn thay đổi mã nguồn, bạn cần phải stop/start lại server, có như vậy đoạn code mới có hiệu lực. Nếu bạn muốn công đoạn này được thực hiện tự động, chỉ cần mỗi khi mã nguồn có thay đổi thì server sẽ tự động khởi động lại. Để làm được điều này, chúng ta cần phải cài đặt thêm thư viện theo dõi mã nguồn. Có nhiều thư viện làm điều như: [forever](#), [nodemon](#)...

Trong cuốn sách này, chúng ta sẽ sử dụng nodemon. Để cài đặt nodemon cho dự án, từ terminal:

```
$ npm install nodemon --save-dev
```

Để mình giải thích qua một chút về các tham số trong câu lệnh trên:

- **Tham số --save:** mục đích là sau khi cài đặt xong vào thư mục node_modules thì cũng thêm thông tin vào mục dependencies trong *package.json*. Với việc thêm vào dependencies, sau này người khác muốn cài đặt các thư viện cần thiết cho dự án thì cần gõ: `npm install` là nó tự cài đặt tất cả dependencies được liệt kê trong *package.json*
- **Tham số -dev:** Có những thư viện chỉ dùng để hỗ trợ phát triển dự án, còn khi triển khai thật ra thị trường thì không cần. Đó là lý do mà phần dependencies của *package.json* được chia làm 2 phần: "dependencies" và "devDependencies". Tham số -dev là để thêm vào "devDependencies".

Npm start

Ở phần trên của cuốn sách, mình có hướng dẫn các bạn khởi động server bằng lệnh:
`node <tên_file_js>`

Tuy nhiên, với cách làm này, nếu bạn cần phải tùy biến lệnh khởi động server thì sẽ rất khó nhớ lệnh cho những lần thực hiện sau. Do đó, để đơn giản và rút gọn câu lệnh, chúng ta có thể sử dụng câu lệnh: `npm start`.

Các bạn mở `package.json`, rồi tìm đến thẻ `scripts` và sửa như sau:

```
"scripts": {  
  "start": "nodemon index.js"  
}
```

Từ giờ, mỗi lần khởi động server, bạn chỉ cần gõ lệnh:

```
$ npm start
```

Với cách làm này, bạn sẽ không còn phải quan tâm file chính của dự án là file nào, `index.js` hay `app.js`... chỉ đơn giản gõ `npm start` là được.

Tạo thư mục public chứa tệp static

Như mình đã giải thích ở phần trước, các tệp static được hiểu là những tệp tạo giao diện, tương tác trực tiếp với người dùng, truy xuất không cần qua router. Những tệp này gồm: ảnh, css, html, js.

Về lý thuyết thì mình đã hướng dẫn ở phần trước cuốn sách rồi, giờ chúng ta bắt tay vào thực hành tạo những tệp static cho blog dự án của chúng ta.

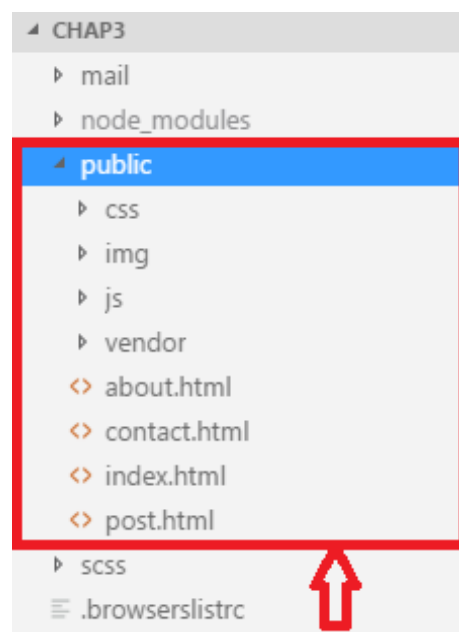
```
const express = require('express')  
const app = new express()  
  
//Đăng ký thư mục public.  
app.use(express.static('public'))  
  
app.listen(4000, () => {  
  console.log('App listening on port 4000')  
})
```

Bởi vì chúng ta đang sử dụng template có sẵn nên đã có những tệp html trước rồi, không cần phải tạo nữa. Giờ bạn chỉ cần chuyển những tệp html vào thư mục *public* này. Hiện tại, trong template có những tệp html sau sẽ được chuyển vào thư mục *public*.

index.html
about.html
contact.html
post.html

Tiếp theo, chúng ta sẽ chuyển tất cả những thư mục *css*, *img*, *js*, *vendor* vào trong thư mục *public*.

OK, giờ bạn sẽ thấy cấu trúc thư mục dự án mới sẽ như này.



Hình 3.2: thư mục *public* trong dự án

Bạn thử mở một tệp html lên xem mã nguồn, bạn sẽ biết tại sao chúng ta lại chuyển những thư mục *css*, *js*... vào thư mục *public*.

Ví dụ, với tệp *index.html*, bạn có thấy các thẻ link tới thư mục *vendor*, *css*... (mình có bôi đậm trong code đó).

```
<head>  
<title>Clean Blog - Start Bootstrap Theme</title>
```

```
<!-- Bootstrap core CSS -->
```

```
<link href="vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">

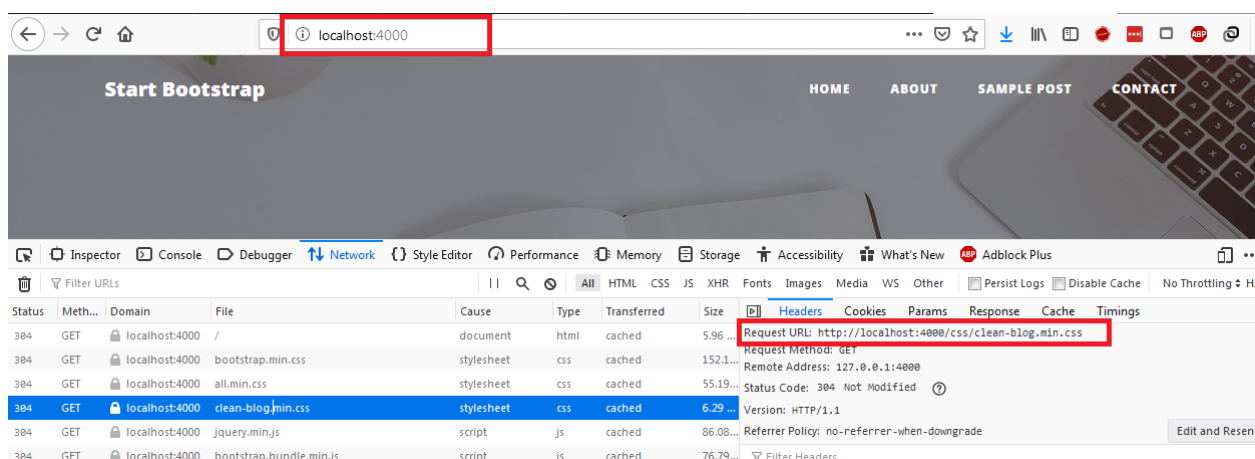
<!-- Custom fonts for this template -->
<link href="vendor/fontawesome-free/css/all.min.css" rel="stylesheet"
type="text/css">
...
<!-- Custom styles for this template -->
<link href="css/clean-blog.min.css" rel="stylesheet">

</head>
```

Với những kiểu tương đối như: **css/clean-blog.min.css** thì khi trình duyệt đọc tệp html này sẽ tự động nối thêm domain vào để tạo thành một link hoàn chỉnh: `http://<domain>/css/clean-blog.min.css`

Để mình chỉ có bạn. Giờ bạn chạy server bằng lệnh: `$ npm start`

Sau đó vào trình duyệt kiểm tra: `http://localhost:4000`



Hình 3.3: Kiểm tra link static resource trên trình duyệt

Nếu bạn click vào các link trên giao diện như: About, Same post, Contact, nó sẽ chuyển đến đúng trang đích. Tuy nhiên, hiện giờ trang web của chúng ta chỉ thuần là trang web tĩnh, các file html liên kết với nhau mà chưa hề xử lý gì cả.

Việc tiếp theo, chúng ta sẽ xử lý việc chuyển trang (about, sample post, contact) thông qua route.

Tạo Page routes

Như mình đã nói ở trên, hiện tại chúng ta mới chỉ đơn giản là phản hồi nguyên một tệp html xuống cho client từ thư mục public. Có một nhược điểm của thư mục public là ai cũng có thể truy cập được, server hầu như chỉ biết trả về cho client tệp mà nó yêu cầu, không thể xử lý gì được nữa.

Để có thể xử lý được logic trước khi trả về cho client, ví dụ như cần xác thực request xem request này có hợp lệ không... thì bắt buộc bạn phải sử dụng routes.

Để thực hành tạo routes, chúng ta tạo một thư mục đặt tên là pages. Sau đó lại copy tệp *index.html* vào đó.

Sau đó, chúng ta mở tệp *index.js* và tạo route như sau:

```
const express = require('express')
const app = new express()
const path = require('path')
...
app.get('/', (request, response) =>{
  response.sendFile(path.resolve(__dirname, 'pages/index.html'))
})
```

Giờ cứ có một request tới trang chủ "/", thì trang *index.html* sẽ được trả về.

Tổng kết

Như vậy, qua phần 3 của cuốn sách này, chúng ta đã bắt đầu tạo được một trang web từ một template có sẵn. Biết cách sử dụng nodemon để tự động phát hiện sự thay đổi trong mã nguồn để khởi động lại node server, giúp đoạn code thay đổi có hiệu lực. Cuối cùng thì bạn đã biết cách tạo route để giúp ứng dụng có thể chuyển qua lại giữa các trang (page).

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap3>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Templating Engine

Giới thiệu Template engine

Templating engines giúp bạn tạo các trang HTML một cách linh động hơn, thay vì fix cứng các giá trị thì giờ bạn có thể thêm logic xử lý chúng trước khi xuất ra HTML. Có một đặc điểm hữu ích khi sử dụng template engine là bạn có thể tái sử dụng được những đoạn code. Với những phần mà xuất hiện ở nhiều màn hình, chúng giống nhau thì bạn có thể tái sử dụng được, điều mà các tệp html thuần khó có thể làm được.

Có rất nhiều template engine cho bạn lựa chọn như: Pug, Handlebars, Jade, EJS... Trong cuốn sách này, mình sẽ chọn [EJS](#). Vì đơn giản đây là template engine phổ biến, được nhiều người sử dụng, đặc biệt là kết hợp với Express.

Khi bạn vào trang chủ của EJS, bạn sẽ thấy nó được giới thiệu là một templating language giúp generate ra HTML sử dụng thuần Javascript. Để bắt đầu với EJS trong dự án, chúng ta sử dụng cặp tag: `<%= ... %>`

Đầu tiên là phải cài đặt EJS đã:

```
$ npm install ejs --save
```

Để sử dụng EJS trong dự án, bạn cần khai báo nó với Express.

```
const express = require('express')
const app = new express()
...
const ejs = require('ejs')
app.set('view engine', 'ejs')
```

Thông qua api: `app.set('view engine', 'ejs')`, chúng ta thông báo cho ExpressJS biết template engine là EJS, rằng các file có đuôi là `.ejs` cần phải render bởi EJS package.

Như ở phần trước, chúng ta đã tạo route để xử lý và phản hồi cho client như sau:

```
app.get('/', (request, response) =>{
  response.sendFile(path.resolve(__dirname, 'pages/index.html'))})
```

Giờ với EJS thì chúng ta chuyển thành như sau:

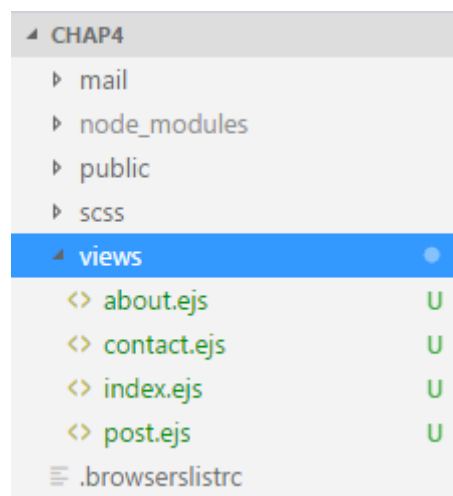
```
app.get('/', (request, response) =>{
  response.render('index')
})
```

Với đoạn code trên, ExpressJS sẽ tìm trong thư mục "views" của dự án xem có tệp *index.ejs* không? Nếu có thì generate ra HTML với sự trợ giúp của EJS engine.

Các bạn nhớ đổi phần mở rộng từ *html* -> *ejs*, vì đó mới là đuôi được hỗ trợ của EJS. Thực hiện tương tự với các trang about, contact, sample post.

```
app.get('/about', (req, res) => {
  //res.sendFile(path.resolve(__dirname, 'pages/about.html'))
  res.render('about');
})
app.get('/contact', (req, res) => {
  //res.sendFile(path.resolve(__dirname, 'pages/contact.html'))
  res.render('contact');
})
app.get('/post', (req, res) => {
  //res.sendFile(path.resolve(__dirname, 'pages/post.html'))
  res.render('post')
})
```

Sau các bạn đổi tên và chuyển vào thư mục views, giờ dự án sẽ trông như thế này (Hình 4.1)



Hình 4.1: thư mục chứa layout của web app

Layout

Để giải quyết vấn đề bị lặp code quá nhiều khi thiết kế giao diện bằng HTML. Ví dụ, bất kỳ trang nào cũng có các thành phần như header, footer... Nếu bạn làm HTML thuần bạn sẽ biết việc tái sử dụng code khó khăn thế nào. Với template engine xuất hiện khái niệm về layout file, giúp bạn tái sử dụng code HTML rất đơn giản.

Bạn sẽ tạo riêng các file giao diện dùng chung như: header, footer, navbar layout, scripts layout... Sau đó sẽ include chúng vào từng page riêng lẻ.

Ví dụ như trong `index.ejs`, sẽ có các thành phần có thể xuất hiện ở các màn hình khác như: `<head>`, `<nav>`, `<footer>`, `<script>`. Mục tiêu của chúng ta là sẽ tách riêng các thành phần này một file riêng: `header.ejs`, `footer.ejs`, `scripts.ejs`, `navbar.ejs`. Sau này bạn chỉ việc include chúng vào các page khác là được.

Chúng ta thực hành nhé. Đầu tiên, bạn thêm một thư mục *layouts* trong *views*.

Tiếp theo, bạn copy toàn bộ nội dung code trong thẻ `<head>` trong tệp `index.ejs` sang `header.ejs`.

```
<head>
<meta charset="utf-8">
<meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
<meta name="description" content="">
<meta name="author" content="">

<title>VNTALKING - Clean Blog</title>

<!-- Bootstrap core CSS -->
<link href="vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">

<!-- Custom fonts for this template -->
<link href="vendor/fontawesome-free/css/all.min.css" rel="stylesheet"
type="text/css">
<link
href='https://fonts.googleapis.com/css?family=Lora:400,700,400italic,700itali
c' rel='stylesheet' type='text/css'>
<link
href='https://fonts.googleapis.com/css?family=Open+Sans:300italic,400italic,6
00italic,700italic,800italic,400,300,600,700,800' rel='stylesheet'
type='text/css'>
```

```
<!-- Custom styles for this template -->
<link href="css/clean-blog.min.css" rel="stylesheet">

</head>
```

Thực hiện tương tự cho các tệp: *footer.ejs*, *scripts.ejs*, *navbar.ejs*

Bạn tham khảo kết quả toàn bộ việc tách file tại đây nhé:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap4/views/layouts>

Khi bạn đã tách thành các tệp common riêng thì giờ bạn phải include chúng vào các page. Sử dụng lệnh include. Ví dụ include tệp header: `<%- include('layouts/header'); -%>`

Giờ tệp *index.ejs* sẽ như sau (bạn chú ý phần mình bôi đậm nhé):

```
<html lang="en">
<%- include('layouts/header'); -%>
<body>
<%- include('layouts/navbar'); -%>
<!-- Page Header -->
<header class="masthead" style="background-image: url('img/home-bg.jpg')">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="site-heading">
          <h1>Clean Blog</h1>
          <span class="subheading">A Blog Theme by Start Bootstrap </span>
        </div>
      </div>
    </div>
  </div>
</header>

<!-- Main Content -->
...
<hr>
<%- include('layouts/footer'); -%>
<%- include('layouts/scripts'); -%>

</body>
</html>
```

Tổng kết

Như vậy, qua phần này chúng ta đã refactoring lại mã nguồn, sử dụng tempate engine để có thể tái sử dụng code nhiều nhất code thể. Nhờ đó mã nguồn của chúng nhìn gọn gàng hơn, dễ maintaince hơn.

