BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC KHOA HỌC

KHOA CÔNG NGHỆ THÔNG TIN

BÀI TIỂU LUẬN

MÔN : LẬP TRÌNH MẠNG

Đề Tài : Tìm hiểu về node.js

Giáo viên hướng dẫn : **Nguyễn Quang Hưng**

Sinh viên thực hiện : Nguyễn Quang Hà

TÌM HIỂU VỀ NODE.JS

I. Giới thiệu.

Node.js là một hệ thống phần mềm được thiết kế để viết các ứng dụng internet có khả năng mở rộng, đặc biệt là máy chủ web. Chương trình được viết bằng JavaScript, sử dụng kỹ thật điều khển theo sự kiện, nhập/xuất không đồng bộ để tối tiểu tổng chi phí và tối đại khả năng mở rộng. Node.js bao gồm có V8 JavaScript engine của Google, libUV, và vài thư viện khác.

Node.js được tạo bởi Ryan Dahl từ năm 2009, và phát triển dưới sự bảo trợ của Joyent.

Mục tiêu ban đầu của Dahl là làm cho trang web có khả năng push như trong một số ứng dụng web như Gmail. Sau khi thử với vài ngôn ngữ Dahl chọn Javascript vì một API Nhập/Xuất không đầy đủ. Điều này cho phép anh có thể định nghĩa một quy ước Nhập/Xuất điểu khiển theo sự kiện, non-blocking.

Vài môi trường tương tự được viết trong các ngôn ngữ khác bao gồm Twisted cho Python, Perl Object Environment cho Perl, libevent cho C và EventMachine cho Ruby. Khác với hầu hết các chương trình Javascript, Nodejs không chạy trên một trình duyệt mà chạy trên Server. Node.js sử dụng nhiều chi tiết kỹ thuật của CommonJS. Nó cung cấp một môi trường REPL cho kiểm thử tương tác.

Node.js được *InfoWorld* bình chọn là "Công nghệ của năm" năm 2012.

Để bắt đầu dùng Node.js, bạn phải hiểu sự khác nhau giữa Node.js với các môi trường truyền thống chạy trên server (server side) phổ biến như PHP, Python, Ruby, etc

1. Lập trình không đồng bộ

Là lợi thế nếu bạn đã quen thuộc với các phương pháp lập trình không đồng bộ. Tất cả các hàm trong Node.js là không đồng bộ. Do đó, tất cả chạy như các block thread thông thường thay vì chạy nền. Đây là điều quan trọng nhất để nhớ về Node.js. Ví dụ, nếu bạn đang đọc một tập tin trên hệ thống tập tin, bạn phải chỉ định một chức năng gọi lại đó là thực hiện khi đã hoàn thành các hoạt động đọc.

2. Bạn phải làm tất cả

Node.js chỉ là môi trường – điều đó có nghĩa là bạn phải tự làm tất cả. Đó không phải là một server http ngầm định hoặc là bất cứ server nào khác. Điều này có thể là hơi khó hiểu với người mới, nhưng thành công thực sự của nó là đưa lại một hiệu năng đáng kinh ngạc . Một scrits có thể điều phối mọi kết nối với các client. Điều này làm sử dụng ít tài nguyên đưa đến một hiệu quả rất cao. Ví dụ sau về một ứng dụng Node.js

```
var i, a, b, c, max;
max = 1000000000;
var d = Date.now();
for (i = 0; i < max; i++) {
    a = 1234 + 5678 + i;
    b = 1234 * 5678 + i;
    c = 1234 / 2 + i;
}
console.log(Date.now() - d);
```

Và đây là tương đương với mã PHP:

```
$a = null;
$b = null;
$c = null;
$i = null;
$max = 1000000000;
$start = microtime(true);
for ($i = 0; $i < $max; $i++) {
    $a = 1234 + 5678 + $i;
    $b = 1234 * 5678 + $i;
    $c = 1234 / 2 + $i;
}
var_dump(microtime(true) - $start);
```

Và giờ ta xem chấm điểm benchmark cho hai đoạn code trên khi chạy trên hai môi trường khác nhau:

Number of iterations	Node.js	PHP
100	2.00	0.14
10’000	3.00	10.53

1’000’000	15.00	1119.24
10’000’000	143.00	10621.46
1’000’000’000	11118.00	1036272.19

Tôi thực hiện chạy hai đoạn code trên từ command line (console command) nên không có trễ thao tác thực thi. Tôi chạy từng thử nghiệm 10 lần và lấy kết quả trung bình. PHP nhanh hơn trong các lần chạy với số lượng nhỏ vòng lặp. Nhưng vấn đề thay đổi khi số lượng vòng lặp tăng lên, số lần xử lý tăng lên thì PHP chạy chậm hơn rất nhiều trong khi Node.js có tốc độ đáng kinh ngạc. Sau tất cả thao tác, PHP chậm hơn 93% so với Node.js.

3. Module

Node.js sử dụng một kiến trúc mô-đun để đơn giản hóa việc tạo ra các ứng dụng phức tạp. Mô-đun giống như các thư viện trong C, hoặc các đơn vị trong Pascal. Mỗi module có chứa một tập hợp các chức năng liên quan đến "đối tượng" của các mô-đun. Ví dụ, các mô-đun http chứa các chức năng cụ thể cho HTTP. Node.js cung cấp một vài mô-đun cơ bản để giúp bạn truy cập các tập tin trên hệ thống tập tin, tạo ra trình điều khiển server HTTP và TCP / UDP và thực hiện các chức năng hữu ích khác.

Để gọi một modul thật dễ dàng, chỉ cần gọi hàm require() như sau

```
Var http = require(‘http’);
```

Hàm require() trả về tham chiếu đến các module quy định. Trong trường hợp của mã này, một tham chiếu đến các module http được lưu trữ trong biến http. Trong đoạn code trên, ta đã truyền tên của module vào trong hàm require(). Việc này chỉ định cho Node.js tìm trong thư mục node_modules module tương ứng để thực hiện. Nếu Node không thấy module tương ứng trong thư mục thì nó sẽ tìm trên global module cache. Bạn cũng có thể chỉ định một module qua một file vật lý qua đường dẫn tương đối hay tuyệt đối như sau:

```
var myModule = require('./myModule.js');
```

Module được đóng gói từng phần mã. Đoạn mã nằm trong một mô-đun chủ yếu là private - có nghĩa là các chức năng và biến được định nghĩa trong họ chỉ có thể truy cập từ bên trong của các mô-đun. Tuy nhiên, bạn có thể tiếp xúc với chức năng và / hoặc các biến được sử dụng từ bên ngoài của mô-đun. Để làm như vậy, phải sử dụng các đối tượng export với các thuộc tính và phương thức của nó với từng phần mã mà bạn muốn gọi từ bên ngoài. Hãy xem xét các modul ví dụ sau đây:

```
var PI = Math.PI;
exports.area = function (r) {
  return PI * r * r;
};
exports.circumference = function (r) {
  return 2 * PI * r;
};
```

Trong ví dụ này PI là biến private và chỉ được sử dụng bên trong đoạn mã, trong đó có hai hàm area() và circumference() được từ khóa exports chỉ định thì sẽ có thể truy cập được từ bên ngoài.

4. Global Scope

Node là một môi trường chạy javascrip với google V8 engine do đó hỗ trợ chạy được ở server side. Do đó bạn cũng nên tuân thủ các kinh nghiệm mà bạn có trong lập trình với các ứng dụng client-side. Ví dụ khi tạo các biến global trong Node không phải lúc nào cũng có thể tạo. Nhưng bạn có thể tạo dễ dàng các biến hoặc hàm global với cách bỏ từ khóa var trước các biến như sau:

```
globalVariable = 1;
globalFunction = function () { ... };
```

Nhưng các biến global nên tránh sử dụng, và xin nhớ cẩn thận rằng khi khai báo biến thì dùng từ khóa var để thực hiện.

5. Cộng đồng

Cộng đồng phát triển Node.js chủ yếu tập trung ở hai nhóm google : nodejs và nodejs-dev, một kênh IRC là #node.js trên mạng freenode. Có một hội thảo về Node.js là NodeConf được tổ chức thường niên.

Hiện nay Node.js được sử dụng bởi nhiều công ty trong đó có Linkedin, Microsoft, Yahoo! và Walmart.

II. Installation

1. Installing Node.js .

Hiển nhiên là bạn phải học cách cài đặt node trước khi muốn viết và chạy bất cứ ứng dụng nào trên nền node. Cài đặt node thì rất là đơn giản, bạn là người sử dụng window hay linux

thì trên website nodejs.org đều đã có những bộ cài tương ứng, bạn chỉ cần download về và cài đặt như thông thường.

Với linux thì bạn sử dụng package manager, bật cửa sổ terminal và type:

```
sudo apt-get update
sudo apt-get install node
```

Hoặc:

```
sudo aptitude update
sudo aptitude install node
```

Bạn có thể cần thêm Node.js vào danh sách mã nguồn bằng lệnh sau:

```
sudo echo deb http://ftp.us.debian.org/debian/ sid main > /etc/apt/
sources.list.d/sid.list
```

Cần trọng khi cài sid packages trên những hệ thống cũ hơn có thể làm hệ thống của bạn bị ảnh hưởng, hay cẩn thận và remove /etc/apt/sources.list.d/sid.list sau khi bạn cài xong Node.

