



Bộ môn Công nghệ Phần mềm
Viện CNTT & TT

Trường Đại học Bách Khoa Hà Nội

CÔNG NGHỆ WEB TIỀN TIẾN

Bài 09: NODEJS và ExpressJS

Mục lục

1. Tổng quan lập trình Server
2. Tổng quan NodeJS
3. ExpressJS
4. Xử lý tham số/trả về
5. Async/await

Mục lục

1. Tổng quan lập trình Server
2. **Tổng quan NodeJS**
3. ExpressJS
4. Xử lý tham số/trả về
5. Async/await

2. Tổng quan NodeJS

2.1. Mô hình hoạt động

2.2. Cài đặt npm

2.3. Câu lệnh node

2.4. Localtunnel

2.5. Ví dụ

2. Tổng quan NodeJS

2.1. Mô hình hoạt động

2.2. Cài đặt npm

2.3. Câu lệnh node

2.4. Localtunnel

2.5. Ví dụ

2.1. Mô hình hoạt động

NodeJS:

- Một bộ JavaScript runtime được viết bằng C ++.
- Có thể thông dịch và thực thi JavaScript.
- Chứa sẵn các API được xây dựng từ trước.

API NodeJS:

- Một tập hợp các thư viện JavaScript hữu ích để tạo các chương trình trên máy chủ.

V8 (trên **Chrome):**

- Trình thông dịch JavaScript ("engine") mà NodeJS sử dụng để thông dịch và thực thi mã JavaScript

2. Tổng quan NodeJS

2.1. Mô hình hoạt động

2.2. Cài đặt npm

2.3. Câu lệnh node

2.4. Localtunnel

2.5. Ví dụ

2.2. Cài đặt npm

Khi cài đặt NodeJS, ta cũng cần cài cả npm:

- **npm**: Node Package Manager:

Công cụ dòng lệnh cho phép cài đặt các **packages** (thư viện và công cụ) viết bằng JavaScript và tương thích với NodeJS

- Có thể tìm đủ mọi thứ ở đây:

<https://www.npmjs.com/>

`npm install package-name`

- Dùng để thêm gói mới

`npm uninstall package-name`

- Dùng để gỡ đi gói cũ



Các gói cần cài thêm

Nhìn chung, hầu như dự án JS nào cũng cần phải cài ExpressJS và body-parser.

```
$ npm install express
```

```
$ npm install body-parser
```

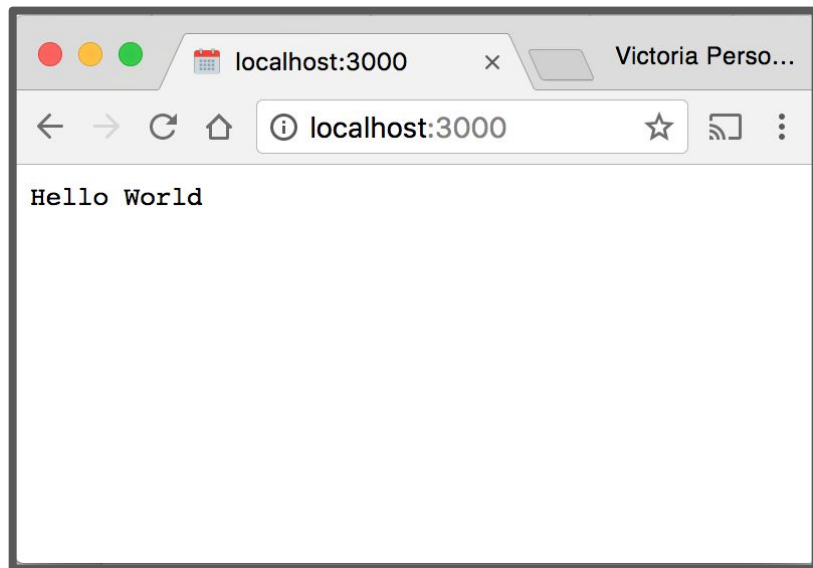
Chúng sẽ được nằm trong thư mục `node_modules`.