Để có thể generate động HTML từ tempate engine, trong node.js bạn gọi hàm `res.render('...');`

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap4>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Giới thiệu MongoDB

Với Node.js, bạn có thể sử dụng bất kỳ loại cơ sở dữ liệu(CSDL) nào, từ SQL tới NoSQL. Với SQL, người ta hay sử dụng Postgresql, còn với NoSQL thì phổ biến nhất là MongoDB. Việc lựa chọn loại cơ sở dữ liệu tùy thuộc vào yêu cầu bài toán của dự án, yêu cầu của khách hàng... Vì mỗi loại cơ sở dữ liệu lại ưu và nhược điểm riêng.

Trong cuốn sách này, mình sẽ sử dụng MongoDB làm hệ quản trị cơ sở dữ liệu cho ứng dụng.

MongoDB là một hệ quản trị cơ sở dữ liệu mã nguồn mở, quản lý dữ liệu kiểu NoSQL và được hàng triệu người sử dụng. Trước khi nói về NoSQL, chúng ta sẽ nói qua về các loại cơ sở dữ liệu quan hệ để các bạn có cái nhìn toàn diện hơn. Với CSDL quan hệ chúng ta có khái niệm bảng, các cơ sở dữ liệu kiểu quan hệ (như MySQL hay SQL Server...) sử dụng các bảng để lưu dữ liệu.

NoSQL là 1 dạng CSDL mã nguồn mở, là viết tắt của: None-Relational SQL hay có nơi thường gọi là Not-Only SQL.NoSQL ra đời như là 1 mảnh vá cho những khuyết điểm và thiếu sót cũng như hạn chế của mô hình dữ liệu quan hệ RDBMS (Relational Database Management System - Hệ quản trị cơ sở dữ liệu quan hệ) về tốc độ, tính năng, khả năng mở rộng,...NoSQL bỏ qua tính toàn vẹn của dữ liệu và transaction để đổi lấy hiệu suất nhanh và khả năng mở rộng.

Có nhiều phần mềm quản trị cơ sở dữ liệu dạng NoSQL như MongoDB, Azure Cosmos DB, Cloud Bigtable (Google), Cassandra ...

Nhưng tại sao lại chọn MongoDB?

Thứ nhất là MongoDB rất phổ biến, điều đó có nghĩa là khi bạn có vấn đề thì với cộng đồng đông đảo sẵn sàng giúp đỡ bạn.

Thứ 2, MongoDB là một trong những hệ quản trị cơ sở dữ liệu NoSQL ra đời rất sớm, có nhiều công ty lớn cũng sử dụng như: eBay, Craigslist hay Orange... Điều này cho chúng ta thấy tin tưởng hơn vào giải pháp mà MongoDB mang lại.

Kiến trúc của MongoDB

Như mình đã nói ở trên, kiến trúc của MongoDB là NoSQL, thông tin được lưu trữ trong document kiểu JSON thay vì dạng bảng như CSDL quan hệ.

Trong MongoDB, chúng ta có khái niệm *Collection*, một *collection* sẽ tương ứng với khái niệm table trong CSDL quan hệ.

Collection sẽ chứa *documents*. Mỗi một document sẽ tương ứng với một record, được biểu diễn dưới dạng JSON.

Ví dụ: một sản phẩm (sách chẳng hạn) có những thông tin như: tên sách, ảnh bìa, giá. Mỗi thông tin sẽ được biểu diễn theo cặp *key-value*. Dưới đây là một ví dụ minh họa cho một cuốn sách được lưu trữ trong MongoDB.

Database

→ Books collection

→ book document

```
{  
  price: 100000,  
  name: "Lập trình Node.js thật là đơn giản",  
  image: "https://vntalking.com/lap_trinh_nodejs.png"
```

```
},
```

→ book document

...

Cài đặt MongoDB

Có nhiều cách để cài đặt MongoDB, bạn có thể tìm trên Google với từ khóa “install MongoDB” là ra hàng loạt cách. Bạn có thể tự chọn một cách mà bạn thích. Vì MongoDB hỗ trợ đa nền tảng nên tùy vào máy tính của bạn sử dụng HĐH nào mà chọn phiên bản MongoDB tương ứng.



Khi bạn vào trang chủ MongoDB, bạn sẽ được họ gợi ý sử dụng dịch vụ MongoDB – Cloud, tức là bạn không cần phải cài đặt MongoDB trên máy tính mà sử dụng luôn dịch vụ của họ. Tuy nhiên, mình muốn các bạn làm quen với việc chạy MongoDB trên máy chủ của mình để sau này còn tự triển khai, không bị phụ thuộc vào dịch vụ của họ.

Trong cuốn sách này mình sẽ hướng dẫn các bạn cài đặt MongoDB trên máy tính Ubuntu (với các OS khác, các bạn tham khảo cách cài trên trang chủ MongoDB).

Đầu tiên, bạn vào link dưới, đây là tài liệu hướng dẫn cài đặt chính chủ của MongoDB:

<https://docs.mongodb.com/manual/tutorial/install-mongodb-on-ubuntu/>

Bước 1: Import “MongoDB public GPG Key” sử dụng command apt-key

```
sudo apt-key adv --keyserver hkp://keyserver.ubuntu.com:80 --recv  
9DA31620334BD75D9DCB49F368818C72E52529D4
```

Bước 2: Thêm repo của MongoDB

```
echo "deb [ arch=amd64 ] https://repo.mongodb.org/apt/ubuntu bionic/mongodb-org/4.0  
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.0.list
```

Bước 3: Cập nhật lại packages

```
sudo apt-get update
```

Bước 4: Cài đặt MongoDB

```
sudo apt-get install -y mongodb-org
```

Bước 5: Khởi động MongoDB

```
sudo service mongod start
```

Sau khi hoàn thành xong, bạn có thể kiểm tra xem trạng thái cài đặt MongoDB.

```
sudo service mongod status
```

Kết quả như bên hình dưới đây là thành công mỹ mãn.

```
ngotuannghia@VNTALKING-VirtualBox:~$ sudo service mongod status
● mongod.service - MongoDB Database Server
   Loaded: loaded (/lib/systemd/system/mongod.service; disabled; vendor preset:
   Active: active (running) since Sat 2020-01-18 23:01:42 +07; 17s ago
     Docs: https://docs.mongodb.org/manual
    Main PID: 32527 (mongod)
      CGroup: /system.slice/mongod.service
              └─32527 /usr/bin/mongod --config /etc/mongod.conf

Jan 18 23:01:42 VNTALKING-VirtualBox systemd[1]: Started MongoDB Database Serve
lines 1-9/9 (END)
```

Hình 5.1: Khởi động dịch vụ MongoDB

Trên đây chỉ là bạn cài xong MongoDB service, nó sẽ chạy ngầm. Để bạn có thể dễ dàng tương tác, tận mắt nhìn dữ liệu được lưu như nào, chỉnh sửa nó khi cần... thì bạn cần phải cài thêm một công cụ nữa.

Trên trang chủ MongoDB có giới thiệu công cụ MongoDB Compass. Tuy nhiên, mình khuyến khích dùng Robo3T, vừa miễn phí lại dễ sử dụng hơn rất nhiều.

Kết nối và quản lý MongoDB với Robo 3T

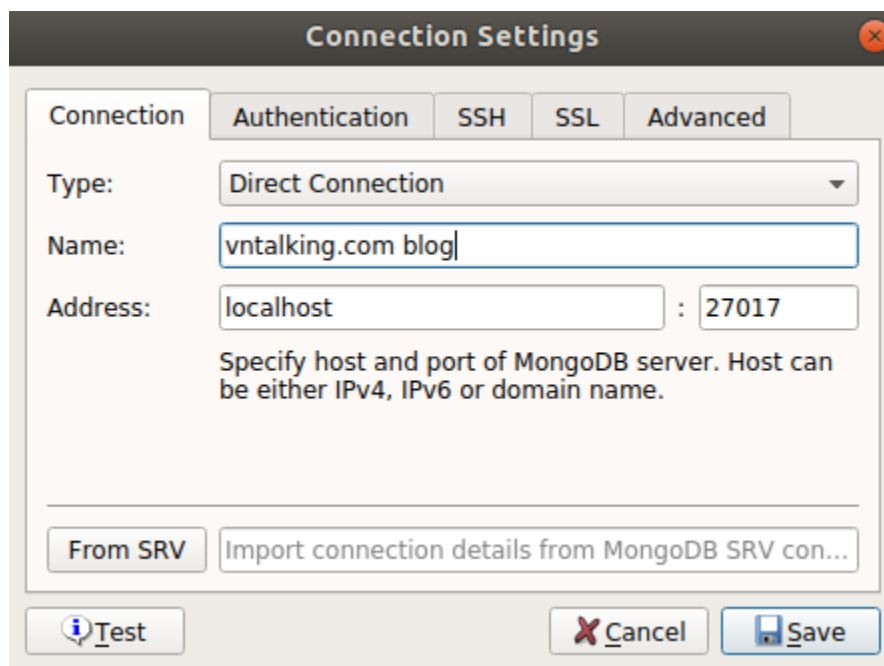
Robo 3T (formerly Robomongo) là một phần mềm mã nguồn mở quản lý MongoDB GUI đa nền tảng, hỗ trợ cả Windows, Linux và MacOS.

Để sử dụng Robo3T, các bạn download phần mềm về tại đây.

<https://robomongo.org/download>

Với Ubuntu thì bạn chỉ việc download về, giải nén và click đúp vào bin/robo3t là được.

Sau khi mở phần mềm lên, bạn tạo một kết nối tới MongoDB service. Nếu bạn kết nối trên localhost thì không cần phải làm gì cả, nhấn nút save là được.



Hình 5.2: Màn hình thiết lập kết nối tới MongoDB

Vậy là phần cài đặt môi trường và công cụ đã xong. Giờ chúng ta quay trở lại với dự án blog bằng Node.js. Để từ Node.js mà tương tác được với MongoDB, bạn cần cài thêm một module hỗ trợ. Phổ biến và dễ dùng nhất là Mongoose.

Cài đặt Mongoose

Mongoose là một thư viện Object Data Modeling (ODM) hỗ trợ làm cầu nối giữa Node.js với MongoDB. Bạn có thể tham khảo các thông tin như lượt cài đặt, hướng dẫn cài đặt tại trang chính thức: <https://www.npmjs.com/package/mongoose>

Việc cài đặt Mongoose cũng rất đơn giản, giống như bao module khác của Node.js

```
npm install mongoose
```

Kết nối MongoDB từ Node.js

Đầu tiên, bạn mở *index.js* và thêm đoạn code này vào.

```
const mongoose = require('mongoose');
mongoose.connect('mongodb://localhost/my_database', {useNewUrlParser: true})
```

Chúng ta định nghĩa một connection thông qua api của mongoose đó là:

mongoose.connect() với tham số là đường dẫn kết nối database và tên database. Trong trường hợp của cuốn sách này, do mình cài MongoDB trên cùng một máy tính nên sẽ để kết nối localhost, *my_database* là tên của DB mình tạo. Có một điểm hay là trong khi kết nối DB, nếu DB chưa tồn tại thì MongoDB sẽ tự động tạo giúp mình.

Định nghĩa Model

Trong thư mục app, bạn tạo thêm một thư mục mới là models. Đây là thư mục sẽ chứa các model, là một Object đại diện cho các Collection trong Database.

Chúng ta sẽ cùng nhau thực hành, từng bước tạo các models tương ứng với mỗi collection.

Đầu tiên là tạo *BlogPost.js*

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const BlogPostSchema = new Schema({
  title: String,
  body: String
});
```

Model được định nghĩa thông qua Schema Interface. Bạn nên nhớ rằng, mỗi một Collection trong MongoDB (tương ứng với một table trong SQL) sẽ tương ứng với một

model. Và Schema sẽ định nghĩa các thuộc tính (các fields) của model như bạn thấy ở đoạn code trên.

Sau khi định nghĩa xong các thuộc tính (fields) của Model thì bạn cần export nó ra để các class khác có thể dùng được.

Vẫn ở trong file *BlogPost.js*:

```
...  
const BlogPost = mongoose.model('BlogPost', BlogPostSchema);  
module.exports = BlogPost
```

Chúng ta truy cập vào cơ sở dữ liệu thông qua hàm *mongoose.model(...)*. Trong đó, tham số đầu tiên chính tên của Collection tương ứng với model này.

Có một lưu ý nho nhỏ là mặc dù bạn đặt tên Collection theo số ít, nhưng trong mongoose sẽ tự động chuyển nó thành số nhiều. Với đoạn code trên, trong MongoDB, chúng ta sẽ có một Collection có tên là *BlogPosts*.

Tạo các action CRUD với Mongoose model

Từ khóa CRUD là viết tắt của 4 hành động: Create - Read - Update - Delete vào cơ sở dữ liệu.

Sau khi chúng ta đã kết nối được với cơ sở dữ liệu, việc tiếp theo là định nghĩa các hành động đọc/ghi.

Để minh họa, mình sẽ tạo một file *test.js* riêng biệt, chúng ta sẽ viết tạm code vào đó trước khi áp dụng cho toàn dự án.

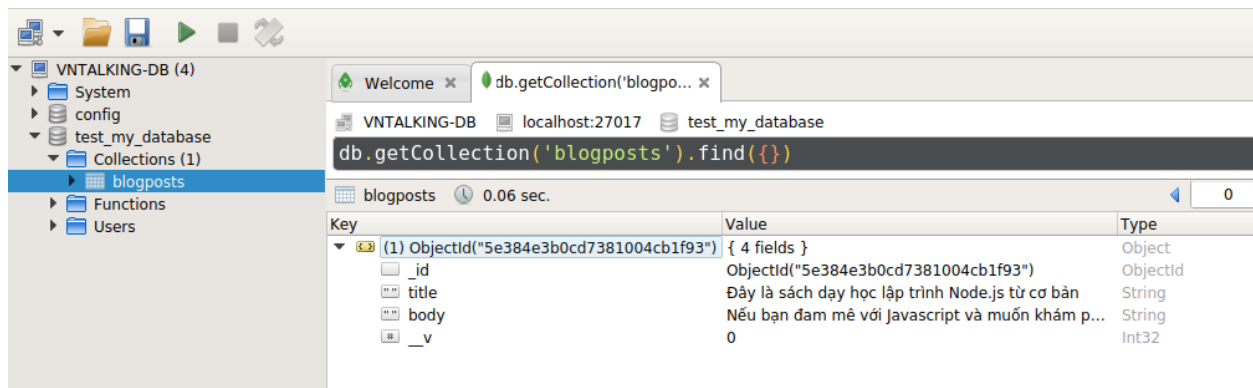
Trong thư mục app, bạn tạo một file đặt tên là *test.js*, rồi viết đoạn code sau:

```
const mongoose = require('mongoose')  
const BlogPost = require('./models/BlogPost')  
  
mongoose.connect('mongodb://localhost/test_my_database', { useNewUrlParser: true });  
  
BlogPost.create({  
  title: 'Đây là sách dạy học lập trình Node.js từ cơ bản',  
  body: 'Nếu bạn đam mê với Javascript và muốn khám phá cách xây dựng ứng dụng với Node.js thì đây là cuốn sách dành cho bạn.'  
}, (error, blogpost) => {  
  console.log(error, blogpost)  
})
```

Đoạn code đơn giản là mình sẽ insert thêm một document (tương ứng là một record trong SQL) vào trong Collection.

Cách thực thi đoạn code trên bằng cách gõ lệnh: `node test.js` trong terminal.

Sau đó bạn kiểm tra trong Robo3T xem đã thêm được document trên chưa. Như hình dưới là đã thêm thành công.



Hình 5.2: Xem thông tin document trong Robo3T

Lấy dữ liệu từ MongoDB

Chúng sẽ sử dụng hàm `find()` để lấy document trong database. Để lấy tất cả documents trong một Collection thì đơn giản là bạn không truyền điều kiện filter khi query.

```
BlogPost.find({}, (error, blogspot) => {  
    console.log(error, blogspot)  
})
```

Còn nếu bạn muốn lọc các document theo một điều kiện thì bạn truyền điều kiện filter vào tham số thứ nhất. Ví dụ, mình muốn lấy ra document có title là:

"Đây là sách dạy học lập trình Node.js từ cơ bản". Bạn làm như sau:

```
BlogPost.find({  
    title: 'Đây là sách dạy học lập trình Node.js từ cơ bản'  
}, (error, blogspot) => {  
    console.log(error, blogspot)  
})
```

Hoặc bạn muốn tìm tất cả những document mà trong title có từ **"Node.js"**.