2. Installing New Modules

Node.js có một ứng dụng quản lý packages, đó là Node Packgate Manager (NPM). Ứng dụng này tự động được cài đặt khi bạn cài Node.js và bạn dùng NPM để cài đặt các module khác. Để cài đặt một module, bạn mở cửa sổ command line của nodejs ra, vào đường dẫn tương ứng và nhập lệnh:

```
npm install module_name
```

Không phụ thuộc vào hệ điều hành bạn dùng, lệnh trên sẽ cài module mà bạn mong muốn chỉ định.

III. Các objects cơ bản.

1. Global Objects (đối tượng toàn cục).

Như chúng ta đã biết, hệ thống mô-đun của node không khuyến khích việc sử dụng biến toàn cục, tuy nhiên node cung cấp một globals quan trọng để sử dụng. Việc đầu tiên và quan trọng nhất là tiến trình global, cho thấy nhiều thao tác như quá trình truyền tín hiệu, xuất cảnh, proccess id (pid), và nhiều hơn nữa. Globals khác, chẳng hạn như console obiects được cung cấp cho những người sử dụng để viết JavaScript cho trình duyệt web.

a. Console

Các console obiects sử dụng một số lệnh được sử dụng để xuất thông tin để stdout hoặc stderr. Chúng là các lệnh như:

console.log ([data], [...])

Phương pháp console obiects được sử dụng thường xuyên nhất là `console.log()`, mà chỉ đơn giản là viết cho stdout và gắn một nguồn cấp dữ liệu dòng (\ n

```
console.log (wahoo ');
// => Wahoo

console.log ({foo: 'bar'});
// => [Object Object]
```

Còn một lệnh có chức năng như `console.log()` đó là `console.info()`.

console.error ([data], [...])

Giống hệt nhau để `console.log()`, tuy nhiên viết cho *stderr*.

```
console.error ('kết nối cơ sở dữ liệu không thành công ');
```

Còn một lệnh có chức năng như `console.error()` đó là `console.warn()`.

console.dir (obj)

Sử dụng phương pháp `inspect()` của mô-đun *sys* khá-in các đối tượng đến *stdout*.

```
console.dir ({foo: 'bar'});
/ / => {Foo: 'bar'}
```

console.assert (expression, [message])

Nếu `expression` bị đánh giá là có giá trị là false thì AssertionError sẽ đưa ra `message` được cho.

```
console.assert (connected, 'Cơ sở dữ liệu kết nối không thành công ');
```

console.time(label)

Đánh dấu thời gian bắt đầu.

console.timeEnd (label)

Thời gian kết thúc, được ghi vào đầu ra.Ví dụ:

```
console.time('100-elements');
for (var i = 0; i < 100; i++) {
  ;
}
console.timeEnd('100-elements');
```

console.trace(label)

In một tập stack các dấu vết stderr của vị trí hiện tại.

b. Process

Các process object gắn liền với goodies. Trước tiên, chúng ta sẽ có một cái nhìn tại một số thuộc tính cung cấp thông tin về node process đó:

process.version

Chuỗi phiên bản nút, ví dụ:

```
console.log (' Version:' + process.version);  // Version v0.8.16
```

process.execPath

Đường dẫn đến thư mục thực thi chính của chương trình *"/ usr / local / bin / node"*.

process.platform

Các nền tảng bạn đang sử dụng. Ví dụ, *"darwin"*.

process.pid

Các process ID.

process.stdout ()

Một luồng có thể ghi được đến stdout.

Ví dụ: Định nghĩa về `console.log()`.

```
console.log = function (d) {
process.stdout.write (d + '\ n');
};
```

process.stderr ()

Tương tự như `process.stdout()` nhưng ở đây là ghi đến stderr.

`process.stderr()` và `process.stdout()` là không giống như luồng khác trong Node, khi viết chúng thường bị blocking. Chúng bị blocking trong trường hợp mà chúng liên quan đến các tập tin thường xuyên hoặc mô tả tập tin TTY. Trong trường hợp chúng liên quan đến các pipes, chúng không bị blocking như những luồng khác.

process.stdin ()

Một luồng có thể đọc được cho stdin. Các dòng stdin bị tạm dừng theo mặc định, do đó, người ta phải gọi process.stdin.resume () để đọc từ nó.

Ví dụ mở đầu vào chuẩn và lắng nghe cả hai sự kiện:

```
process.stdin.resume ();
process.stdin.setEncoding ('utf8');
```

```
process.stdin.on ('data', function (chunk) {
process.stdout.write ('data: ' + chunk);
});
process.stdin.on (end', function () {
process.stdout.write (' end');
});
```

process.cwd ( )

Trả về thư mục làm việc hiện tại. Ví dụ:

```
cd ~ && node
node> process.cwd() "/Users/tj"
```

process.chdir ( )

Thay đổi thư mục làm việc hiện tại.

```
process.chdir('/ foo');
```

process.getuid ( )

Trả về số user ID của process đang chạy.

process.setuid ( )

Thiết lập user ID có hiệu lực cho quá trình đang chạy. Phương pháp này chấp nhận cả một số ID, cũng như một chuỗi. Ví dụ cả hai `process.setuid(501)`, và `process.setuid('tj')` đều hợp lệ.

process.getgid ( )

Trả về số group ID của process đang chạy.

process.setgid ( )

Tương tự như `process.setuid()` tuy nhiên được sử dụng trong group, cũng chấp nhận một số giá trị hoặc chuỗi đại diện. Ví dụ, `process.setgid(20)` hoặc `process.setgid('www')`.

process.chdir (directory)

Thay đổi thư mục làm việc hiện tại của process hoặc đưa một ngoại lệ nếu thất bại.

```
console.log ('Starting directory: '+ process.cwd ());
try {
process.chdir ('/ tmp');
console.log ('New directory:' + process.cwd ());
}
catch (err) {
```

```
console.log ('chdir: '+ err);
}
```

process.env

Một đối tượng có chứa các biến môi trường của người sử dụng. Ví dụ:

```
{ PATH:
'/Users/tj/.gem/ruby/1.8/bin:/Users/tj/.nvm/current/bin:/usr/bin:/bin:/u
s
/sbin:/sbin:/usr/local/bin:/usr/X1
, PWD: '/Users/tj/ebooks/masteringnode'
, EDITOR: 'mate'
, LANG: 'en_CA.UTF-8'
, SHLVL: '1'
, HOME: '/Users/tj'
, LOGNAME: 'tj'
, DISPLAY: '/tmp/launch-YCkT03/org.x:0'
, _: '/usr/local/bin/node'
, OLDPWD: '/Users/tj'
}
```

process.argv

Khi thực hiện một tập tin với các nút thực thi process.argv cung cấp truy cập vào các vector đối số, giá trị đầu tiên là nút thực thi, thứ hai là tên tập tin, và giá trị còn lại là các đối số được thông qua.

Ví dụ, tập tin nguồn của chúng ta ./src/process/misc.js có thể được thực hiện bằng cách chạy:

```
$ node src/process/misc.js foo bar baz
```

mà chúng ta gọi `console.dir(process.argv)`, xuất ra như sau:

```
[ 'node'
, '/Users/tj/EBooks/masteringnode/src/process/misc.js'
, 'foo'
, 'bar'
, 'baz'
]
```

process.exit ([code])

Lệnh `process.exit()` là đồng nghĩa với hàm C `exit()`, trong đó code > 0 được biết là thất bại, hoặc 0 được biết là thành công. Khi được gọi, việc exit được phát ra, cho phép một thời gian ngắn để chế biến tùy ý để xảy ra trước khi `process.reallyExit()` được gọi với mã trạng thái nhất định.

process.on ( )

Process tự nó đã là một *EventEmitter*, cho phép bạn làm những điều như lắng nghe cho các trường hợp ngoại lệ chưa bị bắt thông qua sự kiện *uncaughtException*:

```
process.on('uncaughtException', function(err){
console.log('got an error: %s', err.message);
process.exit(1);
});
setTimeout(function(){
throw new Error('fail');
}, 100);
```

process.kill (pid, [signal])

Lệnh `process.kill()` gửi tín hiệu thông qua pid nhất định, mặc định cho SIGINT. Trong ví dụ dưới đây, chúng ta gửi tín hiệu SIGTERM đến quá trình cùng một nút để minh họa bẫy tín hiệu, sau đó chúng ta ra " terminating " và thoát ra. Lưu ý rằng thời gian chờ thứ hai của 1000 mili giây là không bao giờ đạt được.

```
process.on ('SIGTERM', function () {
console.log('terminating');
process.exit(1);
});
setTimeout(function(){
      console.log('sending SIGTERM to process %d', process.pid);
   process.kill(process.pid, 'SIGTERM');
}, 500);
setTimeout(function(){
console.log('never called');
}, 1000);
});
```

Errno

Process object này lưu trữ của các con số báo hiệu lỗi tại máy chủ, tham khảo những gì bạn sẽ tìm thấy trong C-land. Ví dụ, `process.EPERM` đại diện cho một lỗi dựa trên sự cho phép, trong khi `process.ENOENT` đại diện cho một tập tin hoặc thư mục bị thiếu. Thông thường đây là những được sử dụng trong các ràng buộc để thu hẹp khoảng cách giữa C++ và JavaScript, nhưng chúng hữu ích cho việc xử lý các trường hợp ngoại lệ như:

```
if (err.errno === process.ENOENT) {
// Display a 404 "Not Found" page
} else {
// Display a 500 "Internal Server Error" page
}
```

c. Buffers

Cơ bản JavaScript là Unicode thân thiện, nhưng không phải với dữ liệu nhị phân. Khi giao tiếp với luồng TCP hoặc hệ thống tập tin, liệu nhị phân cần thiết để xử lý các luồng octet. Node cung cấp một số phương pháp cho việc khai thác, tạo và sử dụng luồng octet.