Ví dụ dùng npm để cài Express

```
$ npm install express
```

```
$ node server.js
```

Example app listening on port 3000!



Ví dụ dùng npm để cài body-parser

Dùng thư viện sau [body-parser library](#) để gửi dữ liệu dạng JSON:

```
const bodyParser = require('body-parser');
```

Đây không phải là thư viện có sẵn trong NodeJS API, vậy nên ta cần cài nó:

```
$ npm install body-parser
```

Ví dụ dùng npm để cài body-parser

Dùng thư viện sau [body-parser library](#) để gửi dữ liệu dạng JSON:

```
const bodyParser = require('body-parser');  
const jsonParser = bodyParser.json();
```

Giúp truyền lên server các tham số ở dưới định dạng JSON và phía server có thể chuyển đổi lại dưới dạng các đối tượng.

Tải mã nguồn lên GitHub?

Khi bạn cập nhật mã nguồn NodeJS lên một GitHub repository, **bạn không được tải lên thư mục `node_modules`:**

- Chúng ta không có nhu cầu chỉnh sửa các mã nguồn trong thư mục đó
- Tải lên chỉ làm tăng kích thước của repository đó thôi

CÂU HỎI: NHƯNG LÀM THẾ NÀO ĐỂ NGƯỜI KHÁC BIẾT DỰ ÁN CỦA BẠN CẦN CÁC THƯ VIỆN NÀO KHÁC?

Nếu ta không tải lên thư mục đó, ta cần một cách khác để cho người khác biết rằng họ cần cài thêm các thành phần nào nữa.

npm cung cấp cơ chế đó thông qua: [package.json](#)

package.json

Tạo ra file [package.json](#) trong thư mục gốc của dự án NodeJS để đặc tả các metadata cho dự án.

Tạo file [package.json](#) đó bằng lệnh:

```
$ npm init
```

Sẽ có hàng loạt câu hỏi xuất hiện cho bạn về **package.json** dựa trên câu trả lời của bạn.

Một file package.json tự tạo

```
{  
  "name": "fetch-to-server",  
  "version": "1.0.0",  
  "description": "Example of fetching to a server",  
  "main": "server.js",  
  "dependencies": {  
    "body-parser": "^1.17.1",  
    "express": "^4.15.2"  
  },  
  "devDependencies": {},  
  "scripts": {  
    "test": "echo \"Error: no test specified\" && exit 1",  
    "start": "node server.js"  
  },  
  "author": "Victoria Kirst",  
  "license": "ISC"  
}
```

Lưu trữ sự phụ thuộc vào package.json

Khi cài thêm thư viện mới, có thể truyền thêm tham số để lưu sự phụ thuộc này --save:

```
$ npm install --save express
```

```
$ npm install --save body-parser
```

Sẽ tạo ra một đoạn văn bản sau trong package.json.

```
"dependencies": {  
  "body-parser": "^1.17.1",  
  "express": "^4.15.2"  
},
```


Lưu trữ sự phụ thuộc vào package.json

Nếu bạn lỡ tay xóa mất thư mục này:

```
$ rm -rf node_modules
```

Ta có thể cài đặt lại các gói trong danh sách các phụ thuộc như sau:

```
$ npm install
```

- Như vậy người khác có thể dễ dàng tải mã nguồn của bạn về, nạp lại các thư viện cần thiết chỉ bằng lệnh đơn giản

npm scripts

Trong package.json cũng có thể định nghĩa các scripts:

```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "start": "node server.js"  
},
```

Ta có thể thực thi các mã đó bằng cách chạy lệnh `$ npm scriptName`

Ví dụ để chạy lệnh "node server.js"

```
$ npm start
```

2. Tổng quan NodeJS

2.1. Mô hình hoạt động

2.2. Cài đặt npm

2.3. Câu lệnh node

2.4. Localtunnel

2.5. Ví dụ

2.3. Câu lệnh node

Có thể dùng câu lệnh **node** để thực thi các mã **JS** không nằm trong file

- Giống như thực thi trong **JavaScript** console của **Chrome**

```
$ node
```

```
> let x = 5;
```

```
undefined
```

```
> x++
```

```
5
```

```
> x
```

```
6
```