```
BlogPost.find({  
    title: 'Node.js'
```

```
}, (error, blogspot) => {  
  console.log(error, blogspot)  
})
```

Trên đây chỉ là một số cách tìm kiếm và lấy document bằng hàm *find()*. Bạn có thể đọc thêm rất nhiều điều kiện để tìm và lọc document trong database với *find()* tại đây:

<https://docs.mongodb.com/manual/tutorial/query-documents/>

Update document

Để cập nhật một bản ghi, chúng ta sử dụng hàm *findByIdAndUpdate(...)*, với ID là tham số đầu tiên để xác định bản ghi cần cập nhật.

```
var id = "5cb436980b33147489eadfbb";  
BlogPost.findByIdAndUpdate(id, {  
  title: 'Updated title'  
}, (error, blogspot) => {  
  console.log(error, blogspot)  
})
```

Xóa một document

Xóa một document thì càng đơn giản, chỉ cần truyền ID của document ấy vào hàm *findByIdAndDelete()*.

```
var id = "5cb436980b33147489eadfbb";  
BlogPost.findByIdAndDelete(id, (error, blogspot) => {  
  console.log(error, blogspot)  
})
```

Tổng kết

Qua phần 5, mình đã giới thiệu xong về MongoDB, một loại noSQL để lưu trữ dữ liệu theo collection và document. Trong Node.js, chúng ta sử dụng thư viện mongoose để kết nối tới MongoDB. Ngoài ra, bạn cũng biết được những thao tác CRUD với database thông qua mongoose.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap5>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Ứng dụng MongoDB vào dự án

Đọc được đến phần này thì trước hết mình phải chúc mừng bạn. Bạn đi được quãng đường khá xa rồi đấy.

Ở phần 5, mình đã giới thiệu những kiến thức cơ bản về MongoDB, cũng đã cài đặt môi trường xong hết rồi. Giờ chúng ta bắt tay vào thực hành, sử dụng MongoDB để lưu trữ và quản lý các bài viết trong blog.

Quay trở lại mã nguồn của dự án trong cuốn sách. Chúng ta sẽ hoàn thiện tính năng tạo mới một post của blog.

Trong thư mục *views*, mình tạo mới file *create.ejs*. Để cho nhanh, bạn copy mã nguồn của *contact.ejs* vào *create.ejs*. Lúc này giao diện trang tạo post mới sẽ giống như các trang khác, tức là cũng có thanh điều hướng, header, footer. Chỉ khác là bạn cần thay đổi nội dung thẻ *h1* thành: `<h1> Create New Post</h1>`

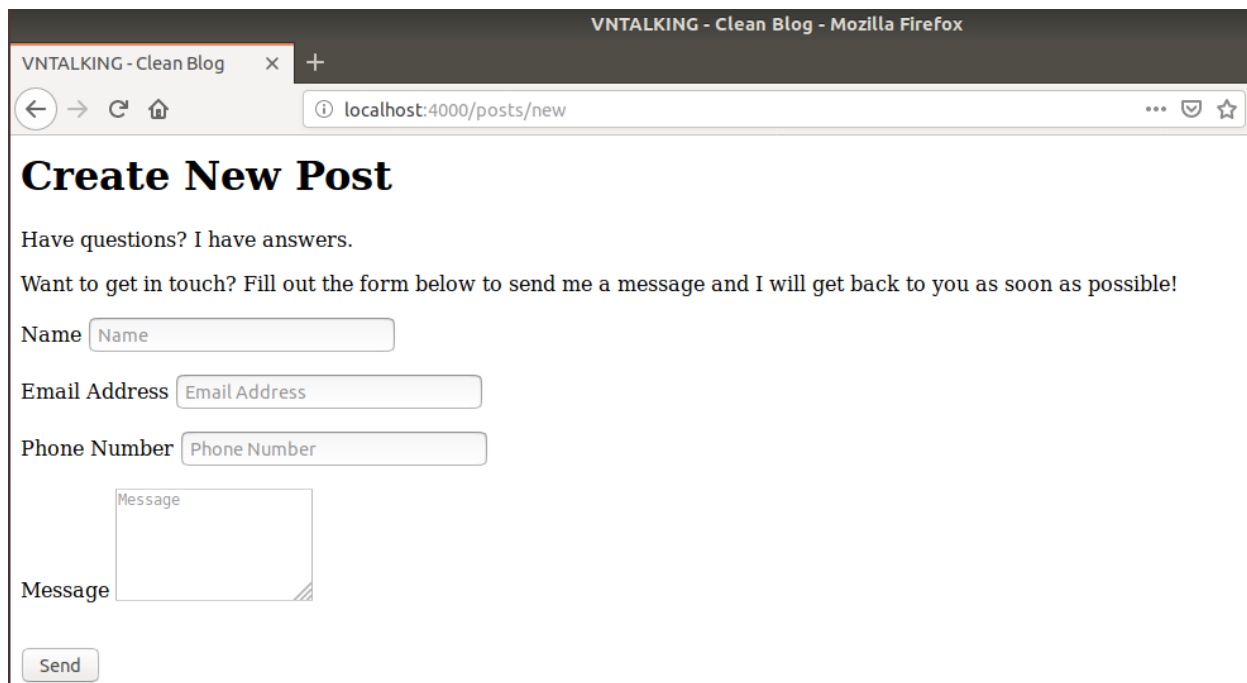
Tiếp theo, chúng ta đăng ký một route cho việc tạo post mới, bằng cách thêm đoạn code sau vào *index.js*

```
app.get('/posts/new', (req, res) => {  
  res.render('create')  
})
```

Giờ chúng ta sẽ thêm menu tạo post mới trên thanh navbar. Bạn chỉ cần chỉnh sửa lại *views/layouts/navbar.ejs*. Chính là phần mình in đậm.

```
<div class="collapse navbar-collapse" id="navbarResponsive">  
  <ul class="navbar-nav ml-auto">  
    ...  
    <li class="nav-item">  
      <a class="nav-link" href="/contact">Contact</a>  
    </li>  
    <li class="nav-item">  
      <a class="nav-link" href="/posts/new">New Post</a>  
    </li>  
  </ul>  
</div>
```

Ok, bạn thử chạy ứng dụng (vẫn là câu lệnh: `npm start`), rồi thử nhấn vào menu "New Post ". Bạn sẽ thấy giao diện như bên dưới.



Hình 6.1: Giao diện trang tạo bài post mới

Dường như các file css, js, images... chưa được tải. Nguyên nhân là do trang tạo bài post mới đang để 2 level route (như này được coi là 2 level route: `"/posts/new "`) nên không thể reference tới các file static cần thiết trong các file mà chúng ta include, như `header.ejs`... Bởi vì thẻ `href` đang để là:

```
<!-- Bootstrap core CSS -->
<link href="vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

Nó không thể thấy được các file trong thư mục vendor từ một sub level. Để sửa lỗi này, thực ra lại rất đơn giản, chỉ cần bạn sửa thẻ href trở tới link trực tiếp là được. Có một cách đó là thêm dấu `"/` vào trước.

```
<!-- Bootstrap core CSS -->
<link href="/vendor/bootstrap/css/bootstrap.min.css" rel="stylesheet">
```

Link trực tiếp có thể là một link đầy đủ kiểu như này:

```
<link href='https://fonts.googleapis.com/css?family=Lora:400,700,400italic,700italic' rel='stylesheet' type='text/css'>
```

Tuy nhiên, mình không khuyến khích cách này, vì bạn sẽ phải hardcode tên miền ở đây. Nhỡ ứng dụng của chúng ta sau này thay đổi tên miền thì sao? Chả nhẽ phải tìm và sửa lại toàn bộ code? Không nên làm vậy.



Bạn nhớ thay đổi lại toàn bộ link cho thẻ `href` của các file static ở tất cả các file layout như `header.ejs`, `navbar.ejs`, `footer.ejs`...

Sau khi cập lại xong thẻ `href`, bạn thử kiểm tra lại xem giao diện đã "ngon lành" chưa?

Hiện tại thì nội dung của `create.ejs` mình lấy nguyên bản từ `contact.ejs`. Giờ chúng ta sẽ sửa đổi lại một chút, màn hình tạo post chỉ cần 2 trường để nhập dữ liệu là: tiêu đề và nội dung bài viết.

```
<!-- Main Content -->
<div class="container">
  <div class="row">
    <div class="col-lg-8 col-md-10 mx-auto">
      <form action="/posts/store" method="POST">
        <div class="control-group">
          <div class="form-group floating-label-form-
group controls">
            <label>Title</label>
            <input type="text" class="form-
control" placeholder="Title" id="title" name="title">
          </div>
        </div>
        <div class="control-group">
          <div class="form-group floating-label-form-group controls">
            <label>Content</label>
            <textarea rows="5" class="form
control" id="body" placeholder="Content" name="body"></textarea>
          </div>
        </div>
        <br>
        <div class="form-group">
          <button type="submit" class="btn btn-
primary" id="sendMessageButton">Create</button>
        </div>
      </form>
    </div>
  </div>
</div>
```

Các bạn để ý phần mình bôi đậm nhé. Trong đó các bạn lưu ý mấy thông số sau:

- `method="POST"`: Tức khi nhấn nút Create, chúng ta sẽ tạo một POST request tới server.
- `action: "/posts/store"`: Là tên route sẽ nhận request.

Tuy nhiên, hiện tại trong phần xử lý back-end, chúng ta chưa có viết code xử lý cho POST request này. Vậy thì làm luôn, bạn mở file *index.js*:

```
app.post('/posts/store', (req, res) => {  
  console.log(req.body)  
  res.redirect('/')  
})
```

Ở hàm này, chúng ta lấy dữ liệu từ trình duyệt gửi lên thông qua trường *body* của request. Nhưng mặc định thì node.js chưa hỗ trợ, chúng ta cần cài thêm module *body-parser*, mục đích để module này sẽ parse những dữ liệu trong POST request rồi đưa vào trường *body* để bạn có thể lấy ra một cách dễ dàng.

Cài đặt *body-parser* module như bình thường bằng lệnh:

```
npm install body-parser
```

Sau đó, trong *index.js* thì cần khai báo nó.

```
const bodyParser = require('body-parser')  
app.use(bodyParser.json())  
app.use(bodyParser.urlencoded({extended:true}))
```

Xong, giờ bạn chạy lại app blog và kiểm tra kết quả. Nếu bạn nhấn nút "create" ở trang tạo post mới, mà trong console thấy log sau là được.

```
{ title: 'title1', body: 'body1' }
```

Lưu dữ liệu bài Post vào Database

Ở phần trước, chúng ta mới chỉ xử lý nhận được dữ liệu từ trình duyệt thông qua trường *req.body*. Tuy nhiên, chúng ta hoàn toàn chưa làm gì với dữ liệu này cả, chỉ đơn giản là in log ra màn hình console: `console.log(req.body)`.

Giờ mình sẽ lưu dữ liệu nhận được vào trong database, cụ thể là lưu vào *BlogPosts* collection. Ở trong phần giới thiệu MongoDB, mình đã viết đoạn code để lưu dữ liệu vào 1 collection sử dụng Mongoose, các bạn xem lại nhé.

Giờ mở *index.js* và thực hiện như bên dưới:

Khai báo model trên đầu file của *index.js*

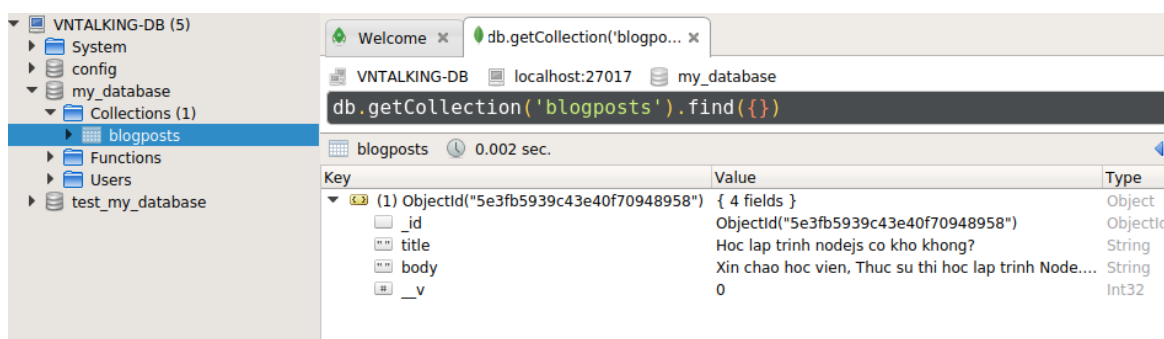
```
const BlogPost = require('./models/BlogPost.js')
```

Sau đó cập nhật lại route xử lý nhận dữ liệu từ trình duyệt:

```
app.post('/posts/store', (req, res) => {  
  // model creates a new doc with browser data  
  BlogPost.create(req.body, (error, blogpost) => {  
    res.redirect('/')  
  })  
})
```

Ở đoạn code trên, sau khi mình lưu dữ liệu xong thì sẽ redirect trình duyệt về trang chủ.

Giờ mình chạy thử, vào trình duyệt và tạo một post mới, nhấn nút "Send", rồi kiểm tra dữ liệu bằng công cụ Robo3T.



Hình 6.2: kết quả lưu một bài post trong MongoDB

Như vậy là mình đã lưu thành công một post vào cơ sở dữ liệu. Bước tiếp theo, chúng ta sẽ lấy dữ liệu từ database hiển thị ra ngoài trang web.

Hiển thị danh sách các bài Post

Để hiển thị danh sách các bài post ra ngoài website, chúng ta sử dụng hàm `find()` của model để query vào database (Nếu bạn quên hàm này thì mời bạn đọc lại phần: "Giới thiệu MongoDB").

Hàm `find()` này có tác dụng query vào database để lấy các documents theo điều kiện lọc nào đó. Nếu bạn không truyền điều kiện gì vào thì nghĩa là lấy tất cả documents trong collection đó.

Giờ mình muốn hiển thị toàn bộ post ở ngoài trang chủ thì sửa lại route `"/` trong `index.js`

```
app.get('/', (request, response) => {  
  BlogPost.find({}, function (error, posts) {  
    console.log(posts);  
  })  
})
```


Đoạn code trên thì mình mới chỉ query vào DB, sau đó in kết quả ra console mà thôi. Bạn thử chạy chương trình và kiểm tra kết quả trong console nhé. Như của mình thì kết quả sẽ như sau:

```
[ { _id: 5e3fb5939c43e40f70948958,
  title: 'Hoc lap trinh nodejs co kho khong?',
  body: 'Xin chao hoc vien,\r\nThuc su thi hoc lap trinh Node.js khong he kho.',
  __v: 0 },
  { _id: 5e3fb60153f5350fcdc55bdd,
    title: 'sada',
    body: 'asdsad',
    __v: 0 } ]
```

Đến đây chúng ta sẽ tiếp tục để hiển thị được dữ liệu ra ngoài trang web. Ở phần 4, mình đã giới thiệu về template engine, giờ là lúc dùng đến nó rồi.

Hiển thị dữ liệu động với Template engine

Trước hết, trong phần route, chúng ta cần truyền dữ liệu lấy được từ database sang file giao diện được viết bởi template engine, cụ thể ở đây là file *index.ejs*

```
app.get('/', (request, response) => {
  BlogPost.find({}, function (error, posts) {
    console.log(posts);
    response.render('index', {
      blogposts: posts
    });
  })
})
```

Phần mình bôi đậm có nghĩa là dữ liệu trả về sẽ được gán cho biến *blogposts*. Đây là biến sẽ được sử dụng trong file *index.ejs*. Giờ mở file *index.ejs*, bạn có thấy trong tag: `<div class="post-preview">` mình đang hard code nội dung hiển thị cho từng post trong danh sách các post.

```
<div class="post-preview">
  <a href="post.html">
    <h2 class="post-title">
      Science has not yet mastered prophecy
    </h2>
    <h3 class="post-subtitle">
      We predict too much for the next year and yet far too little fo
r the next ten.
    </h3>
  </a>
```

```

    <p class="post-meta">Posted by
      <a href="#">Start Bootstrap</a>
      on August 24, 2019</p>
  </div>

```

Giờ đã có dữ liệu từ database, mình sẽ không phải hard code nữa, thay vào đó sẽ lấy dữ liệu từ biến mà được truyền từ bên *index.js* sang.

```

<div class="col-lg-8 col-md-10 mx-auto">

  <% for (var i = 0; i < blogposts.length; i++) { %>
    <div class="post-preview">
      <a href="/post/<%= blogposts[i]._id %>">
        <h2 class="post-title">
          <%= blogposts[i].title %>
        </h2>
        <h3 class="post-subtitle">
          <%= blogposts[i].body %>
        </h3>
      </a>
      <p class="post-meta">Posted by
        <a href="#">Start Bootstrap</a>
        on September 24, 2019</p>
    </div>
    <hr>
  <% } %>
<!-- Pager -->

```

Với đoạn code có nghĩa là chúng ta sẽ duyệt mảng dữ liệu nhận được từ bên *index.js* gửi sang, với một phần tử của mảng, chúng ta sẽ truy xuất dữ liệu như *title*, *body* để điền vào html. Như vậy, với phần khung của html như các thẻ `<div class="post-preview">` thì không hề thay đổi, chỉ có nội dung của các thẻ div như `<div class="post-title">` là thay đổi.