Để xử lý các dữ liệu nhị phân, node cung cấp cho chúng ta với các đối tượng toàn cục. Buffer là tương tự như một mảng các số nguyên, nhưng tương ứng với việc cấp phát bộ nhớ thô bên ngoài V8 heap. Buffer không thể được thay đổi kích cỡ. Có một số cách để xây dựng một trường hợp bộ đệm, và nhiều cách bạn có thể thao tác dữ liệu của nó.

Chuyển đổi giữa Buffers và các đối tượng chuỗi JavaScript đòi hỏi một phương pháp mã hóa rõ ràng. Dưới đây là chuỗi các bảng mã khác nhau.

• '`ascii`' - 7 bit dữ liệu ASCII duy nhất. Phương pháp mã hóa này là rất nhanh chóng, và sẽ loại bỏ các bit cao nếu thiết lập. Lưu ý rằng việc mã hóa này chuyển đổi một ký tự null ('\ 0' hoặc '\ u0000') vào 0x20 (mã ký tự của một không gian). Nếu bạn muốn chuyển đổi một ký tự null vào 0x00, bạn nên sử dụng '`utf8`'.

• '`utf8`' - Nhiều byte mã hóa ký tự Unicode. Nhiều trang web và các định dạng tài liệu khác sử dụng UTF-8.

• '`utf16le` - 2 hoặc 4 byte, ký tự Unicode mã hóa ít về cuối. Các cặp đại diện (U 10.000 FFFF 10 U) được hỗ trợ.

• '`UCS2`' - Tương tự '`utf16le`'.

• '`base64`' - Mã hóa chuỗi Base64.

• 'binary' - Một cách mã hóa dữ liệu nhị phân thành chuỗi bằng cách sử dụng 8 bit đầu tiên của mỗi ký tự. Phương pháp mã hóa này bị phản đối và nên tránh sử dụng các đối tượng bộ đệm nếu có thể. Mã hóa này sẽ được loại bỏ trong các phiên bản tương lai của Node.

• 'hex' - Mã hóa mỗi byte là hai ký tự thập lục phân.

Buffer cũng có thể được sử dụng xem mảng kiểu và DataViews.

```
var buff = new Buffer(4);

var ui16 = new Uint16Array(buff);

var view = new DataView(buff);

ui16[0] = 1;

ui16[1] = 2;

console.log(buff);

view.setInt16(0, 1);       // set big-endian int16 at byte offset 0

view.setInt16(2, 2, true); // set little-endian int16 at byte offset 2

console.log(buff);

// <Buffer 01 00 02 00>

// <Buffer 00 01 02 00>
```

Cách đơn giản nhất để xây dựng một bộ đệm từ một chuỗi chỉ đơn giản là sử dụng chuỗi như là tham số đầu tiên. Như bạn có thể nhìn thấy trong đăng nhập, bây giờ chúng ta có một đối tượng bộ đệm có chứa 5 byte dữ liệu được đại diện trong hệ thập lục phân.

```
var hello = new Buffer ('Hello'); console.log (hello);
// => <Buffer 48 65 6c 6c 6f>
console.log (hello.toString ());
// => "Hello"
```

Theo mặc định, mã hóa là "utf8", nhưng điều này có thể được thay đổi bằng cách đi qua một chuỗi như là đối số thứ hai. Ví dụ, dấu chấm lửng dưới đây sẽ được in stdout như kí tự "&" khi trong bảng mã " ascii ".

```
var buf = new Buffer('â–¦');

  console.log(buf.toString());

  // => â–¦

  var buf = new Buffer('â–¦', 'ascii');

  console.log(buf.toString());

  // => &
```

Một sự thay thế câu lệnh (nhưng trong trường hợp này hàm tương đương) để vượt qua một mảng các số nguyên đại diện cho dòng octet.

```
var hello = new Buffer ([0x48, 0x65, 0x6c, 0x6c, 0x6f);
```

Bộ đệm cũng có thể được tạo ra với một số nguyên đại diện cho số lượng các byte được phân bổ, sau đó chúng ta có thể gọi lệnh `write()`, cung cấp một offset và mã hóa. Dưới đây, chúng ta cung cấp một offset của 2 byte cuộc gọi thứ hai của chúng ta để `write()` (buffering "Hel") và sau đó viết 2 byte với offset của 3 (completing "Hello")

```
var buf = new Buffer(5);

buf.write('He');

buf.write('l', 2);

buf.write('lo', 3);

console.log(buf.toString());

// => "Hello"
```

The `.length` của một trường hợp bộ đệm chứa byte chiều dài của luồng (stream), trái ngược với chuỗi cục bộ, chỉ đơn giản là trả lại số ký tự. Ví dụ, kí tự ellipsis 'â-¦" bao gồm ba byte, do đó, các bộ đệm sẽ phản ứng với chiều dài byte (3), và không phải là chiều dài ký tự (1).

```
var ellipsis = new Buffer('â—¦', 'utf8');

console.log('â—¦ string length: %d', 'â—¦'.length);

// => â—¦ string length: 1


console.log('â—¦ byte length: %d', ellipsis.length);

// => â—¦ byte length: 3


console.log(ellipsis);

// => <Buffer e2 80 a6>
```

Để xác định độ dài byte của một chuỗi cục bộ, truyền đến lệnh `Buffer.byteLength()`.

API được viết bằng một cách như vậy mà nó giống như là String. Ví dụ, chúng ta có thể làm việc với các " slices" của bộ đệm bằng cách truyền offset cho câu lệnh `Slice()`:

```
var chunk = buf.slice(4, 9);

console.log(chunk.toString());

// => "some"
```

Ngoài ra, khi đợi một chuỗi, chúng ta có thể truyền offset đến `Buffer#toString()`:

```
var buf = new Buffer('just some data');
console.log(buf.toString('ascii', 4, 9));
// => "some"
```

2. Sự kiện (Event).

Khái niệm về một "sự kiện" là rất quan trọng trong node, và được sử dụng rất nhiều trong suốt module chính của chương trình và module của bên thứ 3. Module sự kiện chính của Node cung cấp cho chúng ta với một hàm tạo, *EventEmitter*.

a. EventEmitter

Thông thường một đối tượng kế thừa từ *EventEmitter*, tuy nhiên ví dụ nhỏ dưới đây minh họa API. Đầu tiên chúng ta tạo ra một `emitter`, sau đó chúng ta có thể xác định bất kỳ số lượng callbacks sử dụng `emitter.on()` phương pháp, mà chấp nhận tên của các sự kiện và các đối tượng tùy ý thông qua như là dữ liệu. Khi `emitter.emit()` được gọi, chúng ta chỉ required để truyền các tên sự kiện, theo sau bởi bất kỳ số lượng tham số (trong trường hợp này các chuỗi tên đầu tiên và cuối cùng).

```
var EventEmitter = require('events').EventEmitter;
var emitter = new EventEmitter;
emitter.on('name', function(first, last){
console.log(first   ', '   last);
});
emitter.emit('name', 'tj', 'holowaychuk');
emitter.emit('name', 'simon', 'holowaychuk');
```

b. Kế thừa từ EventEmitter (Inheriting From EventEmitter).

Được sử dụng phổ biến và thiết thực của *EventEmitter* là tính kế thừa từ nó. Điều này có nghĩa là chúng ta có thể giữ nguyên *EventEmitter* nguyên mẫu mà không bị ảnh hưởng trong khi sử dụng API của nó đối với phương tiện riêng của chúng ta.

Để làm như vậy, chúng ta bắt đầu bằng cách xác định các hàm khởi tạo Dog, trong đó tất nhiên sẽ bark từ thời gian đến thời gian (còn được biết đến như một sự kiện).

```
var EventEmitter = require('events').EventEmitter;
function Dog(name) {
this.name = name;
}
```

Ở đây chúng ta kế thừa từ *EventEmitter* vì vậy chúng ta có thể sử dụng các phương thức mà nó cung cấp, chẳng hạn như `EventEmitter#on()` và `EventEmitter#emit ()`. Nếu thuộc tính proto bị loại trừ, đừng lo lắng, chúng sẽ được trở lại này sau.

```
Dog.prototype. proto = EventEmitter.prototype;
```

Bây giờ chúng ta có `Dog` được thành lập, chúng ta có thể tạo ra ... Simon! Khi Simon bark, chúng ta có thể cho *stdout* biết bằng cách gọi `console.log()` với callback. Callback chính nó được gọi là trong ngữ cảnh của các đối tượng

```
var simon = new Dog('simon');

  simon.on('bark', function(){

  console.log(this.name   ' barked');

  });
```

Bark hai lần mỗi giây:

```
  setInterval(function(){

  simon.emit('bark');

  }, 500);
```

c. Loại bỏ các sự kiện lắng nghe (Removing Event Listeners).