2.3. Câu lệnh node (2)

Ngoài ra có thể thực thi các mã nguồn nằm trong file cụ thể

simple-script.js

```
function printPoem() {  
    console.log('Roses are red,');  
    console.log('Violets are blue,');  
    console.log('Sugar is sweet,');  
    console.log('And so are you.');
```



```
    console.log();  
}  
  
printPoem();  
printPoem();
```

2.3. Câu lệnh node (3)

Bằng cách đặt tên của file JS đằng sau câu lệnh node:

```
$ node fileName
```

```
$ node simple-script.js
```

```
Roses are red,  
Violets are blue,  
Sugar is sweet,  
And so are you.
```

```
Roses are red,  
Violets are blue,  
Sugar is sweet,  
And so are you.
```

2. Tổng quan NodeJS

2.1. Mô hình hoạt động

2.2. Cài đặt npm

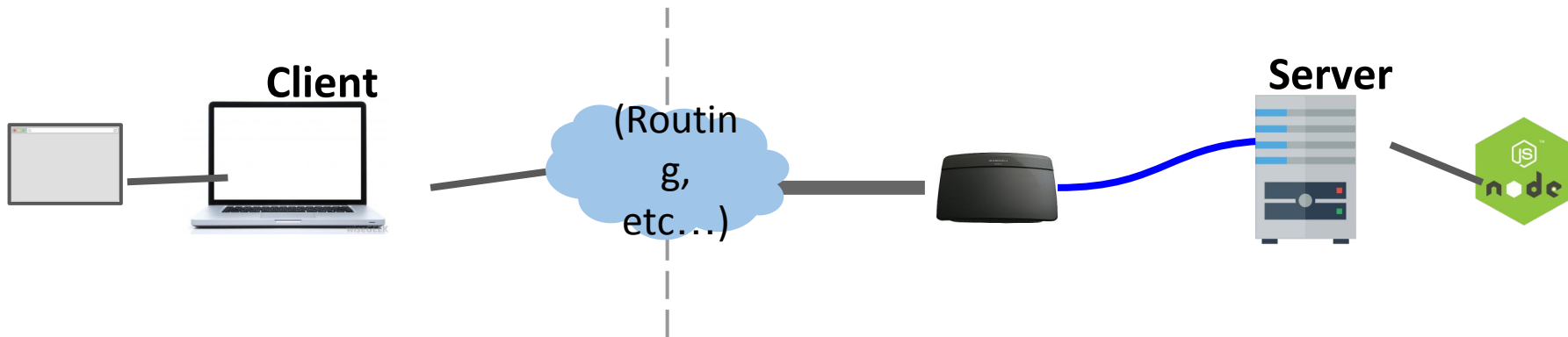
2.3. Câu lệnh node

2.4. Localtunnel

2.5. Ví dụ

2.4. Localtunnel

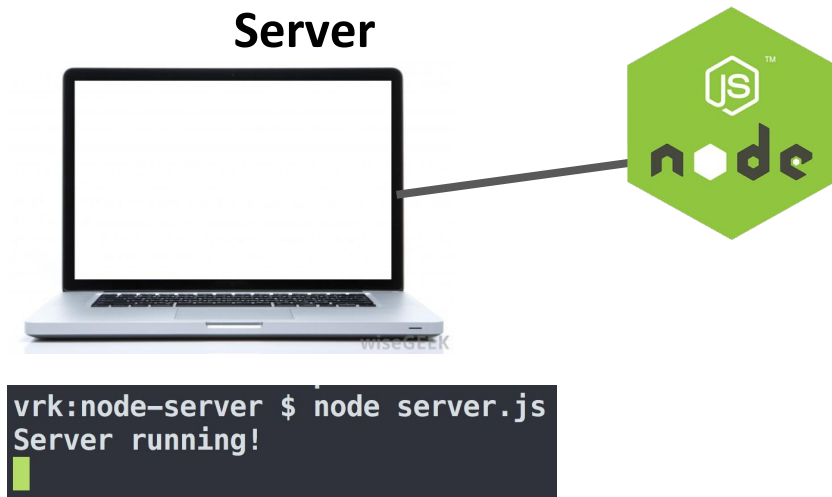
Nhìn chung, các nhà phát triển nhỏ lẻ thường xây dựng hệ thống của mình trên local trước:



Việc phát triển trực tiếp trên server thật đòi hỏi sự cẩn thận rất cao. **Phát triển trên local khiến ta có thể dùng chính laptop với vai trò client và server**

2.4. Localtunnel (2)

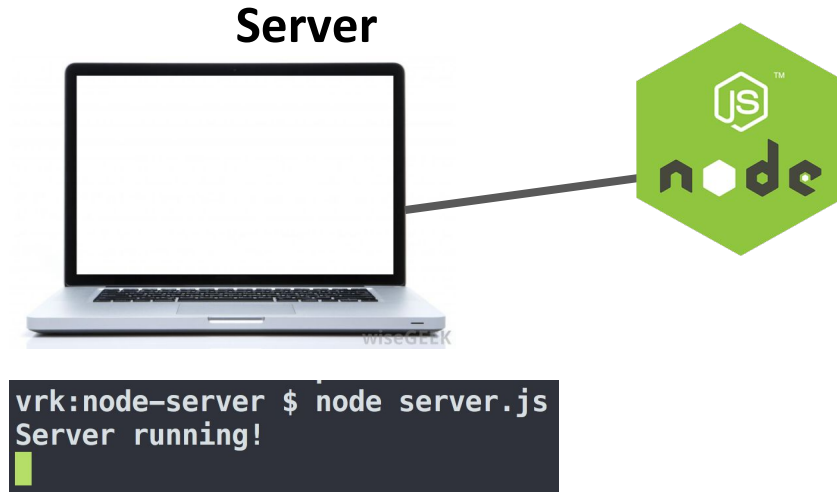
Khi ta thực thi câu lệnh `$ node server.js`, nó sẽ chạy `server.listen(3000)`, **máy laptop thành một server**:



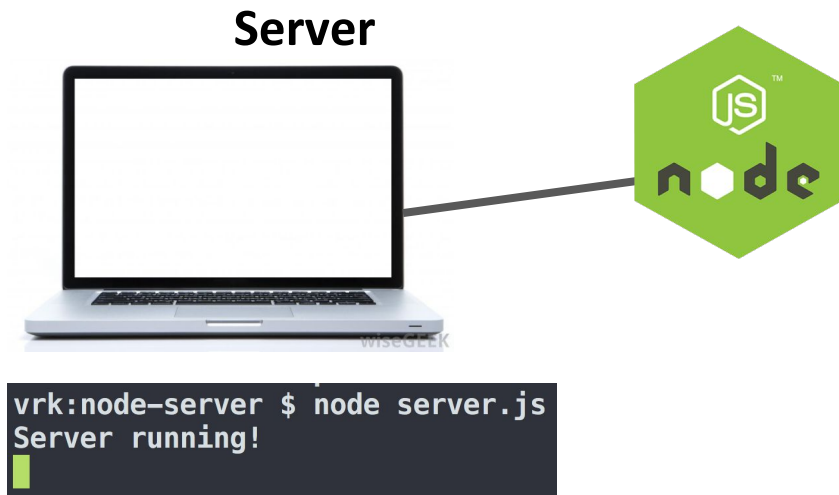
Khi hệ điều hành trên laptop nhận được thông báo HTTP qua cổng 3000, nó sẽ gửi thông báo đó đến Node server.

2.4. Localtunnel (3)

CÂU HỎI: NẾU laptop trở thành một server khi ta thực thi "node server.js", máy tính khác có kết nối được không?