Giờ bạn chạy ứng dụng và hưởng thụ thành quả nào.



Hình 6.3: giao diện trang hiển thị danh sách bài posts

Còn hai trường thông tin mà vẫn còn "hard code" (tức là nội dung không thay đổi theo dữ liệu truyền vào):

- `<p class="post-meta">Posted by ...`
- Và ngày đăng bài: `on September 24, 2019</p>`

Hiện tại thì trong cơ sở dữ liệu mình chưa lưu 2 thông tin này nên tạm thời cứ vậy đi. Chúng ta sẽ hoàn thiện ở phần sau của cuốn sách nhé.

Hiển thị nội dung một Post

Từ danh sách các bài post mà chúng ta đã hiển thị ở phần trước, giờ nếu click vào một post thì trang web sẽ hiển thị toàn bộ nội dung của bài post đó. Để biết được bài post nào thì chúng ta cần phải biết *id* của nó trong database. Ở trên, chúng ta đã truyền *id* của bài post thông qua thẻ href: `<a href="/post/<%= blogposts[i]._id %">`

Chúng ta mở file `index.js` để chỉnh sửa lại route:

```
app.get('/post', (req, res) => {  
  res.render('post')  
})
```

Thành:

```
app.get('/post/:id', (req, res) => {  
  BlogPost.findById(req.params.id, function(error, detailPost){  
    res.render('post', {  
      detailPost  
    })  
  })  
})
```

Ở đoạn code trên, `app.get('/post/:id' ...)`. Chúng ta thêm *id* vào url của route. Lúc này đường URL để request sẽ có dạng kiểu như này:

`http://localhost:4000/post/5cb836f610d8d629530fcf82`.

Ok, như vậy là chúng ta đã query vào database để lấy được toàn bộ nội dung của một post. Giờ phần tiếp theo là chỉnh sửa và đưa dữ liệu đó vào template engine để gen thành html và trả về cho trình duyệt.

Tương tự như phần hiển thị danh sách bài post bằng template engine. Mình sẽ mở file `post.ejs` và chỉnh sửa một chút.

Các bạn để ý những phần mình bôi đậm nhé.

```

<!-- Page Header -->
<header class="masthead" style="background-image: url('img/post-bg.jpg')">
  <div class="overlay"></div>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <div class="post-heading">
          <h1><%= detailPost.title %></h1>
          <h2 class="subheading"><%= detailPost.body %></h2>
          <span class="meta">Posted by
            <a href="#">Start Bootstrap</a>
            on August 24, 2019</span>
        </div>
      </div>
    </div>
  </div>
</header>

<!-- Post Content -->
<article>
  <div class="container">
    <div class="row">
      <div class="col-lg-8 col-md-10 mx-auto">
        <%= detailPost.body %>
      </div>
    </div>
  </div>
</article>

```



detailPost chính là tên biến trùng với tên biến mà bạn đã truyền vào hàm `render` trong route `app.get('/post/:id',...)`

Giờ bạn thử truy cập vào trình duyệt xem chương trình đã chạy đúng chưa nhé.

Như mình đã nói ở trên, giờ mình sẽ hoàn thiện thêm thông tin cho mỗi bài post, đó là thêm hai thông tin: tác giả và ngày đăng bài.

Thêm Fields và Schema

Bởi vì hiện tại chúng ta chỉ mới định nghĩa hai fields: title và body để lưu thông tin cho mỗi bài post. Chúng ta cần thêm hai field nữa là: username và datePosted, để lưu thông tin về người viết bài post và ngày giờ đăng bài post.

Bạn mở file `model/BlogPost.js` và thêm đoạn mình bôi đậm như dưới đây.

```
const mongoose = require('mongoose')
```

```

const Schema = mongoose.Schema;
const BlogPostSchema = new Schema({
  title: String,
  body: String,
  username: String,
  datePosted: { /* can declare property type with an object like this because we need 'default' */
    type: Date,
    default: new Date()
  }
});

```

Tiếp theo, mình sẽ cần thay đổi *index.ejs*:

```

<p class="post-meta">Posted by
  <a href="#"><%= blogposts[i].username %></a>
  on <%= blogposts[i].datePosted.toDateString() %></p>
</div>

```

và *post.ejs*

```

<div class="post-heading">
  <h1><%= detailPost.title %></h1>
  <h2 class="subheading"><%= detailPost.body %></h2>
  <span class="meta">Posted by
    <a href="#"><%= detailPost.username %></a>
    on <%= detailPost.datePosted.toDateString() %></span>
</div>

```

Giờ bạn xóa record trong database và tạo post mới để nó lưu đầy đủ thông tin mà mình đã thêm ở trên. Lúc này thì trường *datePosted* sẽ được gen tự động và điền tự động lúc lưu record vào database. Còn trường *username*, mình sẽ hoàn thiện thêm khi hướng dẫn tạo tính năng login.

Tổng kết

Hoàn thành phần 6 là bạn đã biết cách tạo một ứng dụng kết nối tới database. Đã biết cách lưu và lấy dữ liệu từ Database. Đặc biệt là biết cách sử dụng *body-parser* để nhận dữ liệu từ form field data được gửi từ trình duyệt.

Ngoài ra, đọc xong phần này bạn cũng biết cách sử dụng template engine để bind dữ liệu vào giao diện.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

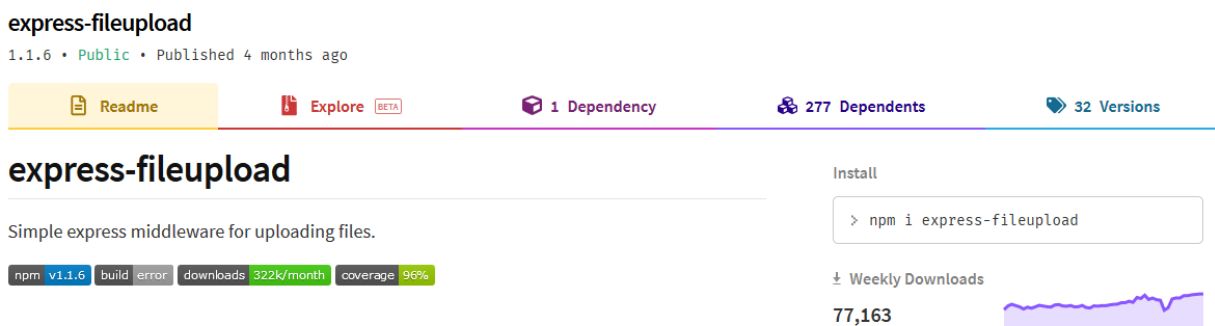
<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap6>

Tạo tính năng upload ảnh với Express

Phần trước, chúng ta đã biết cách lưu một bài viết vào database, nhưng nội dung chỉ có text thôi, để cho bài post sinh động hơn, bạn muốn thêm ảnh cho nó thì sao? Phần này, mình sẽ hướng dẫn các bạn cách upload ảnh cho một bài post.

Để làm tính năng upload ảnh đơn giản hơn, chúng ta sẽ sử dụng module: [express-fileupload](#).

Đây là một module rất được ưu chuộng, với số lượt tải về hàng tuần rất lớn (gần 80K lượt tải/tuần).



Hình 7.1: thông tin module express-fileupload

Việc cài đặt module như mọi module khác thôi.

```
# With NPM
npm install --save express-fileupload
```

Tiếp theo, mình sẽ tạo một upload field để người dùng có thể chọn ảnh trên local của họ và upload lên server.

Trong `views/create.ejs`, thêm đoạn code mình bôi đậm bên dưới đây:

```
<form action="/posts/store" method="POST" enctype="multipart/form-data">
...

<div class="control-group">
  <div class="form-group floating-label-form-group controls">
    <label>Image</label>
```

```

        <input type="file" class="form-
control" id="image" name="image">
      </div>
    </div>
    <br>
    <div class="form-group">

```

Trong đó:

```
<form action="/posts/store" method="POST" enctype="multipart/form-data">
```

Mình thêm thuộc tính `enctype="multipart/form-data"` để báo cho trình duyệt biết là form có chứa dữ liệu multimedia. Trình duyệt sẽ tự động mã hóa chúng trước khi gửi lên server.

Phần tiếp theo, chúng ta sẽ xử lý phần route nhận request `"/posts/store"`. Bạn cần khai báo module `express-fileupload`. Sau đó thêm chúng vào middleware của `express`.

```

...
const fileUpload = require('express-fileupload')
app.use(fileUpload())
...

app.post('/posts/store', (req, res) => {
  let image = req.files.image;
  image.mv(path.resolve(__dirname, 'public/upload', image.name), function (err) {
    // model creates a new doc with browser data
    BlogPost.create(req.body, (error, blogpost) => {
      res.redirect('/')
    })
  })
})

})

```

Với đoạn code trên, chúng ta sẽ lưu ảnh vào thư mục `"public/upload"` trên server.

Giờ bạn chạy ứng dụng bằng lệnh: `npm start`. Kiểm tra kết quả trên trình duyệt.

localhost:4000/posts/new

Title

Day la bai viet co anh girl xinh

Content

Xin chào đọc giả.
Bài viết này man phép upload ảnh girl xinh nhe

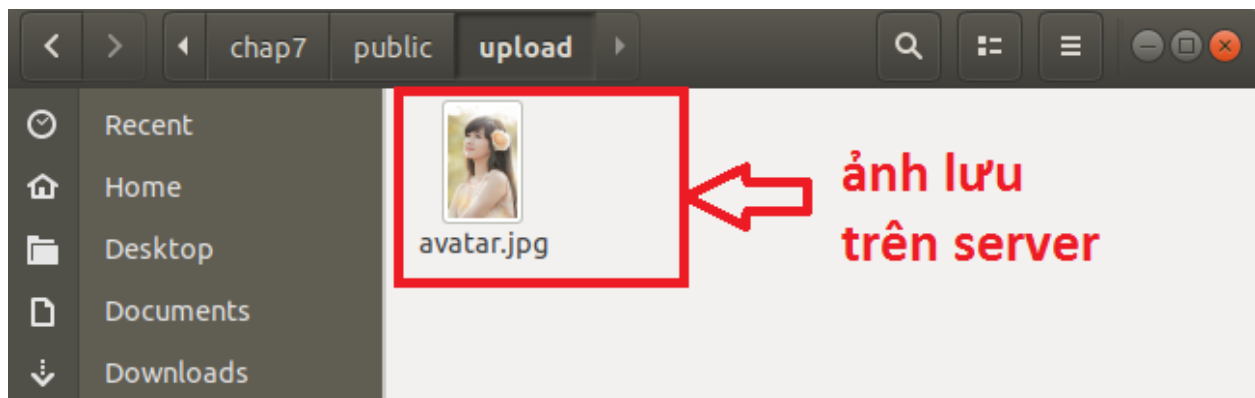
Image

Browse... avatar.jpg

SEND

Hình 7.2: Giao diện trang tạo bài post mới thêm nút upload ảnh

Và kiểm trên server xem ảnh đã được lưu đúng folder chưa.



Hình 7.3: Ảnh được lưu trên server

Ở đây, chúng ta mới chỉ lưu ảnh vào một folder mà chưa có sự liên kết giữa việc ảnh nào của bài post nào. Để tạo sự liên kết đó, chúng ta sẽ cần lưu đường dẫn của ảnh vào trong database tương ứng với bài post đó.

Đơn giản thôi, trong Schema của bài post, chúng ta thêm một field là: *image*

Mở *models/BlogPost.js* và thêm field như bên dưới:

```
const BlogPostSchema = new Schema({
  ...
  image: String
});
```

Và phần xử lý route *"/post/store"* trong *index.js*

```
app.post('/posts/store', function (req, res) {
  let image = req.files.image;
  image.mv(path.resolve(__dirname, 'public/upload', image.name), function (
  error) {
    BlogPost.create({
      ...req.body,
      image: '/upload/' + image.name
    }, function (err) {
      res.redirect('/')
    })
  })
})
})
```

Giờ bạn kiểm tra bằng cách tạo post mới và upload một ảnh. Sau đó kiểm tra trong DB bằng robo3T xem nhé.



Hình 7.4: đường dẫn ảnh được lưu trong DB

Ok, sau khi đã lưu được ảnh vào DB, giờ bạn có thể hiển thị nó mỗi khi hiển thị một bài post. Chúng ta cần sửa lại phần code giao diện một chút.

Mở *view/post.ejs*, thay đổi đoạn code như mình bôi đậm.

```
<header class="masthead" style="background-image: url('img/post-bg.jpg')">
```

Chuyển thành:

```
<header class="masthead" style="background-  
image: url('<%= detailPost.image %>')">
```

Bạn thử mở một post để hưởng thụ thành quả nhé.



Hình 7.5: Ảnh được hiển thị lên trang web

Tổng kết

Qua phần này, chúng ta đã tìm hiểu và biết cách upload một hình ảnh lên server, sau đó thì hiển thị nó ra ngoài bài post. Bạn muốn tìm hiểu kỹ về những api của module: *express-fileupload* thì có thể tham khảo thêm tại trang chủ của nó nhé:

<https://www.npmjs.com/package/express-fileupload>

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap7>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Tìm hiểu Express Middleware

Middleware là một functions cho phép ExpressJS thực hiện một công việc sau khi nhận được một request từ client, nó sẽ thực hiện "xào nấu" để có được một kết quả có thể trả về cho client hoặc cho một middleware tiếp theo.

Chúng ta có thể có nhiều middleware, và chúng sẽ thực hiện theo tuần tự mà chúng ta khai báo.

Trong bài thực hành của cuốn sách này, chúng ta đã sử dụng middleware rồi đó. Ví dụ như trong *index.js*, khi chúng ta gọi `app.use(...)`, tức là chúng ta đã apply một middleware cho Express rồi.

```
app.use(express.static('public'))
app.use(bodyParser.json())
app.use(bodyParser.urlencoded({extended:true}))
app.use(fileUpload())
```

Hàm `use()` là một API của ExpressJS, nó cho phép chúng ta đăng ký một middleware. Vì vậy, khi trình duyệt tạo một request tới server, ExpressJS sẽ thực thi tất cả những middleware được đăng ký bởi hàm `use()` trước khi xử lý request đó.

Middleware tùy chỉnh

Bạn có thể tự tạo một middleware để xử lý một tác vụ nào đó trước khi trả kết quả về cho client.

Ví dụ trong *index.js*, bạn có thể tạo một middleware như sau:

```
...
const customMiddleware = (req, res, next) => {
  console.log('Custom middle ware called')
  next()
}
app.use(customMiddleware)
...
```

Giờ thì mỗi lần bạn refresh ứng dụng, message "*Custom middle ware called*" sẽ được in trong màn hình console.

Hàm `next()` có mục đích báo cho Express rằng: mỗi khi middleware này hoàn thành thì sẽ tự động chuyển sang thực thi middleware tiếp theo. Nếu bạn bỏ hàm `next()` thì ứng dụng sẽ treo vì nó sẽ không biết thực hiện gì tiếp theo. Tất cả những middleware đăng ký với Express thông qua `use()` để phải gọi `next()`.

Tạo và đăng ký Validation middleware

Có một use case (tình huống sử dụng) phổ biến của middleware đó là trong form validation. Có thể là form đăng ký thành viên, form mua hàng... Những thông tin được gửi từ trình duyệt đều phải validate trước khi đưa vào xử lý.