Như chúng ta đã biết, lắng nghe sự kiện chỉ đơn giản là hàm đó được gọi khi chúng ta `emit()` một sự kiện. Chúng ta có thể loại bỏ những người nghe bằng cách dùng lệnh `removeListener(type, callback)`, mặc dù điều này không được dùng thường xuyên.

Trong ví dụ dưới đây, chúng ta phát ra thông báo "foo bar" mỗi 300 mili giây, trong đó có một cuộc gọi lại của `console.log()`. Sau 1000 mili giây, chúng ta gọi `removeListener()` với các đối số tương tự mà tôi đã thông qua `on()` ban đầu. Chúng ta cũng có thể sử dụng `removeAllListeners(type)`, trong đó loại bỏ tất cả các người nghe được đăng ký *type* nhất định.

```
  var EventEmitter = require('events').EventEmitter; var emitter = new

  EventEmitter; emitter.on('message', console.log);

  setInterval(function(){

  emitter.emit('message', 'foo bar');

  }, 300);

  setTimeout(function(){

  emitter.removeListener('message', console.log);

  }, 1000);
```

3. Luồng (Streams).

Streams là một khái niệm quan trọng trong nút. Các luồng API là một cách duy nhất để xử lý luồng giống như dữ liệu. Ví dụ, dữ liệu có thể được xem trực tiếp một tập tin, trực tiếp vào một socket để đáp ứng một HTTP request, hoặc trực tiếp từ một nguồn chỉ cho đọc như *stdin*. Để bây giờ, chúng ta sẽ tập trung vào các API, để lại các chi tiết cụ thể luồng chương sau.

a. Readable Streams

Readable Streams được xem như một HTTP request kế thừa từ `EventEmitter` để lộ dữ liệu đến qua các sự kiện. Việc đầu tiên của những sự kiện này là sự kiện dữ liệu, là một đoạn tùy ý của các dữ liệu được truyền đi để xử lý sự kiện như là một trường hợp đệm (`Buffer instance`).

```
req.on('data', function(buf){
// Làm gì đó với Buffer
});
```

Một sự kiện quan trọng khác là kết thúc, đại diện cho sự kết thúc của dữ liệu sự kiện. Ví dụ, đây là một HTTP echo server, chỉ đơn giản là "simply" các request body data thông qua các response. Vì vậy, nếu chúng ta POST "hello world", response của chúng ta sẽ là " hello world ".

```
var http = require('http');
http.createServer(function(req, res){
res.writeHead(200);
req.on('data', function(data){
res.write(data);
});
req.on('end', function(){
res.end();
});
}).listen(3000);
```

Module sys thực sự có một chức năng được thiết kế đặc biệt cho hành động "simply" này, aptly tên `sys.pump()`. Nó chấp nhận một luồng đọc như là đối số đầu tiên, và viết dòng thứ hai.

```
var http = require('http'),
    sys = require('sys');
http.createServer(function(req, res){
res.writeHead(200);
```

```
sys.pump(req, res);
}).listen(3000);
```

stream.readable

Giá trị boolean được mặc định là true, nhưng sẽ thành false sau khi xảy ra một lỗi, luồng đến một 'kết thúc', hoặc `destroy()` được gọi.

stream.setEncoding ([encoding])

Làm cho 'data' sự kiện phát ra một chuỗi thay vì một bộ đệm. encoding có thể là `'utf8'`, `'utf16le'('UCS2'),'ascii'`, hoặc `'hex'`. Mặc định là `'utf8'`. Như chúng ta biết, chúng ta có thể gọi `toString()` trên một bộ đệm để trả về một chuỗi đại diện của dữ liệu nhị phân.

Tương tự như vậy, chúng ta có thể gọi `setEncoding()` trên một luồng, sau đó dữ liệu sự kiện sẽ phát ra chuỗi.

```
req.setEncoding('utf8'); req.on('data', function(str){
// Làm gì đó với String
});
```

stream.pause ()

Vấn đề là một tín hiệu tư vấn cho các lớp giao tiếp cơ bản, yêu cầu không có thêm dữ liệu được gửi cho đến khi `resume()` được gọi.

Lưu ý rằng, do tính chất tư vấn, luồng nhất định sẽ không được tạm dừng ngay lập tức, và do đó, sự kiện 'data' có thể được phát ra cho một khoảng thời gian không xác định, ngay cả sau khi `pause()` được gọi.

stream.resume ()

Tiếp tục lại sự kiện 'data' sau khi `pause()`.

stream.destroy ()

Đóng tập tin mô tả cơ bản. Stream là không còn có thể ghi và cũng không thể đọc được.

Các dòng sẽ không phát ra bất kỳ chi tiết 'data', hoặc sự kiện 'kết thúc'. Bất kỳ dữ liệu ghi xếp hàng sẽ không được gửi đi. Các luồng được tải sự kiện 'close' khi nguồn lực của mình đã được xử lý.

stream.pipe (destination, [options])

Đây là một phương pháp Stream.prototype có sẵn trên tất cả các luồng.

Kết nối này đọc dòng để WriteStream điểm đến. Dữ liệu đến trên luồng này được ghi đến đích. Các luồng đích và nguồn được giữ đồng bộ bằng cách tạm dừng và khôi phục khi cần thiết. Chức năng này trả về các luồng đích.

Theo mặc định `end()` được gọi là điểm đến khi đi qua luồng nguồn phát ra cuối cùng, do đó, điểm đến mà không thể ghi được. {end: false} là tùy chọn để giữ cho luồng đích mở.

Điều này giữ `process.stdout` mở rằng "Goodbye" có thể được viết ở cuối.

```
process.stdin.resume();
process.stdin.pipe (process.stdout, {end: false});
process.stdin.on ("end", function () {
process.stdout.write ("Goodbye \ n");});
```

b. Writable Stream.

Một lớp cơ sở cho việc tạo ra các Writable Stream. Tương tự như Readable Stream, bạn có có thể tạo ra các lớp con bằng cách ghi đè không đồng bộ khi sử dụng câu lệnh `_write(chunk, cb)`.

stream.writable

Giá trị boolean được mặc định đó là true, nhưng sẽ thành false sau khi lỗi xảy ra hoặc `end() /destroy()` được gọi.

stream.Write (string, [encoding])

Viết chuỗi với encoding cho luồng. Trả về true nếu chuỗi đã được bỏ vào bộ đệm kernel. Trả về false để cho biết rằng bộ đệm kernel đã đầy, và dữ liệu sẽ được gửi đi trong tương lai. Sự kiện 'drain' sẽ cho biết khi nào bộ đệm kernel là rỗng một lần nữa. Việc mã hóa mặc định `'utf8'`.

stream.Write (buffer)

Tương tự như trên, ngoại trừ với một bộ đệm thô.

stream.end ()

Kết thúc dòng với EOF hoặc FIN. Cuộc gọi này sẽ cho phép hàng đợi ghi dữ liệu được gửi trước khi đóng luồng.

stream.end (string, encoding)

Gửi chuỗi với mã hóa nhất định và chấm dứt dòng với EOF hoặc FIN. Điều này rất hữu ích để giảm số lượng các gói tin gửi đi.

stream.end (buffer)

Tương tự như trên, nhưng với một bộ đệm.

stream.destroy ()

Đóng mô tả tập tin cơ bản. Stream là không còn có thể ghi và cũng không thể đọc được. các luồng sẽ không phát ra bất kỳ chi tiết 'data', hoặc sự kiện 'kết thúc'. Bất kỳ hàng đợi dữ liệu ghi sẽ không được gửi đi. các luồng được tải sự kiện 'close' khi tài nguyên của mình đã

được xử lý.

stream.destroySoon ()

Sau khi ghi hàng đợi được giải phóng, đóng tập tin mô tả. `destroySoon()` vẫn có thể hủy ngay lập tức, miễn là không có dữ liệu còn lại trong hàng đợi để viết.

4. File System

Để làm việc với hệ thống tập tin, node cung cấp module "fs". Các lệnh thực thi các hoạt động POSIX, và hầu hết các phương pháp làm việc đồng bộ hoặc không đồng bộ. Chúng ta sẽ xem xét làm thế nào để sử dụng cả hai, sau đó thiết lập lựa chọn tốt hơn.

a. Làm việc với tập tin hệ thống.

Cho phép bắt đầu với một ví dụ cơ bản làm việc với tập tin hệ thống. Ví dụ này tạo một thư mục, tạo ra một tập tin bên trong nó, sau đó viết nội dung của tập tin đến console:

```
var fs = require('fs');


fs.mkdir('./helloDir',0777, function (err) {
if (err) throw err;


fs.writeFile('./helloDir/message.txt', 'Hello Node', function (err)
{ if (err) throw err;
console.log('file created with contents:');


fs.readFile('./helloDir/message.txt','UTF-8' ,function (err, data)
 { if (err) throw err;
 console.log(data);
});
   });
});
```

Rõ ràng trong ví dụ trên, ứng với mỗi callback đều đặt trong callback trước đây được gọi là callbacks chainable. Mô hình này cần được theo sau khi sử dụng phương pháp không đồng bộ, không có đảm bảo rằng các hoạt động sẽ được hoàn thành theo thứ tự họ đang tạo ra. Điều này có thể dẫn đến hành vi không thể đoán trước.