Ví dụ: form đăng ký thành viên, mục điền email thì không được để trống, địa chỉ email phải hợp lệ... Tất cả những rule đều phải kiểm tra trước khi chúng ta chuyển sang bước xử lý tiếp theo như kiểm tra/lưu vào database.

Chúng ta sẽ tạo một validation middleware với những rule đơn giản như sau:

```
...
const validateMiddleware = (req, res, next) => {
  if (req.files == null || req.body.title == null || req.body.title == null)
} {
  return res.redirect('/posts/new')
}
next()
}
...
```

`validateMiddleWare` đơn giản chỉ là kiểm tra tất cả fields gửi lên có bị null (có thể do người dùng không nhập gì cả) hay không? Nếu có một trong số field bị null thì redirect về lại màn hình tạo post mới.

Nếu chúng ta sử dụng hàm `app.use(validateMiddleWare)` để apply hàm middleware này thì nó sẽ được gọi mỗi khi có bất kỳ request nào tới server, mà như thế thì không đúng. Chúng ta chỉ cần gọi validation middleware mỗi khi tạo bài post mới mà thôi. Vì vậy, mình sẽ apply hàm middleware thông qua URL cụ thể, như sau:

```
...
app.use('/posts/new', validateMiddleware)
...
```

Như vậy là mỗi khi có request tới URL: `/posts/new`, Express sẽ thực thi middleware `validateMiddleWare`.



Lưu ý: Đảm bảo đoạn code đăng ký **`validateMiddleware`** này nằm bên dưới đoạn code `app.use(fileUpload())`. Bởi vì chúng ta cần phải sử dụng thuộc tính files của đối tượng req.

Tổng kết

Như vậy là mình đã giới thiệu kiến thức cơ bản nhất về middleware trong Express. Đây là khái niệm sử dụng rất nhiều trong express, các bạn lưu ý nhé.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap8>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Refactoring theo mô hình MVC

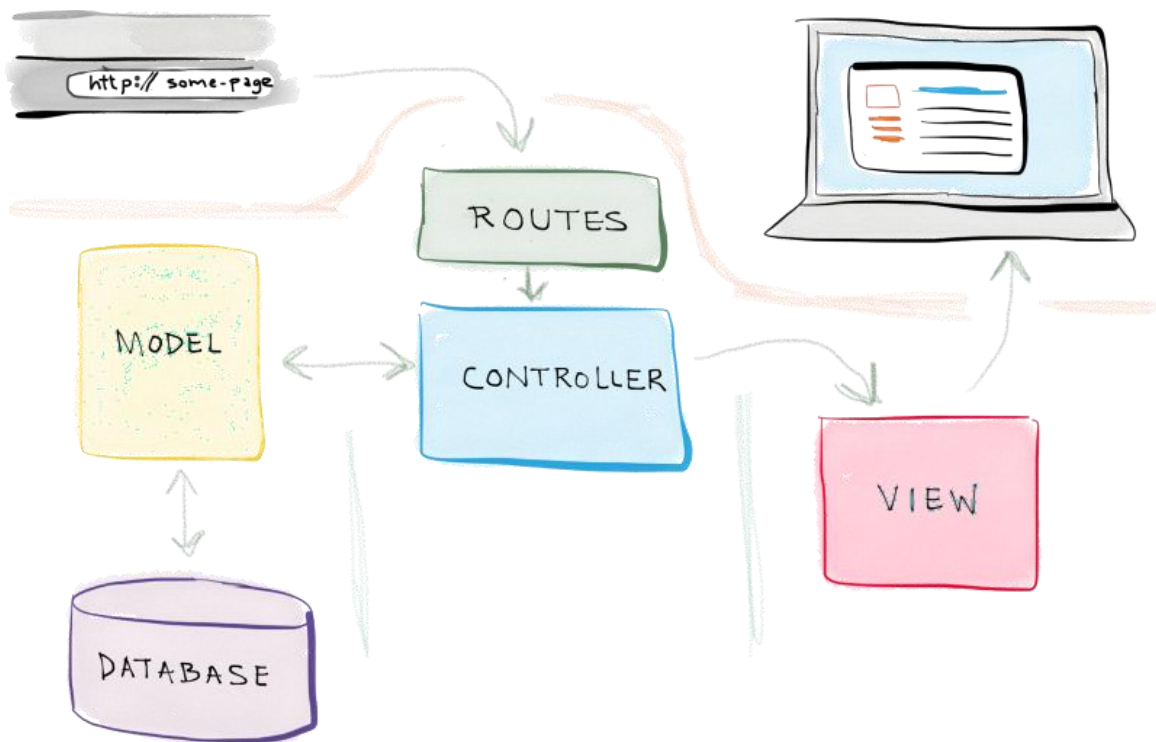
Cho đến hết phần 8 của cuốn sách này, chúng ta chủ yếu viết code trong tệp *index.js*, tất cả việc xử lý các request như *app.get(...)*, *app.post(...)* đều đặt trong *index.js*. Đây không phải là cách tiếp cận tốt khi mà dự án trở lên phức tạp hơn.

Bản thân ExpressJS framework cũng đã tạo sẵn cấu trúc dự án theo mô hình MVC. Các bạn để ý sẽ thấy, ngay khi khởi tạo dự án với ExpressJS, chúng ta đã có sẵn các thư mục như: *models*, *views*, *public*...

Do vậy, để cho chuyên nghiệp hơn, mình sẽ dành hẳn một phần của cuốn sách để "tái cấu trúc"(refactoring) lại mã nguồn theo đúng chuẩn của mô hình MVC.

Giới thiệu mô hình MVC

Dành cho bạn bạn chưa biết về khái niệm MVC.



Hình 9.1: Minh họa mô hình MVC

MVC là từ viết tắt của "**Model View Controller**". Với mô hình MVC, chúng ta xem xét cấu trúc ứng dụng liên quan đến cách luồng dữ liệu của ứng dụng của chúng ta hoạt động như thế nào.

Dễ hiểu hơn, đây là mô hình phân bố source code thành 3 phần, mỗi thành phần có một nhiệm vụ riêng biệt và độc lập với các thành phần khác. Vì vậy, khi bạn muốn thay đổi một thành phần thì các thành phần khác sẽ không phải cập nhật lại mã nguồn.

Các thành phần trong mô hình MVC:

Mô hình MVC được chia làm 3 lớp xử lý gồm Model – View – Controller :

- **Model** : là nơi chứa những nghiệp vụ tương tác với dữ liệu hoặc hệ quản trị cơ sở dữ liệu (mysql, MongoDB...). Nó sẽ bao gồm các class/function xử lý nhiều nghiệp vụ như kết nối, truy vấn, thêm – sửa - xóa cơ sở dữ liệu...
- **View** : là nơi chứa những giao diện như một nút bấm, khung nhập, menu, hình ảnh... nó đảm nhiệm nhiệm vụ hiển thị dữ liệu và giúp người dùng tương tác với hệ thống.
- **Controller** : là nơi tiếp nhận những yêu cầu xử lý được gửi từ người dùng, nó sẽ gồm những class/ function xử lý nhiều nghiệp vụ logic giúp lấy đúng dữ liệu thông tin cần thiết nhờ các nghiệp vụ lớp Model cung cấp và hiển thị dữ liệu đó ra cho người dùng nhờ lớp View.

Quay trở lại mã nguồn của chúng ta, hiện tại thì đã có Model và View layer, giờ sẽ bổ sung thêm Controller layer.

Tiến hành Refactoring

Đầu tiên, chúng ta sẽ tạo thêm một thư mục mới, đặt tên là *controllers*, sau đó tạo thêm một file mới là *newPost.js*. File này sẽ chứa tất cả các hàm để xử lý các request từ user khi tạo một bài post mới.

Trong *newPost.js*, thêm đoạn code sau:

```
module.exports = (req, res) => {  
  res.render('create')  
}
```

Trong *index.js*, thay thế đoạn code xử lý request tạo bài post mới:

```
app.get('/posts/new', (req, res) => {  
  res.render('create')  
})
```

Thành

```
...  
const newPostController = require('./controllers/newPost')  
...  
  
app.get('/posts/new', newPostController)
```

Như vậy là chúng ta đã tách đoạn code xử lý logic cho request tạo bài post mới. Cách tiếp cận này sẽ giúp cho mã nguồn của index.js gọn nhẹ hơn.

Do phần mã nguồn chúng ta sử dụng lại template có sẵn nên có một số mục mà trong phạm vi cuốn sách này, mình không thực hiện. Đó là các trang như: *about*, *contact*, *sample post*. Mình sẽ bỏ chúng ra khỏi mã nguồn để cho gọn gàng hơn. Sau khi thực hành xong cuốn sách này, các bạn thử tự mình thực hiện chúng, coi như là bài tập về nhà.

Phần tiếp theo, chúng ta tiếp tục "quy hoạch" lại phần nhận và xử lý request từ client cho home page, tạo bài post mới và hiển thị một bài post cụ thể.

Trong thư mục controllers, chúng ta tạo thêm 3 controllers nữa gồm: *home.js*, *storePost.js*, và *getPost.js*.

home.js

```
const BlogPost = require('../models/BlogPost.js')  
module.exports = (req, res) => {  
  BlogPost.find({}, function (error, posts) {  
    console.log(posts);  
    res.render('index', {  
      blogposts: posts  
    });  
  })  
}
```

getPost.js

```
const BlogPost = require('../models/BlogPost.js')  
module.exports = (req, res) => {  
  BlogPost.findById(req.params.id, function (error, detailPost) {  
    res.render('post', {  
      detailPost  
    })  
  })  
}
```


storePost.js

```
const BlogPost = require('../models/BlogPost.js')
const path = require('path')
module.exports = (req, res) => {
  let image = req.files.image;
  image.mv(path.resolve(__dirname, '..', '/public/upload', image.name), function (error) {
    BlogPost.create({
      ...req.body,
      image: '/upload/' + image.name
    }, function (err) {
      res.redirect('/')
    })
  })
}
```

Cuối cùng là import những controllers này vào *index.js*

```
...
const homeController = require('./controllers/home')
const storePostController = require('./controllers/storePost')
const getPostController = require('./controllers/getPost')
...

app.get('/', homeController)
app.get('/post/:id', getPostController)
app.post('/posts/store', storePostController)
```

Bởi vì *index.js* không còn sử dụng đến *path* và *BlogPost* object nữa, nên chúng ta bỏ chúng luôn.

```
const path = require('path')
const BlogPost = require('../models/BlogPost.js')
```

Cuối cùng, nhìn thấy mấy cái middleware khá là "ngứa mắt". Tiện thể quy hoạch thì mình sẽ để hết những middleware vào một thư mục. Mình sẽ tạo thêm một thư mục "middleware", sau đó thêm *validationMiddleware.js*. Cách làm tương tự như refactoring cho controller.

validationMiddleware.js

```
module.exports = (req, res, next) => {
  if (req.files == null || req.body.title == null || req.body.title == null) {
    return res.redirect('/posts/new')
  }
  next()
}
```

Sửa lại *index.js*

```
const validateMiddleware = require("../middleware/validationMiddleware");  
app.use('/posts/store', validateMiddleware)
```

Sau tất cả, bạn thử chạy lại ứng dụng xem có gặp bất kì lỗi lầm nào không? Nếu mọi thứ vẫn chạy như cũ thì việc refactoring đã thành công.

Tổng kết

Qua phần 9, mình đã tiến hành refactoring lại source code theo đúng mô hình MVC. Chúng ta đã giảm kích thước của *index.js* để mã nguồn được tổ chức tốt hơn, dễ bảo trì hơn.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap9>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Tạo tính năng đăng ký thành viên

C ho đến hiện tại thì dự án blog của chúng ta vẫn đang tạo các bài post mới mà không có gán cho bất kỳ thành viên nào. Trong thực tế, trước tiên người dùng cần phải đăng ký tài khoản, đăng nhập vào blog trước khi có thể đăng bài post mới.

Trong phần này, chúng ta sẽ cùng nhau xây dựng tính năng đăng ký thành viên nhé.

Đầu tiên, chúng ta sẽ xây dựng giao diện cho tính năng này. Bạn vào thư mục views, tạo thêm một file đặt tên là `register.ejs`. Về cơ bản thì màn hình đăng ký thành viên cần mấy fiels để nhập thông tin như tên dùng, mật khẩu và nút gửi đăng ký. Nó khá tương đồng với màn hình tạo bài post mới, nên để tiết kiệm công code, mình sẽ clone code từ `create.ejs` sang `register.ejs`, rồi chỉnh sửa lại.

Dưới đây là một số chỉnh sửa:

Trong `register.ejs`, bạn thay đổi tiêu đề thành *"Register a new account"*.

```
...  
<div class="page-heading">  
  <h1>Register a new account</h1>  
</div>  
...
```

Thay đổi form action thành: `"/users/register"`

```
...  
<form action="/users/register" method="POST" enctype="multipart/form-data">  
...
```

Trong trường title thì thay đổi thành username.

```
<div class="control-group">  
  <div class="form-group floating-label-form-group controls">  
    <label>User Name</label>  
    <input type="text" class="form-control" placeholder="User Name" id="username" name="username">  
  </div>  
</div>
```

Xóa trường description vì nó dùng cho nội dung bài viết, không cần thiết cho màn hình đăng ký thành viên, tiện thể xóa luôn cả trường image.

Clone trường username và đổi tên thành password. Lưu ý là type của field là password, để khi người dùng nhập thì nó sẽ chuyển thành các ký tự * hoặc chấm, tránh người khác nhìn thấy password.

```
<div class="control-group">
  <div class="form-group floating-label-form-group controls">
    <label>Password</label>
    <input type="password" class="form-
control" placeholder="Password" id="password"
name="password">
  </div>
</div>
```

Cuối cùng là đổi tên nút submit thành "register"

```
<button type="submit" class="btn btn-primary">Register</button>
```

Như vậy là xong phần giao diện người dùng. Chúng ta sẽ xử lý tiếp đến phần controller.

Trong thư mục controllers, bạn tạo thêm file và đặt tên là: *newUser.js*. Nội dung như sau:

```
module.exports = (req, res) => {
  res.render('register') // render register.ejs
}
```

Sau đó thì khai báo controller trong *index.js*:

```
...
const newUserController = require('./controllers/newUser')
...
```

Và tạo một route cho nó.

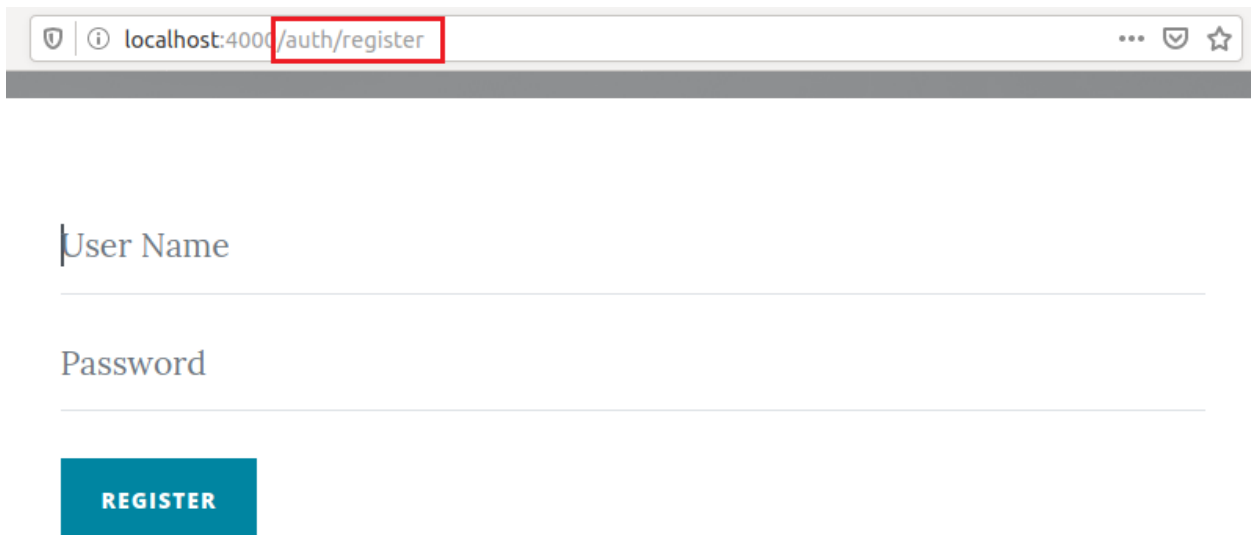
```
...
app.get('/auth/register', newUserController)
...
```

Giờ thì mọi thứ đã sẵn sàng, chúng ta chỉ còn tạo menu để người dùng có thể tới được màn hình đăng ký thành viên.