Ví dụ có thể được viết lại để sử dụng một cách tiếp cận đồng bộ:

```
fs.mkdirSync('./helloDirSync',0777);
fs.writeFileSync('./helloDirSync/message.txt', 'Hello Node');
var data = fs.readFileSync('./helloDirSync/message.txt','UTF-8');
console.log('file created with contents:');
console.log(data);
```

Nó là tốt hơn để sử dụng phương pháp không đồng bộ trên các máy chủ với một tải cao, như các phương pháp đồng bộ sẽ làm cho toàn bộ quá trình để ngăn chặn và chờ cho các hoạt động để hoàn thành. Điều này sẽ ngăn chặn bất kỳ kết nối đến hoặc các sự kiện khác.

b. File thông tin

Các đối tượng fs.Stats có chứa thông tin về một tập tin hoặc thư mục cụ thể. Điều này có thể được sử dụng để xác định loại đối tượng mà chúng ta đang làm việc. Trong ví dụ này, chúng ta đang nhận được tất cả các đối tượng tập tin trong một thư mục và hiển thị cho dù chúng là một tập tin hoặc một đối tượng thư mục.

```
var fs = require('fs');
fs.readdir('/etc/', function (err, files)
{ if (err) throw err;
files.forEach( function (file) {
fs.stat('/etc/' + file, function (err, stats)
 { if (err) throw err;
 if (stats.isFile()) {
  console.log("%s is file", file);
}
 else if (stats.isDirectory ()) {
  console.log("%s is a directory", file);
}
console.log('stats:  %s',JSON.stringify(stats));
});
    });
 });
```

c. Xem các tập tin.

Phương pháp fs.watchfile theo dõi một tập tin và thay đổi một sự kiện bất cứ khi nào tập tin được thay đổi.

```
var fs = require('fs');
fs.watchFile('./testFile.txt', function (curr, prev) {
console.log('the current mtime is: ' + curr.mtime);
console.log('the previous mtime was: ' + prev.mtime);
});
fs.writeFile('./testFile.txt', "changed", function (err) {
if (err) throw err;
console.log("file write complete");
});
```

Một tập tin cũng có thể được unwatched bằng cách sử dụng phương pháp gọi fs.unwatchFile. Cách này chỉ nên sử dụng một lần khi tập tin không còn cần được giám sát.

5. HTTP.

Để sử dụng HTTP server và client phải dùng lệnh `require('http')`.

Các giao diện HTTP trong Node được thiết kế để hỗ trợ nhiều tính năng của các giao thức truyền thống khó sử dụng. Trong đó, có thể là đoạn mã hóa, tin nhắn.

HTTP headers được biểu diễn bởi một đối tượng như thế này:

```
{ 'content-length': '123',
  'content-type': 'text/plain',
  'connection': 'keep-alive',
  'accept': '*/*' }
```

Key được lowercased và giá trị không được sửa đổi.

http.STATUS_CODES

Một bộ sưu tập của tất cả các mã trạng thái tiêu chuẩn của HTTP response, và mô tả ngắn gọn cho từng cái. Ví dụ, `http.STATUS_CODES [404] === 'Not Found'`.

http.createServer ([requestListener])

Trả về một đối tượng web server mới. RequestListener là một chức năng được tự động thêm vào sự kiện `'request'`.

http.createClient ([port], [host])

Hàm này bị phản đối sử dụng, hãy sử dụng `http.request()` để thay thế. Xây dựng một HTTP client mới. Port và host tham chiếu đến máy chủ để được kết nối.

a. Class: http.Server.

server.listen(port, [hostname], [backlog], [callback])

Bắt đầu chấp nhận các kết nối trên port chỉ định và hostname. Nếu hostname được bỏ qua, các server sẽ chấp nhận các kết nối trực tiếp đến bất kỳ địa chỉ IPv4 (INADDR_ANY). Để nghe một socket unix, cung cấp một tên tập tin thay vì port và hostname.

Backlog là chiều dài tối đa của hàng đợi kết nối đang chờ. Chiều dài thực tế sẽ được xác định bởi hệ điều hành thông qua các thiết lập sysctl như `tcp_max_syn_backlog` và `somaxconn` trên Linux. Giá trị mặc định của tham số này là 511 (không phải 512).

Hàm này là không đồng bộ. Gọi lại tham số cuối cùng sẽ được thêm vào như là một listener cho sự kiện `'listen'`. Xem thêm `net.Server.listen(port)`.

server.listen (path, [callback])

Bắt đầu một máy chủ socket UNIX lắng nghe cho các kết nối trên đường dẫn nhất định. Hàm này là không đồng bộ. Gọi lại tham số cuối cùng sẽ được thêm vào như là một người biết lắng nghe cho sự kiện`'listen'`. Xem thêm `net.Server.listen (path)`.

server.listen (handle, [callback])

Các đối tượng có thể được thiết lập để xử lý một máy chủ hoặc socket, hoặc đối tượng `{fd: <n>}`.

Lắng nghe trên một mô tả tập tin không được hỗ trợ trên Windows.

Chức năng này là không đồng bộ. Gọi lại tham số cuối cùng sẽ được thêm vào như là một người biết lắng nghe cho sự kiện `'listen'`. Xem thêm `net.Server.listen()`.

server.close ([callback])

Dừng server và chấp nhận các kết nối mới. Xem `net.Server.close()`.

server.maxHeadersCount

Giới hạn tối đa headers count đến, là 1000 theo mặc định. Nếu thiết lập là 0 - không có giới hạn được áp dụng.

b. Class: http.ServerRequest.

Đối tượng này được tạo ra trong nội bộ của một máy chủ HTTP - không phải bởi người sử dụng - và thông qua như là đối số đầu tiên để một listener `'request'`. Request thực hiện các giao diện Readable Streams.

request.method

Yêu cầu một chuỗi. Chỉ đọc. Ví dụ: `'GET'`, `'DELETE'`.

request.url

Yêu cầu chuỗi URL. Câu lệnh này chỉ chứa các URL theo thực tế trong HTTP request. Nếu request là:

```
GET /status?name=ryan HTTP/1.1\r\n
Accept: text/plain\r\n
\r\n
Thì request.url sẽ là:
'/status?name=ryan'
```

Nếu bạn muốn để phân tích các URL thành các phần của nó, bạn có thể sử dụng `require('url').parse(request.url)`. Ví dụ:

```
node> require('url').parse('/status?name=ryan')
{ href: '/status?name=ryan',
  search: '?name=ryan',
  query: 'name=ryan',
  pathname: '/status' }
```

Nếu bạn muốn trích xuất các params từ chuỗi truy vấn, bạn có thể sử dụng `require('querystring').parse`, hoặc thông qua các đối số thứ hai để `require('url').parse`. Phân tích cú pháp. Ví dụ:

```
node> require('url').parse('/status?name=ryan', true)
{ href: '/status?name=ryan',
  search: '?name=ryan',
  query: { name: 'ryan' },
  pathname: '/status' }
```

request.headers

Đọc bản đồ duy nhất của tên header và giá trị. Tên header là lower-cased. Ví dụ:

```
// In một cái gì đó như:
```

```
// {User-agent ':' curl/7.22.0 '
// Host: '127 .0.0.1:8000 ',
// accept: '*/*' }
console.log(request.headers);
```

request.trailers

Chỉ đọc HTTP trailer (nếu có).

request.httpVersion

Phiên bản giao thức HTTP như một chuỗi. Chỉ đọc. Ví dụ: '1.1 ', '1.0'. Ngoài ra `request.httpVersionMajor` là số nguyên đầu tiên và `request.httpVersionMinor` là thứ hai.

request.setEncoding ([encoding])

Thiết lập mã hóa cho cơ chế request. Xem `stream.setEncoding()` để biết thêm thông tin.

request.pause ()

Tạm dừng request từ các sự kiện phát ra. Rất hữu ích để tăng tốc tải lên.

request.resume ()

Tiếp tục lại request tạm dừng.

request.connection

Các đối tượng net.Socket kết hợp với kết nối. Với hỗ trợ HTTPS, sử dụng `request.connection.verifyPeer()` và `request.connection.getPeerCertificate()` để có được thông tin xác thực của client.

c. Class: http.ServerResponse.

Đối tượng này được tạo ra trong nội bộ của một HTTP server - không phải bởi người sử dụng. Nó được thông qua như là tham số thứ hai của sự kiện `'request'`. Response thực hiện các giao diện Writable Stream.

response.writeHead (statusCode [reasonPhrase], [headers])

Gửi một response headers để đáp ứng yêu cầu. statusCode là một trạng thái HTTP có 3 chữ số mã, giống như 404. Tùy chọn có thể đưa ra một reasonPhrase như là đối số thứ hai.

Ví dụ:

```
var body = 'hello world';
response.writeHead(200, {
  'Content-Length': body.length,
```

```
  'Content-Type': 'text/plain' });
```

Câu lệnh này chỉ được gọi là một lần trên một tin nhắn và nó phải được gọi trước khi `response.end()` được gọi.

Nếu bạn gọi `response.write()` hoặc `response.end()` trước khi gọi, header ngầm định sẽ được tính toán và gọi hàm này cho bạn.

Lưu ý: rằng `Content-Length` được đưa ra trong các byte không phải ký tự. Ví dụ trên làm việc vì chuỗi 'hello world' chứa các byte ký tự duy nhất. Nếu nó chứa các ký tự được mã hóa cao hơn thì sau đó `Buffer.byteLength()` nên được sử dụng để xác định số byte trong một đoạn mã hóa nhất định. Và Node không kiểm tra `Content-Length` và độ dài của thân đã được truyền đi bằng nhau hay không.

response.statusCode

Khi sử dụng các header ẩn (không gọi `response.writeHead()` một cách rõ ràng), hàm điều khiển các mã trạng thái đó sẽ được gửi đến cho client khi các header nhận được.

Ví dụ:

```
response.statusCode = 404;
```

Sau khi response header đã được gửi cho client, hàm này chỉ ra các mã trạng thái được gửi.

response.setHeader (name, value)

Thiết lập một giá trị header duy nhất cho header ẩn. Nếu header này đã tồn tại trong phần header được gửi, giá trị của nó sẽ được thay thế. Sử dụng một mảng các chuỗi ở đây nếu bạn cần phải gửi nhiều header với cùng một tên.

Ví dụ:

```
response.setHeader ("Content-Type", "text / html");
```

hoặc

```
response.setHeader ("Set-Cookie", "loại = ninja", "ngôn ngữ = javascript
"]);
```

response.sendDate

Khi giá trị là true, date header sẽ được tự động tạo ra và gửi response nếu nó không phải là đã có trong các header. Mặc định là true.

response.getHeader (name)

Đọc ra một header đó đã được xếp hàng đợi nhưng không gửi cho client. Điều này chỉ có thể được gọi là trước khi header nhận được.

Ví dụ:

```
var contentType = response.getHeader ('content-type');
```

response.removeHeader (name)

Loại bỏ một header xếp hàng đợi để gửi ngầm.

Ví dụ:

```
response.removeHeader ("Content-Encoding");
```

response.write (chunk, [encoding])

Nếu hàm này được gọi và `response.writeHead()` đã được gọi, nó sẽ chuyển sang chế độ header ẩn và luôn xuất ra các header ẩn. Phương pháp này có thể được gọi là nhiều lần để cung cấp các bộ phận liên tiếp nhau.

chuck có thể là một chuỗi hoặc một bộ đệm. Nếu chuck là một chuỗi, tham số thứ hai xác định làm thế nào để mã hóa nó thành một dòng byte. Theo mặc định, mã hóa là 'utf8'.

Khi `response.write()` được gọi lần đầu tiên, nó sẽ gửi các thông tin header đệm và thân đầu tiên cho client. Khi `response.write()` được gọi lần thứ hai, Node giả định bạn sẽ được dữ liệu, và gửi riêng dữ liệu.

Trả về true nếu toàn bộ dữ liệu được xuất thành công đến kernel buffer. Trả về false nếu tất cả hoặc một phần của dữ liệu được xếp hàng đợi trong bộ nhớ của người sử dụng. 'drain' sẽ được phát ra khi bộ đệm trống một lần nữa.

response.end ([data], [encoding])

Với hàm này, tín hiệu đến server mà tất cả các response headers và phần thân đã được gửi, server nên xem xét hoàn thành tin nhắn này. `response.end()` phải được gọi là trên mỗi câu trả lời. Nếu dữ liệu được quy định cụ thể, đó là tương đương với gọi `response.write (data, encoding)` tiếp theo `response.end()`.

d. http.ClientResponse.

Đối tượng này được tạo ra khi thực hiện một yêu cầu với `http.request()`. Nó được thông qua với sự kiện `'response'` của đối tượng yêu cầu. `response` thực hiện các giao diện Readable Stream.

response.statusCode

3 chữ số mã trạng thái của HTTP response. VÍ DỤ: 404.

response.httpVersion

Các phiên bản HTTP của máy chủ để kết nối. Có thể hoặc là '1.1 'hoặc '1.0'. Ngoài ra `response.httpVersionMajor` là số nguyên đầu tiên và `response.httpVersionMinor` là

thứ hai.

response.headers

Đối tượng response headers .

response.trailers

Trailer đối tượng response.

response.setEncoding ([encoding])

Thiết lập mã hóa cho cơ thể phản ứng. Xem `stream.setEncoding ()` để biết thêm thông tin.

response.pause ()

Tạm dừng response từ các sự kiện phát ra. Hữu ích để tăng tốc tải về.

response.resume ()

Tiếp tục lại một response bị tạm dừng.

6. Đọc thêm Nodejs Docs.

Trên đây chỉ là các đối tượng cơ bản được nên ra, vẫn chưa được đầy đủ vì vậy muốn tìm hiểu sâu về Node.js thì nên xem các tài liệu của Node API được biên soạn rất chi tiết và liệt kê tất cả các lệnh hệ thống tập tin có thể có sẵn khi làm việc với Node.js.

IV. Các ứng dụng được xây dựng trên nền Node.js.

1. Ứng dụng đầu tiên.

Ứng dụng đầu tiên bạn nên tạo với nodejs là ứng dụng in chữ 'Hello World' trên cửa sổ console. Tạo một file với tên hello.js, với nội dung sau:

```
console.log('Hello World!');
```

Để chạy script, mở cửa sổ command line của node, đưa đường dẫn đến hello.js và chạy lệnh:

```
node hello.js
```

Bạn sẽ thấy chữ 'Hello World' hiện lên trên cửa sổ console.

2. HTTP Server.

Hãy đến với một ví dụ phức tạp hơn, có thể hơi rối khi bạn tiếp xúc lần đầu. hãy đọc các dòng code sau và các phần ghi chú:

```
    // Khai báo http module.
```

```
var http = require("http");

    // Tạo server. Hàm được thông qua như một tham số được gọi là trên
mọi request được tạo ra.
    // Biến request nắm giữ tất cả các tham số request
    // Biến response cho bạn làm bất cứ điều gì với response gửi tới
client.
http.createServer(function (request, response) {

    // Gán listener tại thời điểm kết thúc sự kiện.

    // Sự kiện này được gọi khi client gửi tất cả dữ liệu và đợi
response từ server.

   request.on("end", function () {

      // Viết headers cho response.

      // 200 là mã trạng thái của HTTP (điều này có nghĩa là thành công)

      // Tham số thứ hai giữ trường header trong đối tượng

      // Chúng ta đang gửi một văn bản đơn giản, do đó Content-Type nên
là text/plain

      response.writeHead(200, {

         'Content-Type': 'text/plain'

      });

      // Gửi dữ liệu và kết thúc response.

      response.end('Hello HTTP!');

   });
// Lắng nghe tại cổng 8080.
}).listen(8080);
```

Thật là đơn giản, bạn có thể gửi dữ liệu đến các client bằng cách sử dụng `response.write()`, nhưng bạn phải gọi nó trước khi gọi hàm `response.end()`. Lưu mã trên thành file `http.js` và đánh lệnh sau trên console:

```
node http.js
```

Sau đó mở cửa sổ trình duyệt browser đến địa chỉ `http://localhost:8080`. Bạn sẽ thấy dòng chữ “Hello HTTP!”.

Đoạn mã sau là một máy chủ TCP lắng nghe trên cổng 8080 và echo chuỗi ‘hello’ ra mỗi kết nối:

```
var net = require('net');

net.createServer(function (stream) {
```

```
        stream.write('hello\r\n');
        stream.on('end', function () {
            stream.end('goodbye\r\n');
        });
        stream.pipe(stream);
    }).listen(8080);
```

3. Xử lý các tham số URL

Như đã nói trước, ta phải tự làm mọi thứ trong node. Hãy xem code của ví dụ sau:

```
    // Khai báo http module.
    var http = require("http"),
    // và url module, nó rất hữu ích trong việc phân tích tham số request.
       url = require("url");
    // Tạo server.
    http.createServer(function (request, response) {
       // Gán listener và kết thúc sự kiện.
       request.on('end', function () {
          // Phân tích request cho các đối số và lưu chúng trong biến _get.
         // Hàm này phân tích các url từ reqest và trả về các đối tượng đại
diện.
          var _get = url.parse(request.url, true).query;
          // Viết headers cho response.
          response.writeHead(200, {
             'Content-Type': 'text/plain'
          });
          // Gửi dữ liệu và kiết thúc response.
          response.end('Here is your data: ' + _get['data']);
       });
    // Lắng nghe tại cổng 8080.
    }).listen(8080);
```

Phần code này dùng phương thức `parse()` của module url, một module lõi của Nodejs, để convert từ request url thành object. Phương thức trả lại object là phương thức query, sẽ lấy lại thông số của URL. Lưu file này thành `get.js` và thực thi trên cửa sổ lệnh:

```
node get.js
```

Sau đó vào đường dẫn: `http://localhost:8080/?data=put_some_text_here` để xem kết quả.