Trong `views/layouts/navbar.ejs`, thêm menu "New User"

```
<div class="collapse navbar-collapse" id="navbarResponsive">
  <ul class="navbar-nav ml-auto">
    <li class="nav-item">
      <a class="nav-link" href="/">Home</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="/posts/new">New Post</a>
    </li>
    <li class="nav-item">
      <a class="nav-link" href="/auth/register">New User</a>
    </li>
  </ul>
</div>
```

Xong, bây giờ bạn chạy ứng dụng và kiểm tra kết quả. Trên menu, bạn chọn "new user", nếu ứng dụng chuyển sang màn hình đăng ký như dưới đây là thành công.



The screenshot shows a web browser window with the address bar displaying 'localhost:4000/auth/register'. The page content includes a 'User Name' input field, a 'Password' input field, and a blue button labeled 'REGISTER'.

Hình 10.1: Giao diện màn hình đăng ký tài khoản thành viên

User Model

Phía trên, chúng ta mới chỉ hoàn thành phần giao diện và xử lý request, chứ chưa có lưu được vào database. Để lưu vào database, chúng ta sẽ tạo thêm UserModel (Nếu bạn quên cách tương tác với DB thì mời bạn đọc lại **phần 5: [Giới thiệu MongoDB](#)**).

Trong thư mục models, tạo thêm tệp User.js với nội dung như sau:

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;

const UserSchema = new Schema({
  username: String,
  password: String
});
// export model
const User = mongoose.model('User', UserSchema);
module.exports = User
```

Thông tin của user lưu vào trong database gồm user name và mật khẩu, hiện tại thì mình chỉ cần hai thông tin này là đủ. Nếu bạn thích thì có thể lưu thêm các thông tin khác: tuổi, giới tính, quốc tịch... Tất nhiên là bạn cũng cần phải update lại phần front-end để người dùng có thể nhập được những thông tin đó và gửi lên server.

Ok, tiếp theo thì chúng ta sẽ thiết lập route để khi người dùng nhấn nút "register" và lưu thông tin user vào database (cái này các bạn làm giống như cách lưu bài post vào database vậy).

```
...
const storeUserController = require('./controllers/storeUser')
app.post('/users/register', storeUserController)
...
```

Các bạn nhớ lúc trước đã tạo giao diện cho màn hình đăng ký user, đó là lý do tại sao chúng ta lại định nghĩa route như trên.

```
<form action="/users/register" method="POST" enctype="multipart/form-data">
```

Lúc này chương trình vẫn chưa chạy được đâu, vì chúng ta chưa viết mã cho phần controller xử lý.

Controller xử lý đăng ký user

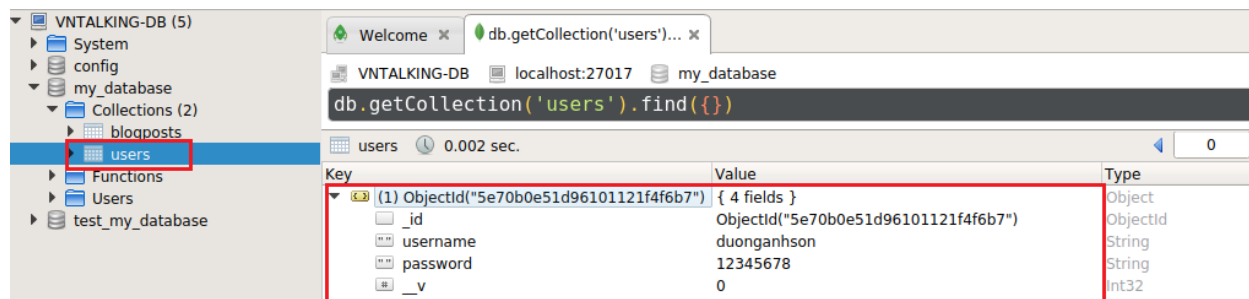
Trong thư mục controllers, tạo thêm tệp *storeUser.js*

```
const User = require('../models/User.js')

module.exports = (req, res) => {
  User.create(req.body, (error, user) => {
    res.redirect('/')
  })
}
```

Đã xong, giờ bạn chạy thử ứng dụng và đăng ký thông tin user xem đã lưu vào database chưa.

Dùng phần mềm robot3T để kiểm tra. Nếu xuất hiện thêm users collection và record mà bạn vừa mới đăng ký như dưới là ok nhé.



Hình 10.2: Thông tin tài khoản mới đăng ký trong mongoDB

Nhìn vào hình trên, bạn có thấy có gì đó "sai sai" không?

Đó là thông tin mật khẩu chưa hề mã hóa. Chúng ta cần phải mã hóa chúng trước khi lưu vào mongoDB.

Mã hóa mật khẩu

Thay vì viết một hàm mã hóa mật khẩu trong controller thì chúng ta sẽ ngay hook của mongoose để làm điều này.

Một hook hiểu nôm na giống như hàm middleware. Đầu tiên, chúng ta sẽ cài đặt một module [bcrypt](#) để mã hóa mật khẩu: `npm i --save bcrypt`

node.bcrypt.js

build passing dependencies up to date

A library to help you hash passwords.

You can read about [bcrypt](#) in [Wikipedia](#) as well as in the following article: [How To Safely Store A Password](#)

Install

> npm i bcrypt

Weekly Downloads

407,771

Version

License

Hình 10.3: bcrypt trên npm repository

Sau khi cài đặt xong thì bạn mở tệp models/User.js, khai báo và sử dụng bcrypt để mã hóa mật khẩu trước khi lưu vào database.

```
const mongoose = require('mongoose')
const Schema = mongoose.Schema;
const bcrypt = require('bcrypt')
```

```

const UserSchema = new Schema({
  username: String,
  password: String
});

UserSchema.pre('save', function (next) {
  const user = this
  bcrypt.hash(user.password, 10, (error, hash) => {
    user.password = hash
    next()
  })
})

// export model
const User = mongoose.model('User', UserSchema);
module.exports = User

```

Giải thích:

```

UserSchema.pre('save', function (next) {
  ...
}

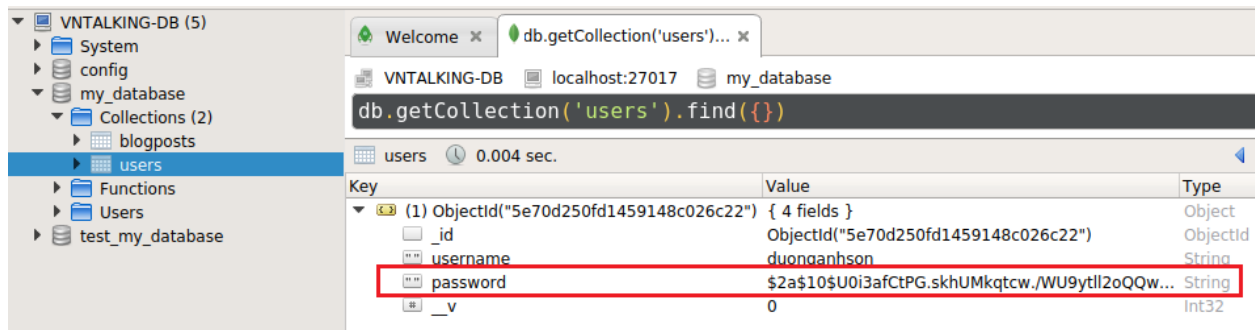
```

Hàm `.pre(...)` để thông báo cho Mongoose biết là sẽ cần thực hiện hàm trong tham số thứ 2 trước khi lưu vào Users collection. Điều này cho phép chúng ta thay đổi dữ liệu trước khi lưu vào DB.

Trong đó, hàm `bcrypt.hash(...)` để mã hóa cần hai giá trị đầu vào: một là mật khẩu dạng thô, hai là số lần mã hóa. Như ví dụ của mình ở trên là mã hóa 10 lần. Tất nhiên, bạn có thể để bao nhiêu lần cũng được, số này càng lớn thì càng lâu nhưng sẽ an toàn hơn.

Tham số cuối cùng là callback, được gọi sau khi quá trình mã hóa hoàn thành. Hàm `next()` có tác dụng là báo cho Mongoose chuyển sang hàm tiếp theo, tiếp tục tạo dữ liệu vào lưu vào DB (hàm `next()` mình đã đề cập trong phần giới thiệu [middleware](#), bạn có thể đọc lại để hiểu rõ hơn).

Kết quả thu được như hình bên dưới là ok.



Hình 10.4: Mật khẩu đã được mã hóa

Mongoose Validation

Chúng ta đã sử dụng mongoose để mã hóa mật khẩu trước khi lưu. Mongoose còn làm được nhiều hơn thế, trong đó có việc cần validate dữ liệu.

Với tính năng đăng ký thành viên, chúng ta không cho phép tạo 2 member trùng username, hay như mật khẩu để trống. Để mongoose tự kiểm tra và trả về lỗi nếu như vi phạm thì làm như sau.

Trong models/User.js, chúng ta sẽ update lại UserSchema:

```
...
const UserSchema = new Schema({
  username: {
    type: String,
    required: true,
    unique: true
  },
  password: {
    type: String,
    required: true
  }
});
...
```

Nhìn đoạn code trên, chúng ta cũng hiểu phần nào rồi đúng không? Đơn giản là chúng ta cấu hình scheme với từng field, quy định loại dữ liệu, quy tắc validation. Ví dụ như field username và password bắt buộc phải có dữ liệu, không được null thì thêm rule: *required: true*. Riêng với username thì không được trùng lặp, phải là duy nhất trong cơ sở dữ liệu thì thêm thuộc tính: *unique: true*.

Nếu một trong các rule bị vi phạm thì mongoose sẽ lưu dữ liệu đó vào trong DB và trả về một error. Để in ra error thì đơn giản gọi *console.log(error)* trong *controllers/storeUser.js*

```

module.exports = (req, res) => {
  User.create(req.body, (error, user) => {
    console.log(error)
    res.redirect('/')
  })
}

```

Ok, giờ thử kiểm tra bằng cách đăng ký một user mà đã đăng ký trước đó rồi xem sao. Trong màn hình console sẽ có error này ngay.

...

```
{ MongoError: E11000 duplicate key error collection: my_database.users index:
username_1 dup key: { username: "duonganhson" }
```

...

Đến đây thì cũng tạm được rồi đấy. Tuy nhiên, về mặt trải nghiệm người dùng thì quá tệ. Vì sao? Vì khi người dùng tiến hành đăng ký mà gặp lỗi mà họ không hề hay biết, họ đâu có màn hình console. Thông thường thì bạn nên hiển thị một message để thông báo cho họ biết là họ bị lỗi ở đâu. Nhưng thôi, để cho dễ, giờ nếu có lỗi thì chúng ta sẽ refresh lại trang đăng ký, còn nếu đăng ký thành công thì chuyển trang về trang chủ.

```

module.exports = (req, res) => {
  User.create(req.body, (error, user) => {
    if (error) {
      return res.redirect('/auth/register')
    }
    res.redirect('/')
  })
}

```

Như vậy là phần tạo thành viên đã xong. Khi đã có thông tin thành viên trong database rồi, công việc tiếp theo cho phép thành viên có thể đăng nhập.

Tạo tính năng đăng nhập

Mình sẽ nói qua về cách hoạt động của tính năng đăng nhập này. Trên menu sẽ có một nút "Login", khi người dùng click vào menu "login" và nhập thông tin username, password đã đăng ký trước đó. Nếu đăng nhập thành công, menu "login" sẽ chuyển thành menu "logout".

Chúng ta bắt đầu bằng việc tạo giao diện cho màn hình đăng nhập. Trong thư mục *views*, tạo mới một tệp đặt tên là *login.ejs*. Về cơ bản thì giao diện của màn hình đăng nhập hoàn toàn giống với màn hình đăng ký thành viên, chỉ khác là thay vì nút "register" thì là nút "login". Do vậy, mình sẽ clone mã nguồn của *register.ejs* sang.

Trong login.ejs, bạn thay đổi heading và text của nút:

```
...
<div class="col-lg-8 col-md-10 mx-auto">
  <div class="page-heading">
    <h1>Login</h1>
  </div>
</div>
...

<div class="form-group">
  <button type="submit" class="btn btn-primary">Login</button>
</div>
```

Trong form thì thay đổi action:

```
<form action="/users/login" method="POST" enctype="multipart/form-data">
```

Khi định nghĩa action mới thì cần phải định nghĩa cả trong controller nữa. Trong thư mục *controllers*, tạo thêm tệp đặt tên là *login.js* có nội dung như sau:

```
module.exports = (req, res) => {
  res.render('login')
}
```

Tương tự như các phần trước, chúng ta khai báo login controller trong *index.js*

```
...
const loginController = require('./controllers/login')
app.get('/auth/login', loginController);
...
```

Cuối cùng thì thêm một nút "Login" lên thành navbar. Chúng ta sửa trong *views/layouts/navbar.ejs*

```
<li class="nav-item">
  <a class="nav-link" href="/auth/login">Login</a>
</li>
```

Về cơ bản, chúng ta đã chuẩn bị xong các thủ tục cần thiết cho tính năng đăng nhập rồi, nào là route, controller, layout... Phần tiếp theo, khi người dùng nhập thông tin xong (user, password) và nhấn nút "login", chúng ta cần xử lý thông tin đăng nhập này. Những thông tin cần xử lý bao gồm:

- Kiểm tra user đó có tồn tại trong DB không?
- Password được nhập có đúng với user đó không?
- Nếu mọi thứ đều OK thì redirect về trang chủ.

Trong thư mục *controllers*, chúng ta tạo file mới, đặt tên là *loginUser.js*, có nội dung như sau:

```
const bcrypt = require('bcrypt')
const User = require('../models/User')

module.exports = (req, res) => {
  const { username, password } = req.body;
  User.findOne({ username: username }, (error, user) => {
    if (user) {
      bcrypt.compare(password, user.password, (error, same) => {
        if (same) { // if passwords match
          // store user session, will talk about it later
          res.redirect('/')
        } else {
          res.redirect('/auth/login')
        }
      })
    } else {
      res.redirect('/auth/login')
    }
  })
}
```

Mình sẽ giải thích đoạn code trên. Đầu tiên là chúng ta sẽ *'bcrypt'* module để mã hóa password. Tại sao lại cần mã hóa. Thực ra bạn hiểu đơn giản là lúc trước khi người dùng đăng ký tài khoản, chúng ta đã dùng module này để mã hóa mật khẩu rồi lưu vào cơ sở dữ liệu. Thì đến bây giờ, khi họ nhập mật khẩu, chúng ta cũng phải mã hóa mật khẩu đó rồi đem đi so sánh trong cơ sở dữ liệu. Nếu chuỗi mã hóa này mà giống nhau thì tức là họ nhập mật khẩu đúng.

Chúng ta dùng hàm *.findOne(...)* để query vào DB xem user có tồn tại không.

```
User.findOne({ username: username }, (error, user) => {
  if (user) {
    ...
  }
})
```

Nếu User đó tồn tại, chúng ta tiến hành so sánh mật khẩu. Ở đây, do mật khẩu là thông tin nhạy cảm, nên thay vì so sánh kiểu đơn giản như dùng toán tử *"==="* thì chúng ta dùng api *.compare(...)* của *bcrypt*.

```
bcrypt.compare(password, user.password, (error, same) => {
  if (same) {
    ...
  }
})
```

Cuối cùng thì khai báo controller này trong *index.js*

...

```
const loginUserController = require('./controllers/loginUser')
app.post('/users/login', loginUserController)
```

Lưu ý là phần định nghĩa route *"/users/login"* phải giống với url đã khai báo trong phần view của form login (trong *login.ejs*).

```
<form action="/users/login" method="POST" enctype="multipart/form-data">
```

...