4. Đọc và viết file

Để đọc và ghi file trong Node ta dùng fs module, một module sẵn có trong Node. Các hàm đọc và ghi file là `fs.readFile()` và `fs.writeFile()`. Ta có ví dụ sau:

```
// Khai báo http module,
var http = require("http"),
// và mysql module.
   fs = require("fs");
// Tạo http server.
http.createServer(function (request, response) {
   // Gán listener và kết thúc sự kiện.
   request.on('end', function () {
      // Kiểm tra nếu là user requests /
      if (request.url == '/') {
         // đọc file.
         fs.readFile('test.txt', 'utf-8', function (error, data) {
            // Viết headers.
            response.writeHead(200, {
               'Content-Type': 'text/plain'
            });
            // Tăng số thu được từ file.
            data = parseInt(data) + 1;
            // Viết số đã tăng vào file.
            fs.writeFile('test.txt', data);
            // Kết thúc response với tin nhắn.
            response.end('This page was refreshed ' + data + ' times!');
```

```
            });
        } else {
            //Cho biết nếu tập tin request không được tìm thấy.
            response.writeHead(404);
            // và kết thúc request mà không gửi dữ liệu nào.
            response.end();
        }
    });
// Lắng nghe tại cổng 8080.
}).listen(8080);
```

Lưu file lại thành file `files.js`. Trước khi chạy file này trên cửa sổ command, tạo một file `test.txt` cùng thư mục với file `file.js`.

Hãy chạy và kiểm tra kết quả.

5. Nodejs với mysql

Hầu hết các môi trường truyền thống chạy server side đều có những chức năng kèm theo để thao tác với database. Với Node.js, bạn phải cài thêm thư viện. Với bài viết này, tôi chọn một thư viện khá ổn định để dùng. Tên đầy đủ của module thư viện là: `mysql@2.0.0-alpha2` (phía sau @ chỉ là tên phiên bản). Mở cửa sổ command, dùng npm cài module này lên với lệnh:

```
npm install mysql@2.0.0-alpha2
```

Lệnh này sẽ download và cài đặt module, và nó cũng tạo một folder trong thư mục hiện hành. Bạn hãy quan sát phần code sau để biết thao tác với csdl:

```
// Khai báo http module,
var http = require('http'),
// và mysql module.
    mysql = require("mysql");
// Tạo kết nối.
// Dữ liệu là mặc định để cài đặt mới mysql và nên được thay đổi theo cấu hình của bạn.
var connection = mysql.createConnection({
    user: "root",
    password: "",
```

```
    database: "db_name"
  });
  // Tạo http server.
  http.createServer(function (request, response) {
    // Gán listener và kết thúc sự kiện.
    request.on('end', function () {
      // Truy vấn cơ sở dữ liệu.
        connection.query('SELECT * FROM your_table;', function (error,
rows, fields) {
          response.writeHead(200, {
            'Content-Type': 'x-application/json'
          });
          // Gửi dữ liệu là chuỗi JSON.
          // Biến rows giữ kết quả của các truy vấn.
          response.end(JSON.stringify(rows));
      });
    });
  // Lắng nghe cổng 8080.
  }).listen(8080);
```

Truy xuất dữ liệu với thư viện này rất đơn giản, bạn chỉ cần nhập câu lệnh truy xuất và gọi hàm. Trong các ứng dụng thực tế, bạn nên kiểm tra nếu xãy ra lỗi để dễ dàng debug, và trả lại các kết quả mã lỗi khi thực hiện cầu lệnh thành công hay không. Lưu ý là trong ví dụ này ta cũng set Content-type với giá trị `x-application/json`, đó là một giá trị MIME của JSON. Tham số rows sẽ lưu giữ kết quả của truy vấn, và ta đã chuyển đổi dữ liệu trong rows sang cấu trúc của JSON qua phương thức `JSON.stringify()`.

Lưu file lại với tên `mysql.js` sau đó thực thi trên cửa sổ command, với csdl mysql đã được cài đặt

```
Node mysql.js
```

Sau đó vào đường dẫn: `http://localhost:8080` trên trình duyệt và bạn sẽ được nhắc download file JSON-formatted

V. WebSocket với Node.js và Socket.IO

Socket.IO là một thư viện javascript có mục đích tạo ra các ứng dụng realtime trên trình duyệt cũng như thiết bị di động. Việc sử dụng thư viện này cũng rất đơn giản và giống nhau ở cả server lẫn client.

Sau đó cài mở cửa sổ console và cài đặt Socket.io bằng lệnh sau:

```
npm install socket.io
```

1. Tìm hiểu về Socket.IO

Với Node.js, bạn chỉ cần biết vài hàm cơ bản như requires() để import thư viện. Công việc còn lại, chỉ cần dùng Socket.IO nên bạn cần phải tìm hiểu về thư viện này tại đây: `http://socket.io/#how-to-use`.

- Server: tạo một đối tượng socket bằng phương thức `listen(port)`. Phương thức này chờ đợi một yêu cầu kết nối từ client.

- Client: Kết nối đến server bằng phương thức `connect(url,{port: server_port})`.

- Socket.IO cung cấp 3 event chính là connect, message và disconnect. Chúng được kích hoạt khi client/server:
  + connect: tạo kết nối
  + message: nhận được thông điệp
  + disconnect: ngắt kết nối

Ví dụ:

```
socket.on("message", function(msg){
      // console.log("Received: "+ msg);
   });
```

- Để gửi dữ liệu, ta dùng lệnh `send()`. Dữ liệu có thể là đối tượng (được chuyển thành chuỗi json) và sẽ nhận được qua sự kiện message.

Ví dụ: `socket.send("Hello world");`

- Socket.IO có thể gửi và nhận các event tự tạo với phương thức `emit()`. Hai phía gửi và nhận phải biết được tên của event đó để thực hiện giao tiếp:

Ví dụ:

```
// gửi:
socket.emit("hello",{msg: "welcome");
// nhận:
socket.on("hello", function (data) {
            console.log(data);
```

```
            }
        );
```

2. Ứng dụng tính kết quả biểu thức cho trước.

Sử dụng chương trình notepad hoặc công cụ soạn thảo lập trình nào đó để tạo 2 file sau.

Server.js:

Tạo một tập tin `server.js` với nội dung sau:

```
var io = require('socket.io');
var socket = io.listen(8080);
socket.sockets.on('connection',function(socket){
        socket.on('message', function(expr){
            console.log('Received expression from client ',expr);
                    // Bắt lỗi đối với biểu thức xấu
                    try{
                            socket.send(eval(expr));
                    }catch(err){
                            socket.emit("error",err.message);
                    }
        });
        socket.on('disconnect', function(){
            // console.log('Disconnected');
        });
     });
```

Client.html:

```
<html>
    <head>
    <title>WebSocket Client</title>
    <script src="http://localhost:8080/socket.io/socket.io.js"></script>
    <script>
    window.onload = function(){
          var input = document.getElementById('input');
          var output = document.getElementById('output');
```

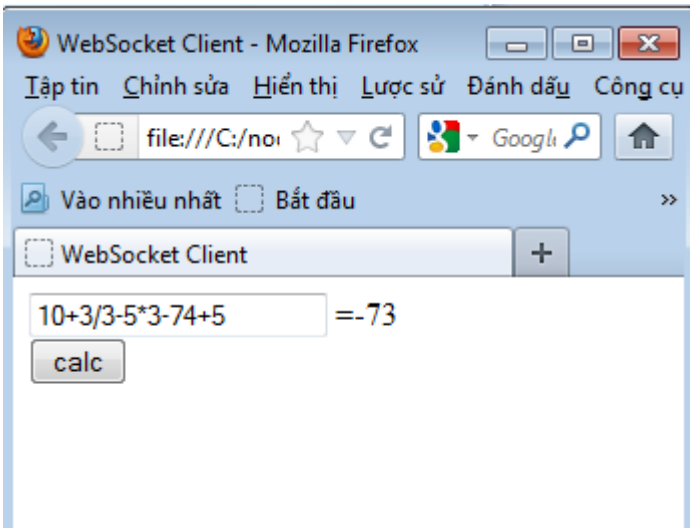
```
            var socket = io.connect('localhost',{port: 8080});
            socket.on("connect",function(){
                  console.log("Connected to the server");
            });
            socket.on('message',function(data) {
                  output.innerHTML = '=' + data;
            });
            socket.on('error', function (data) {
                  console.log("error:",data);
               }
            );
            window.sendMessage = function(){
                  socket.send(input.value);
            };
      }
      </script>
      </head>
      <body>
            <input type='text' id='input' />
            <span id='output'></span>
            <br/>
            <input type='button' id='send' value='calc'
onclick='sendMessage();' />
      </body>
  </html>
```

Lưu và thoát khỏi các tập tin. Di chuyển vào bên trong thư mục bieuthuc và bắt đầu chạy ứng dụng.

```
node Server.js
```

Bây giờ mở file Client.html nhập biểu thức và nhận kết quả. Ứng dụng chỉ làm việc được với biểu thức tính toán không có dấu ngoặc đơn.

Kết quả:



3. Ứng dụng webchat.

a. Xây dựng một webchat server cơ bản.

Trước tiên, ta sẽ nhập các mô-đun Socket.IO.

```
var exp = require('express');
var app = exp.createServer();
```

Sau đó, ta sẽ tạo ra một biến toàn cục duy nhất của Socket.IO được sử dụng để chia sẻ trên nhiều ứng dụng.

```
global.socket = require('socket.io').listen(app);
global.socket.set('log level', 1);
global.socket.set('transports', [ 'websocket', 'flashsocket', 'htmlfile', 'xhr-polling', 'jsonp-polling']);
```

Ta sẽ tải tập tin cấu hình, router, và một mô-đun chat-socket đã có sẵn trong cùng một thời điểm.

```
require('./app/config')(app, exp);
require('./app/server/router')(app);
require('./app/server/modules/chat-socket');
```

Và sau đó khởi động server.

```
app.listen(8080, function(){
    console.log("Express server listening on port %d in %s mode", app.address().port, app.settings.env);
```

```
});
```

b. Tạo module Server-Side chat socket.

Điều đầu tiên ta cần làm là xác định không gian tên cho ứng dụng này.Ta làm điều này với câu lệnh `socket.of ('namespace').`

```
module.exports = function()
{
    global.socket.of('/chat').on('connection', function(socket) {
     console.log("a user has connected to the 'chat' namespace");
    });
}();
```

Bây giờ ta có thể thêm một số code để theo dõi người sử dụng kết nối, phát tin nhắn của họ và lắng nghe khi họ ngắt kết nối.

Nhưng trước tiên hãy tạo một mảng các màu mà ta ngẫu nhiên sẽ gán cho người dùng khi họ kết nối để phân biệt nhau trong bảng chat.

```
var colors = ['#AE331F', '#D68434', '#116A9F', '#360B95', '#5F209E'];
```

Tiếp theo ta sẽ tạo ra một " connections object " để theo dõi người dùng kết nối.

```
var connections = { };
```

Kịch bản phía client sẽ phát ra một sự kiện "user-ready" sau khi một người kết nối thành công và được giao một tên. Server sẽ lưu trữ giá trị đó trong socket server để sử dụng trong tương lai. Server cũng sẽ giao cho người sử dụng mới được kết nối một màu ngẫu nhiên từ mảng màu sắc có sẵn.

```
global.socket.of('/chat').on('connection', function(socket) {
    socket.on('user-ready', function(data) {
    socket.name = data.name;
    socket.color = data.color = colors[Math.floor(Math.random() *
colors.length)];
// let everyone else in the room know a new user has just connected //
    brodcastMessage('user-ready', data);
});
```

Ta sẽ lắng nghe cho đến khi họ ngắt kết nối, do đó ta có thể loại bỏ chúng từ các đối tượng kết nối và thông báo cho người sử dụng.

```
socket.on('disconnect', function() {
```

```
        delete connections[socket.id];
        dispatchStatus();
        brodcastMessage('user-disconnected', { name : socket.name, color :
socket.color });
    });
```

`dispatchStatus` và `broadcastMessage` chỉ là các shortcut để phát sự kiện trở lại với phần còn lại của nhóm.

```
    function dispatchStatus()
    {
        brodcastMessage('status', connections);
    }
    function brodcastMessage(message, data)
    {
        socket.emit(message, data);
        socket.broadcast.emit(message, data);
    }
```

Khi một người dùng gửi tin nhắn cho thêm màu sắc kết hợp với Socket của mình vào tin nhắn để chúng tôi có thể hiển thị màu sắc thích hợp trong nhật ký trò chuyện khi phát trở lại với phần còn lại của nhóm.

```
    socket.on('user-message', function(data) {
        data.color = socket.color;
        brodcastMessage('user-message', data);
    });
```

c. Khởi tạo kết nối trên Client

Khi một người dùng kết nối với ứng dụng webchat, ta sẽ cung cấp cho họ một tên mặc định. Khi gửi tin nhắn, ta sẽ lấy giá trị này và thêm nó vào tin nhắn được gửi đi.

```
    $('#name').val(Math.random().toFixed(8).toString().substr(2));
    socket = io.connect('/chat');
```

Đó là những không gian tên riêng biệt cho phép các ứng dụng chatcó thể duy trì quyền tự chủ trong khi chạy như tên miền phụ của ứng dụng Node.

```
$('#btn-send').click(function(){ sendMessage(); })
    var sendMessage = function() {
```

```
        socket.emit('user-message', {name : $('#name').val() , message :
$('#msg').val() });

        $('#msg').val('');

    }
```

`SendMessage` chỉ đơn giản là lấy các giá trị được lưu trữ trong trường nhập tên và trường văn bản `# msg #`, khiến chúng thành một đối tượng mà chúng ta có thể phát ra trở lại máy chủ. Sau khi tin nhắn được gửi chúng tôi sẽ tự động xóa vùng văn bản `# msg #` vì vậy người dùng có thể gửi một tin nhắn mới.

Phần còn lại của các mã trên client chỉ đơn giản là lắng nghe cho các sự kiện đến và cập nhật các cuộc hội thoại phù hợp.

```
    socket.on('user-ready', function (data) {

        $('#incoming').append('<span class="shadow"
style="color:'+data.color+'">'+data.name +' :: connected</span><br>');

    });

    socket.on('user-message', function (data) {

        $('#incoming').append('<span class="shadow"
style="color:'+data.color+'">'+data.name +' :: '+
data.message+'</span><br>');

    });

    socket.on('user-disconnected', function (data) {

        $('#incoming').append('<span class="shadow"
style="color:'+data.color+'">'+data.name +' :: disconnected</span><br>');
```

Lưu ý, trong trang HTML hãy chắc chắn rằng có bao gồm thư viện Socket.IO được để trong thư mục node_modules tại thư mục gốc của server.

```
    <script src="/socket.io/socket.io.js"></script>
```

Tại thời điểm này, chúng ta đã có một server webchat đơn giản có thể lắng nghe và gửi tin nhắn cho tất cả người dùng kết nối không gian tên của ứng dụng "/ chat".

Để có được các module và chạy, cd vào từng ứng dụng và cài đặt phụ thuộc của nó:

```
    cd mydomain.com

    npm install -d
```

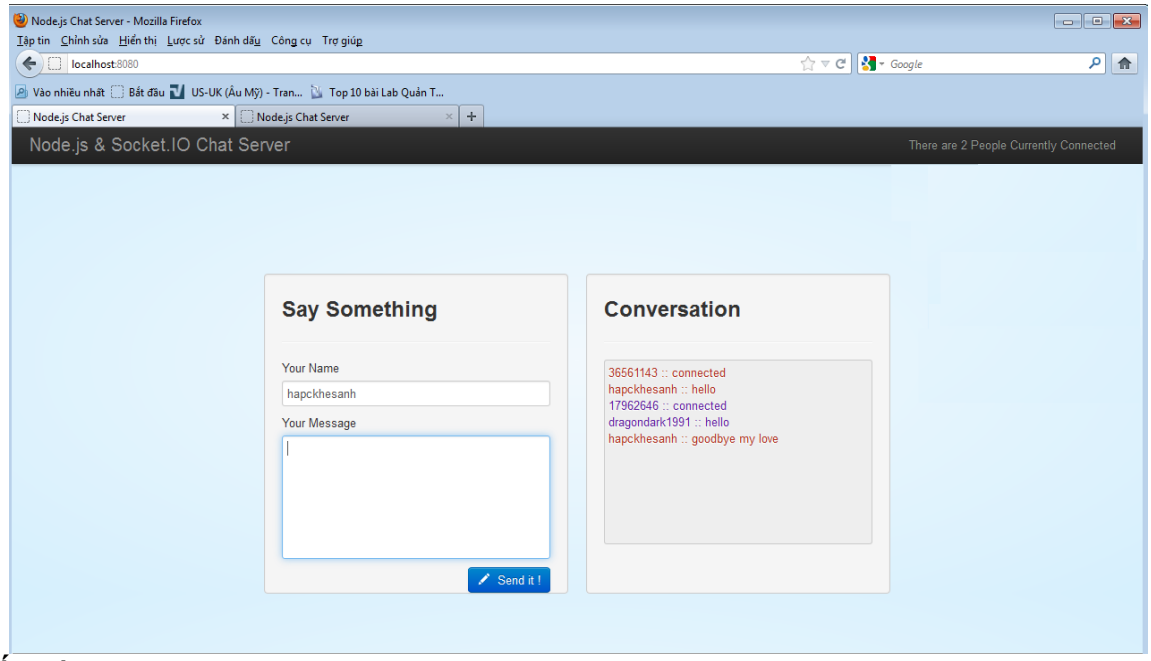
Lưu và thoát khỏi các tập tin. Di chuyển vào bên trong thư mục mydomain.com và bắt đầu chạy ứng dụng.

```
    node app.js
```

Bây giờ mở trình duyệt và vào đường dẫn :

```
http://localhost:8080
```

Kết quả:



Kết luận

Node.js đòi hỏi phải làm thêm nhiều việc, nhưng phần thưởng của một ứng dụng nhanh chóng và mạnh mẽ là giá trị nó. Node.js là một công nghệ đầy hứa hẹn và là sự lựa chọn tuyệt với cho một ứng dụng cần có hiệu năng cao, nhiều kết nối. Nó đã được chứng minh bởi các công ty như Microsoft, eBay và Yahoo.