```
  <div class="form-group">
    <button type="submit" class="btn btn-primary">Login</button>
  </div>
```

...

Vậy là tạm xong phần đăng nhập, bạn chạy chương trình và hưởng thụ thành quả nhé.

Tổng kết

Hoàn thành xong phần 10, chúng ta đã tạo xong tính năng đăng nhập, sử dụng "bcrypt" module để mã hóa và so sánh mật khẩu. Ngoài ra, chúng ta còn sử dụng mongoose để validate dữ liệu nhập vào trước khi lưu vào DB.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap10>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Xác thực với Express Sessions

Phần trước, chúng ta đã hoàn thành tính năng cho phép người dùng đăng ký và đăng nhập. Tuy nhiên, mỗi khi người dùng refresh lại trình duyệt thì họ lại phải đăng nhập lại. Lý do là chúng ta chưa có lưu thông tin phiên đăng nhập đó. Mỗi phiên đăng nhập gọi theo từ chuyên ngành là Sessions.

Sessions là cách chúng ta giữ phiên đăng nhập trên web, bằng cách lưu thông tin của họ trên trình duyệt. Ngoài ra, mỗi khi người dùng thực hiện một request, những thông tin đăng nhập này cũng được gửi tới server để kiểm tra. Do đó mà server biết được người nào thực hiện request đó, đã đăng nhập hay chưa.

Những thông tin được lưu trên trình duyệt gọi cookies.

Để làm phần này, chúng ta cài thêm [express-session](#) module.

```
npm install --save express-session
```

Tiếp thì mình import module này trong *index.js*

```
const expressSession = require('express-session');
```

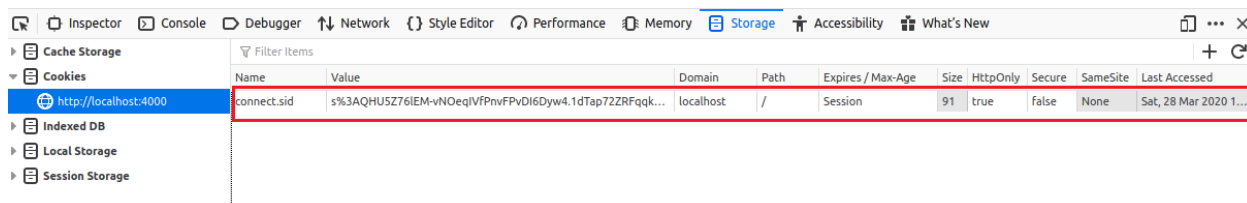
Và thêm đoạn code bên dưới:

```
...
app.use(expressSession({
  secret: 'keyboard cat'
}))
...
```

Ở đoạn code trên, chúng ta đăng ký *expressSession* middleware và truyền vào cấu hình cho middleware. *secret* là một string để *express-session* đăng ký và mã hóa các session ID được gửi bởi trình duyệt. Đây là chuỗi bất kỳ, bạn có thể thay đổi thoải mái, ở trên thì mình chọn là chuỗi *'keyboard cat'*.

Nào, bây giờ bạn mở trình duyệt và refresh lại trang web. Bạn có thể kiểm tra xem cookie bằng cách chuột phải chọn *Inspect Element(Q) -> Storage -> Cookies* (trình duyệt Firefox)

còn với Chrome thì *Developer Tools* -> *application*. Bạn sẽ nhìn thấy cookie của trang localhost được sinh ra.



Hình 11.1: Cookie được tạo ra bởi *express-session* module

Nếu bạn comment đoạn code liên quan đến *express-session*, sau đó refresh lại trình duyệt thì cookie sẽ không được tạo như bạn thấy ở trên.

Implementing User Sessions

Chúng ta bắt tay thực hiện implement cho tính năng tạo và lưu session đăng nhập của người dùng.

Mở *controllers/loginUser.js*, thêm đoạn code mình bôi đậm như bên dưới:

```
const bcrypt = require('bcrypt')
const User = require('../models/User')

module.exports = (req, res) => {
  const { username, password } = req.body;
  User.findOne({ username: username }, (error, user) => {
    if (user) {
      bcrypt.compare(password, user.password, (error, same) => {
        if (same) { // if passwords match
          req.session.userId = user._id
          res.redirect('/')
        } else {
          res.redirect('/auth/login')
        }
      })
    } else {
      res.redirect('/auth/login')
    }
  })
}
```

Chúng ta chỉ định *user_id* cho mỗi session. *express-session* module sẽ lưu thông tin này xuống cookie trình duyệt của người dùng, để mỗi khi người dùng gửi yêu cầu thì trình duyệt gửi cookie lại cho server kèm authenticated id. Đây là cách để server biết được người dùng đó đã đăng nhập hay chưa.

Để xem chính xác những thông tin trong một session object, chúng ta vào *controllers/home.js* và thêm dòng `console.log`:

```
module.exports = (req, res) => {
  BlogPost.find({}, function (error, posts) {
    console.log(req.session)
    console.log(posts);
    res.render('index', {
      blogposts: posts
    });
  })
}
```

Ok, giờ bạn thử vào trang web, và tiến hành đăng nhập thành công, kiểm tra màn hình console.

```
Session {
  cookie:
    { path: '/',
      _expires: null,
      originalMaxAge: null,
      httpOnly: true },
  userId: '5e70d250fd1459148c026c22' }
```

Thông tin `userId: '5e70d250fd1459148c026c22'` sẽ được chia sẻ giữa trình duyệt và server khi có bất kỳ request nào. Đây chính là thông tin để biết người dùng đã đăng nhập hay không.

Tiếp theo, để kiểm tra session id trước khi cho phép người dùng tạo bài post, chúng ta thêm đoạn code sau vào trong *controllers/newPost.js*

```
module.exports = (req, res) => {
  if (req.session.userId) {
    return res.render("create");
  }
  res.redirect('/auth/login')
}
```

Đoạn code có ý nghĩa là: kiểm tra xem session có chứa user id hay không? Nếu không có tức là request này của người dùng chưa đăng nhập -> tiến hành redirect sang màn hình login.

Protect một Pages nào đó với Authentication Middleware

Trong một ứng dụng web, bạn sẽ thấy là có một số trang chỉ có người dùng đã đăng nhập mới có thể truy cập. Ví dụ như trang tạo bài post mới, chỉ người nào đã đăng nhập thì mới được phép truy cập.



Mở rộng yêu cầu: Có rất nhiều trang web, người ta phân quyền tài khoản. Ví dụ như có trang thuộc admin panel thì phải tài khoản admin mới vào được, hay tài khoản thông thường thì chỉ được phép truy cập giới hạn tính năng không được thay đổi giá trị hệ thống... Cuối cùng thì khách truy cập thì chỉ được xem nội dung, mà không được sửa gì cả. Sau khi bạn hoàn thành xong phần 11 cuốn sách này, bạn thử thực hành làm tính năng phân quyền như mình gợi ý ở trên xem sao.

Đầu tiên, chúng ta cần tạo một custom middleware đặt tên là:
`/middleware/authMiddleware.js`

```
const User = require('../models/User')
module.exports = (req, res, next) => {
  User.findById(req.session.userId, (error, user) => {
    if (error || !user)
      return res.redirect('/')
    next()
  })
}
```

Trong middleware này, chúng ta sẽ query vào DB để tìm `userId`: `User.findById(req.session.userId...)`. Nếu kết quả trả về mà có tồn tại thì gọi hàm `next()` để chuyển sang middleware khác. Ngược lại, redirect về trang chủ.

Tiếp theo, chúng ta sẽ import middleware này trong `index.js`

```
...
const authMiddleware = require('./middleware/authMiddleware')
...
```

Để áp dụng middleware vào route khi tạo bài post thì cần đặt middleware trước `newPostController` là được.

```
...
app.get('/posts/new', authMiddleware, newPostController)
...
```

Chúng ta làm tương tự khi lưu bài post vào DB.

```
app.post('/posts/store', authMiddleware, storePostController)
```

Để kiểm tra xem đoạn code đã hoạt động đúng chưa, bạn cần xóa cookie trình duyệt, sau đó thử chọn menu "new post", nếu web mà redirect về trang chủ là đúng. Sau đó bạn login và làm lại, nếu vào được trang "new post" là được.

Tiếp tục nhé!

Hiện tại, khi một người dùng đã đăng nhập thành công, họ vẫn nhìn thấy menu "login" và "new user". Điều này có vẻ sai sai đúng không? Bởi vì họ đã login thì không cần login nữa. Tương tự, họ đã là user rồi, cần gì phải đăng ký nữa???

Tương tự với trường hợp của menu "new post". Menu này chỉ dành cho người dùng đã đăng nhập, còn với khách thì không hiện ra.

Giờ chúng ta sẽ xử lý trường hợp này.

Trong thư mục middleware, tạo thêm một middleware đặt tên là *redirectIfAuthenticatedMiddleware.js*

```
module.exports = (req, res, next) => {
  if (req.session.userId) {
    return res.redirect('/') // if user logged in, redirect to home page
  }
  next()
}
```

Sau đó thì import middleware trong *index.js*

```
...
const redirectIfAuthenticatedMiddleware = require('./middleware/redirectIfAuthenticatedMiddleware')
...
```

Áp dụng cho 4 routes sau:

```
app.get('/auth/register', redirectIfAuthenticatedMiddleware, newUserController)
app.post('/users/register', redirectIfAuthenticatedMiddleware, storeUserController)
app.get('/auth/login', redirectIfAuthenticatedMiddleware, loginController)
app.post('/users/login', redirectIfAuthenticatedMiddleware, loginUserController)
```

Giờ khi bạn đã đăng nhập thành công mà cố tình nhấp vào menu "login", trang web sẽ điều hướng về trang chủ.

Nhưng vẫn chưa được. Chúng ta sẽ tiếp tục xử lý phần giao diện, cần phải ẩn các menu "Login" và "New User" khi người dùng đã đăng nhập.

Để làm điều này, trong *index.js*

```
...  
global.loggedIn = null;  
app.use("*", (req, res, next) => {  
  loggedIn = req.session.userId;  
  next()  
});  
...
```

Chúng ta khai báo một biến *loggedIn* kiểu global, mục đích là có thể truy cập biến này trong các file EJS.

Với khai báo *app.use("*", (req, res, next) => ...)*, với toán tử "*" tức là áp dụng cho mọi request, và chúng ta sẽ gán *UserId* cho biến *loggedIn*

Tiếp theo, chúng ta sẽ sửa *Navbar* tại *views/layout/navbar.ejs*. Thêm kiểm tra điều kiện cho các menu "Login", "new post" và "new user".

```
<div class="collapse navbar-collapse" id="navbarResponsive">  
  <ul class="navbar-nav ml-auto">  
    <li class="nav-item">  
      <a class="nav-link" href="/">Home</a>  
    </li>  
    <% if(loggedIn) { %>  
    <li class="nav-item">  
      <a class="nav-link" href="/posts/new">New Post</a>  
    </li>  
    <% } %>  
    <% if(!loggedIn) { %>  
    <li class="nav-item">  
      <a class="nav-link" href="/auth/login">Login</a>  
    </li>  
    <li class="nav-item">  
      <a class="nav-link" href="/auth/register">New User</a>  
    </li>  
    <% } %>  
  </ul>  
</div>
```

Giờ bạn thử chạy ứng dụng và kiểm tra thành quả nhé.

User Logout

Hiện tại thì người dùng chưa có cách nào để có thể logout tài khoản. Để làm điều này thì về cơ bản là bạn chỉ cần gọi hàm *session.destroy()* là được. Cách thực hiện và tạo giao

diện người dùng hoàn toàn tương tự như các phần trước. Mình sẽ trình bày rất nhanh thôi.

Trong `views/layout/navbar.ejs`

```
<% if(loggedIn) { %>
  <li class="nav-item">
    <a class="nav-link" href="/posts/new">New Post</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="/auth/logout">Log out</a>
  </li>
<% } %>
```

Tạo thêm controller cho logout: `/controllers/logout.js`

```
module.exports = (req, res) => {
  req.session.destroy(() => {
    res.redirect('/')
  })
}
```

Với `req.session.destroy()`, chúng xóa tất cả dữ liệu liên quan session, kể cả session user id, sau khi xóa xong thì redirect về trang chủ.

Trong `index.js`

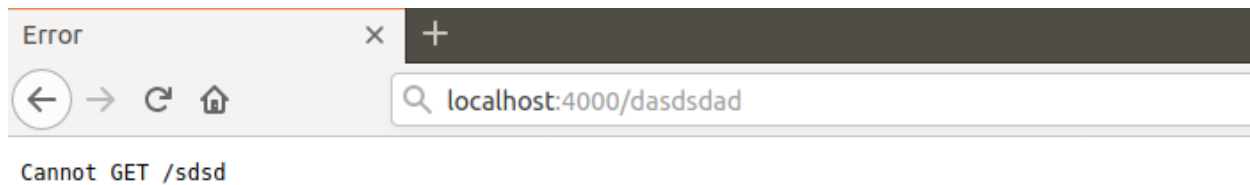
```
...
const logoutController = require('./controllers/logout')
...

app.get('/auth/logout', logoutController)
```

Đã xong, giờ bạn chạy ứng dụng và kiểm tra tính năng logout xem thế nào nhé.

Tạo trang 404

Có trường hợp khi người dùng truy cập vào một URL không tồn tại. Ví dụ như: `localhost:4000/dasdsdad`. Hiện tại thì chúng ta chưa có xử lý gì cả, nên sẽ bị crash như bên dưới đây.



Hình 11.1: Ứng dụng bị crash khi vào một URL bất kỳ

Điều này không mang trải nghiệm tốt cho người dùng. Với những trường hợp này, chúng ta cần tạo trang 404, mục đích là thông báo cho người dùng biết là URL đó không tồn tại.

Trong thư mục `views`, tạo thêm file đặt tên là `notfound.ejs`

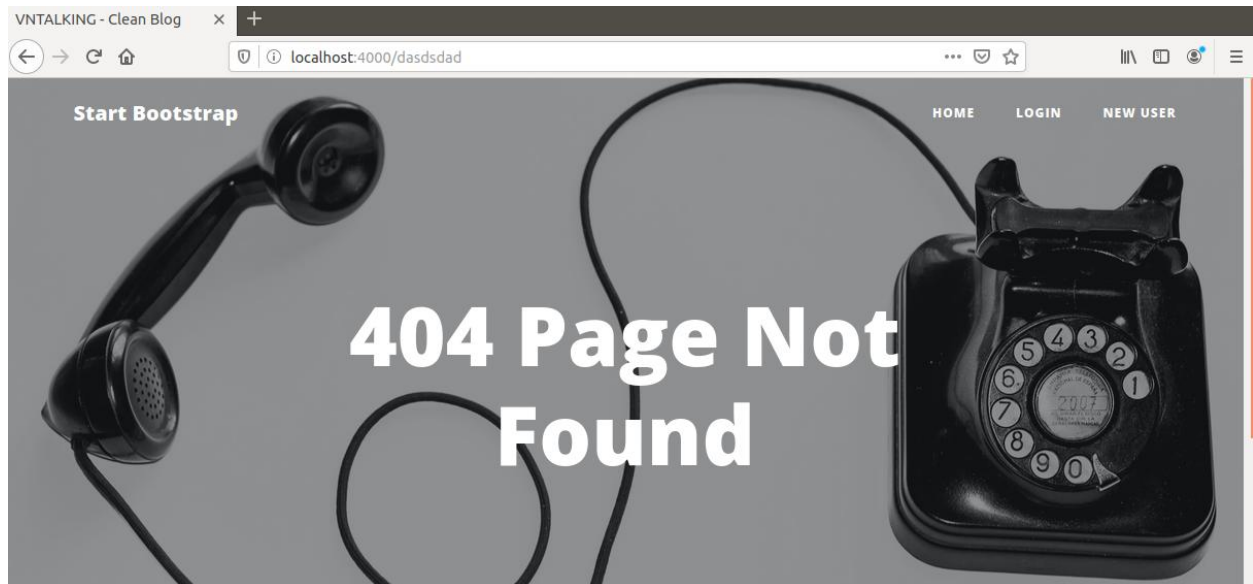
```
<body>

  <%- include('layouts/navbar'); -%>
  <!-- Page Header -->
  <header class="masthead" style="background-image: url('/img/contact-
bg.jpg')">
    <div class="overlay"></div>
    <div class="container">
      <div class="row">
        <div class="col-lg-8 col-md-10 mx-auto">
          <div class="page-heading">
            <h1>404 Page Not Found</h1>
          </div>
        </div>
      </div>
    </div>
  </header>
  <hr>
  <%- include('layouts/footer'); -%>
  <%- include('layouts/scripts'); -%>
</body>
```

Trong `index.js`, chúng ta đăng ký route cho not found.

```
app.get('/', homeController)
app.get('/posts/new', authMiddleware, newPostController)
app.get('/post/:id', getPostController)
app.post('/posts/store', authMiddleware, storePostController)
app.get('/auth/register', redirectIfAuthenticatedMiddleware, newUserController)
app.post('/users/register', redirectIfAuthenticatedMiddleware, storeUserController)
app.get('/auth/login', redirectIfAuthenticatedMiddleware, loginController);
app.post('/users/login', redirectIfAuthenticatedMiddleware, loginUserController)
app.get('/auth/logout', logoutController)
app.use((req, res) => res.render('notfound'));
```

Đoạn code trên có ý nghĩa là, nếu các request mà không thuộc vào route nào thì nó sẽ nhảy vào route cuối cùng (chính là trường hợp not found).



Hình 11.3 : Giao diện trang 404

Tổng kết

Hoàn thành xong phần 11, chúng ta đã hoàn thiện hơn tính năng đăng nhập của người dùng. Mỗi phiên đăng nhập sẽ được lưu vào cookie của trình duyệt, từ đó server có thể xác thực mỗi request từ trình duyệt xem có phải người đã đăng nhập hay chưa.

Các bạn có thể tham khảo mã nguồn của phần này tại đây:

<https://github.com/vntalking/nodejs-express-mongodb-co-ban/tree/master/chap11>

Nếu bạn có bất kỳ thắc mắc hoặc chỗ nào chưa hiểu, đừng ngại liên hệ với mình qua support@vntalking.com.

Triển khai web app lên server thật

Đây là công đoạn cuối cùng của dự án, khi tất cả mã nguồn đã sẵn sàng, bạn tiến hành triển khai ứng dụng web lên server thật, chính thức ra mắt sản phẩm cho người dùng thật.

Về nguyên tắc thì việc triển khai lên server thật cũng không khác là mấy so với việc bạn cài đặt trên localhost. Điểm khác biệt cơ bản là:

- Server sử dụng public IP, ai cũng có thể truy cập được bất kể lúc nào.
- Cần phần mềm monitor ứng dụng web. Ở đây mình sử dụng [PM2](#).

Ở mức cơ bản chỉ có vậy, còn tất nhiên, khi hệ thống lớn và có lượng truy cập nhiều thì sẽ cần phải có đội quản trị server chuyên nghiệp, tối ưu và xử lý sự cố...

Trong các nhà cung cấp server, mình khuyến khích sử dụng [Vultr](#). Đây là nhà cung cấp server VPS nổi tiếng, giá rẻ mà chất lượng đảm bảo. Đặc biệt là khi bạn đăng ký tài khoản mới, được tặng tới 100\$ để dùng thử tất cả các dịch vụ của Vultr.

Cài đặt server

Sau khi bạn tạo một Cloud VPS mới, thường thì nó sẽ ở trạng thái mặc định, tức là chưa có thêm bất cứ ứng dụng nào khác được cài thêm.

Ngoài ra, tùy nhà cung cấp VPS mà bạn sẽ được cấp tài khoản root hoặc tài khoản user bình thường nhưng có quyền sudo. Trong trường hợp bạn có tài khoản root thì sẽ không cần chạy những dòng lệnh bắt đầu bằng sudo.

Bạn sẽ truy cập vào server thông qua giao thức SSH và sẽ cấu hình server bằng dòng lệnh trong một cửa sổ terminal.



Lưu ý: Khi bạn tạo VPS server, nếu bạn theo bài hướng dẫn trong cuốn sách này thì nên chọn Ubuntu OS. Mọi hướng dẫn trong sách này, mình

```
► ssh ccm@123.36.109.27
ccm@123.36.109.27's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0-79-generic x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:    https://landscape.canonical.com
 * Support:       https://ubuntu.com/advantage

42 packages can be updated.
8 updates are security updates.

*** System restart required ***
Last login: Wed Sep 20 12:35:29 2017 from 138.69.52.190
ccm@ccm:~$
```

Hình 12.1: màn hình ssh khi kết nối thành công

1. Cài đặt NGINX và Git

```
sudo apt install -y nginx git
```

NGINX sẽ đóng vai trò *reversed proxy* và *static file server* để tiếp nhận request thông qua port mặc định 80 (http) và 443 (https). **Git** dùng để lấy source code của app để tiến hành build và deploy.

2. Cài đặt NodeJS và mongoDB

Việc cài đặt hai thành phần này trên server hoàn toàn tương tự như trên localhost. Bạn tham khảo lại cách cài đặt trong [phần 1: cài đặt Node.js](#) và [phần 5: cài đặt mongoDB](#)

Sau khi cài đặt xong thì khởi động dịch vụ MongoDB bằng 2 lệnh:

```
# Khởi động mongod service
sudo systemctl start mongod
```

```
# Bật chức năng tự chạy khi restart Ubuntu
sudo systemctl enable mongod
```

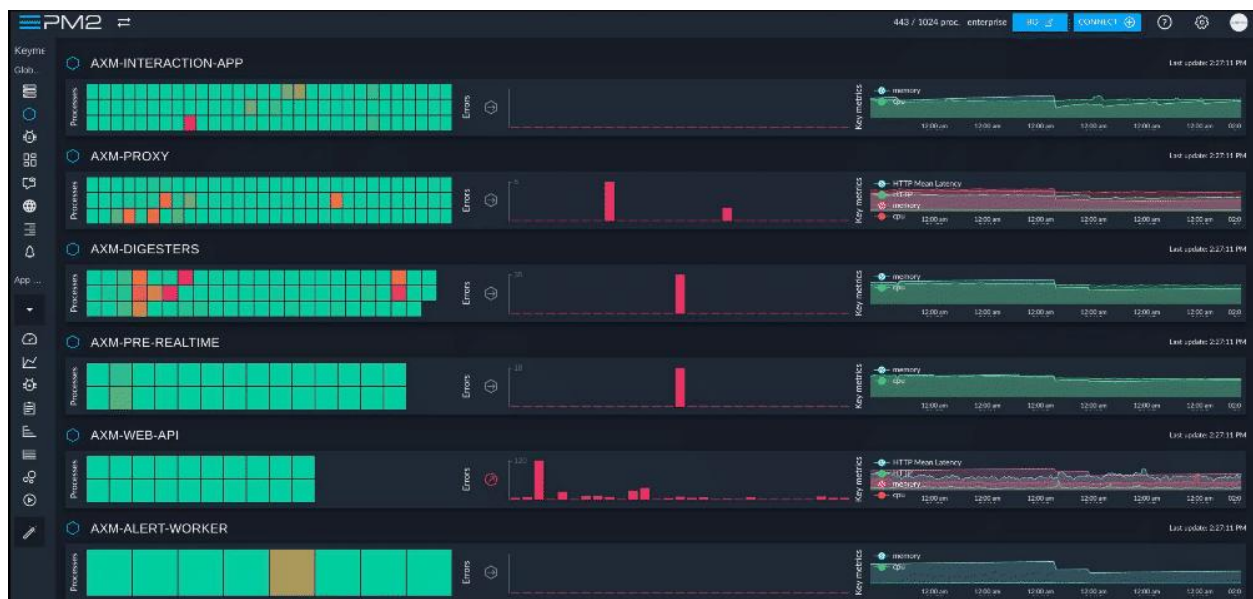

3. Cài đặt PM2

PM2 là một trình quản lý các process (tiến trình) dành cho các ứng dụng Nodejs, được viết bằng chính Nodejs và Shell. Bạn có thể giữ cho các process của server còn sống và reload/restart với zero downtime. Và như bạn biết, zero downtime là cái mà bất kể ứng dụng web nào cũng muốn đạt tới.

Cài đặt PM2 như bình thường với dòng lệnh:

```
sudo npm install -g pm2
```

Chúng ta sử dụng PM2 qua các dòng lệnh trên command line đồng thời có thể quản lý bằng giao diện người dùng thông qua Key Metrics.



Hình 12.2: Giao diện quản lý của PM2

Đưa sourcecode Node.js lên VPS

Đầu tiên ta phải cài đặt git cho VPS với cú pháp:

```
$ sudo apt-get install git
```

Tiếp theo bạn cần phải clone sourcecode của bạn về. Ở đây mình sẽ lưu vào trong thư mục `/var/apps/demo_vntalking` nên các bạn hãy tạo thư mục theo đường dẫn như mình nhé, hoặc lưu ở một nơi nào đó khác tùy các bạn.

```
$ git clone https://github.com/vntalking/nodejs-express-mongodb-co-ban-final.git /var/apps/demo_vntalking
```

Các bạn nhớ thay đường dẫn của các bạn vào nhé.

Sau khi clone về các bạn có thể chạy dòng lệnh sau để chạy ứng dụng:

```
$ node index.js
```

Quản lý ứng dụng Node.js bằng PM2

Như mình đã giới thiệu ở trên, sau khi PM2 đã cài đặt, chúng ta sẽ dùng PM2 để quản lý ứng dụng web.

Để chạy một ứng dụng nodejs bằng PM2 ta thực hiện theo cú pháp:

```
$ pm2 start index.js
```

Sau khi chạy lệnh trên các bạn để ý rằng ứng dụng của chúng ta đã chạy và bạn hoàn toàn có thể thực hiện các lệnh khác và không ảnh hưởng gì đến nó. Hoặc các bạn cũng có thể chạy thêm nhiều ứng dụng khác với cú pháp tương tự như trên.

```
[PM2] Spawning PM2 daemon with pm2_home=/home/sonduong/.pm2
[PM2] PM2 Successfully daemonized
[PM2] Starting /home/sonduong/VNTALKING/Nodejs-basic-ebook/nodejs-express-mongodb-co-ban/chap11/index.js in fork_mode (1 instance)
[PM2] Done.
```

| id | name | mode | ♾ | status | cpu | memory |
|----|-------|------|---|--------|-----|--------|
| 0 | index | fork | 0 | online | 0% | 16.1mb |

Sau khi chạy pm2, bạn hoàn toàn có thể vào trang web của mình tại địa chỉ theo cú pháp: ***http://<địa chỉ IP server>:<port>***

Ở ví dụ của mình: port là **4000**

PM2 có rất nhiều câu lệnh để quản lý ứng dụng và monitor các thông số server. Dưới đây là một số câu lệnh phổ biến:

#Dừng một ứng dụng

```
$ pm2 stop app_name_or_id
```

#Khởi động lại một ứng dụng

```
$ pm2 restart app_name_or_id
```

#em danh sách các ứng dụng đang chạy

```
$ pm2 list
```

#Xem các thông tin của ứng dụng

```
$ pm2 info app_name_or_id
```

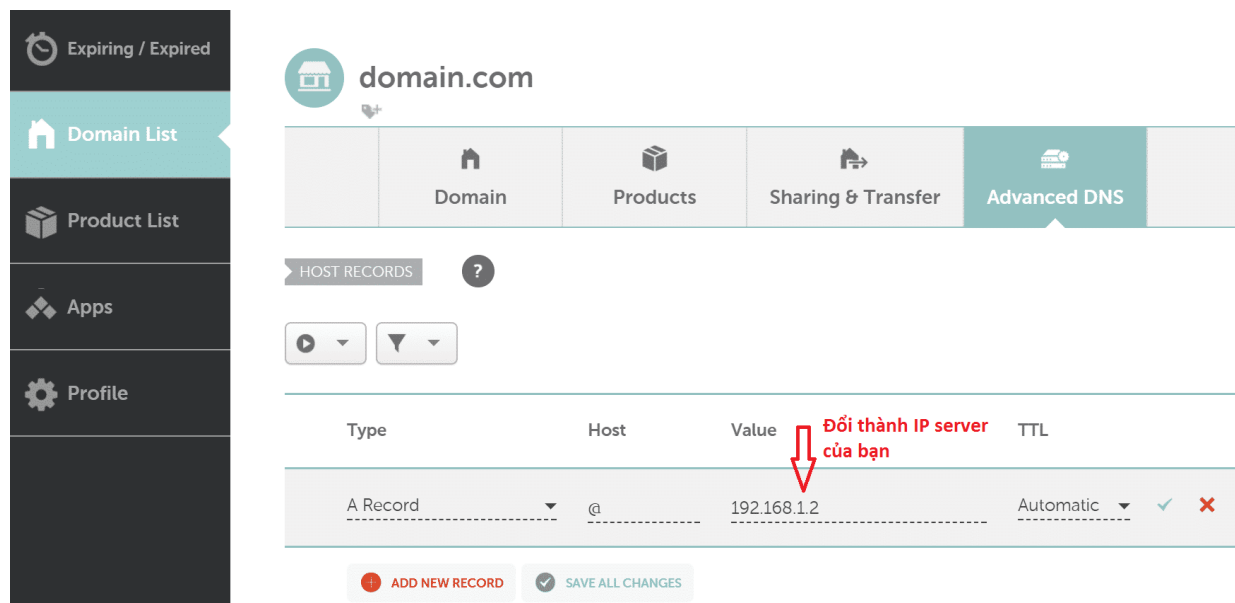
```
#Xem mức CPU và bộ nhớ sử dụng
$ pm2 monit
```

```
#Xem log của ứng dụng
$ pm2 logs app_name_or_id
```

Kết nối domain vào vps

Sau khi deploy, web của bạn sẽ vẫn ở định dạng `<IP server>:<port>`, để gán domain vào server, chúng ta cần một tên miền, và quyền chỉnh sửa DNS của tên miền đó.

Mình mua tên miền tại [NameCheap](#), nên sẽ chỉnh sửa tại trang web của nhà cung cấp tên miền này. Hoặc bạn có thể sử dụng dịch vụ DNS trung gian như Cloudflare.



Hình 12. 3: Thêm A record trong NameCheap

Sau khi chuyển, cần chờ tầm 1 giờ để hệ thống cập nhật. Sau 1 giờ, bạn có thể vào web theo cú pháp: **`http://<domain_của_bạn>:4000`**

Như bạn có thể thấy, chúng ta vẫn còn hiển thị port, phần tiếp theo mình sẽ dùng Nginx loại bỏ port khỏi domain.

Cấu hình Nginx Reverse Proxy Server

Mở file cấu hình nginx

```
sudo nano /etc/nginx/sites-available/default
```

Xóa toàn bộ nội dung trong đó, và thêm cấu hình của mình vào:

```
server {  
    listen 80;  
  
    server_name example.com;  
  
    location / {  
        proxy_pass http://localhost:4000;  
        proxy_http_version 1.1;  
        proxy_set_header Upgrade $http_upgrade;  
        proxy_set_header Connection 'upgrade';  
        proxy_set_header Host $host;  
        proxy_cache_bypass $http_upgrade;  
    }  
}
```

Với example.com là domain, thay localhost và port 4000 trùng với thông tin trên server của bạn. Sau đó khởi động lại Nginx

```
sudo service nginx restart
```

Giờ đây bạn vào ứng dụng web của mình chỉ với domain mà không cần port:

http://<domain>

Chào tạm biệt

Khi đọc đến dòng chữ này, mình xin chúc mừng vì bạn đã hoàn thành hành trình chinh phục Node.js.

Chúng ta đã trải qua khá nhiều nội dung đủ để trang bị cho bạn kỹ năng cần thiết để tự tạo cho riêng mình một ứng dụng Node.js kết hợp với Express và MongoDB.

Hi vọng bạn sẽ thích cuốn sách này, và muốn tìm hiểu thêm về thế giới lập trình.

Trong quá trình biên soạn cuốn sách sẽ không tránh khỏi những thiếu sót. Mình rất mong nhận được phản hồi từ bạn, hãy gửi email tới support@vntalking.com.

Toàn bộ mã nguồn được mình up lên github: <https://github.com/vntalking/nodejs-express-mongodb-co-ban>

Cuốn sách là một phần trong dự án học lập trình của VNTALKING. Mong bạn ủng hộ website học lập trình cùng bọn mình tại: <https://vntalking.com>

Hẹn gặp lại bạn ở cuốn sách tiếp theo.

Tài liệu tham khảo

1. Beginning Node.js, Express & MongoDB Development 2019 - Greg Lim
2. Node.js official guides - <https://nodejs.org/en/docs/guides/>
3. Mongoose document - <https://mongoosejs.com/docs/4.x/docs/guide.html>
4. Express documents - <https://expressjs.com/en/api.html>
5. Hướng dẫn: Cài đặt hoàn chỉnh Nodejs app lên VPS - <https://int3ractive.com/2017/09/huong-dan-cai-dat-nodejs-app-len-vps-phan-1.html>
6. Deploy production Node.js với PM2 và Nginx - <https://blog.duyet.net/2016/04/deploy-production-nodejs-pm2-nginx.html>
7. Deploy ứng dụng Node.js lên VPS - <https://www.tuankhaiit.com/deploy-ung-dung-node-js-len-vps/>