2.4. Localtunnel (3)



CÂU HỎI: NẾU laptop trở thành một server khi ta thực thi "node server.js", máy tính khác có kết nối được không?

TRẢ LỜI: ĐƯỢC. Và nếu dùng công cụ [localtunnel](#), mọi việc sẽ dễ dàng hơn nữa.

2.4. Localtunnel (4)

Localtunnel là một **công cụ dòng lệnh** và cài đặt được bằng npm.

```
npm install -g package-name
```

- Có thể sử dụng tham số để thiết lập cho nó **phổ quát (globally)** bằng -g
- **Chỉ áp dụng cho các công cụ dòng lệnh**

Vậy để cài localtunnel, ta thực thi:

```
$ npm install -g localtunnel
```

Khi cài thành công thì thực thi qua câu lệnh lt.

2.4. Localtunnel (5)

Sau khi khởi động server bằng câu lệnh:

```
$ node server.js
```

Sau đó chạy localtunnel trong một cửa sổ khác với quy định về cổng của server (chẳng hạn 3000):

```
$ lt --port 3000
```

Localtunnel sẽ cung cấp một địa chỉ URL để người khác có thể truy cập được server cục bộ của bạn.

CHÚ Ý: Công cụ này chỉ dùng cho khi phát triển hoặc chạy thử, **KHÔNG** phải đem đi triển khai cho hệ thống thật!

2.4. Localtunnel (9)

Chú ý:

- Nếu ta hủy đi tiến trình "node server.js", địa chỉ URL sẽ không còn hiệu lực (báo lỗi timeout)
- Nếu nhiều người cùng truy cập URL tại một thời điểm, server chỉ xử lý một yêu cầu tại một thời điểm.
- Nếu máy tính cá nhân làm nhiều việc khác nhau (lướt web; dùng PhotoShop) hiệu năng của server sẽ giảm xuống

2.4. Localtunnel (11)

Nhìn chung: Một máy server riêng lẻ có thể phục vụ ~1000 đến 10,000 request đồng thời

- Sẽ nhận được nhiều request tại một thời điểm
- Sẽ phục vụ song song nhiều request tại một thời điểm.

(Nói cách khác: Trừ khi bạn a) mong đợi > 1000 yêu cầu đồng thời đến máy chủ web của mình và b) kén chọn cách các yêu cầu đó được phục vụ, bạn thực sự không cần phải triển khai máy chủ của mình qua **AWS**, **Google Cloud Platform**, hoặc **IaaS** khác.)

2. Tổng quan NodeJS

2.1. Mô hình hoạt động

2.2. Cài đặt npm

2.3. Câu lệnh node

2.4. Localtunnel

2.5. Ví dụ

2.5. Ví dụ

Lập trình NodeJS cho server

Một mã nguồn đơn giản được thực thi trên server, viết bằng NodeJS:

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

const server = http.createServer();

server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.on('listening', function() {
  console.log('Server running!');
});

server.listen(3000);
```

(THỰC TẾ: Sẽ phải có thư viện hỗ trợ chứ không chỉ đơn giản như đây!!! Ở môn học sẽ được giới thiệu thư viện ExpressJS

2.5. Ví dụ (2)

require()

```
const http = require('http');  
const server = http.createServer();
```

Hàm NodeJS **require()** sẽ nạp một module, tương tự như **import** của **Java** hoặc **include** trong **C++**.

- Có thể dùng **require()** để nạp module định nghĩa sẵn trong **NodeJS**, hoặc các module tự viết.
- Trong ví dụ trên, **'http'** đã được định nghĩa sẵn [HTTP NodeJS module](#)

2.5. Ví dụ (3)

require()

```
const http = require('http');  
const server = http.createServer();
```

- Biến **http** được trả về bởi **require('http')** dùng để truy cập các API hỗ trợ cho **HTTP**:
- **http.createServer()** tạo ra đối tượng **Server**

2.5. Ví dụ (4)

Trên thực tế, sẽ đăng ký sự kiện bằng hàm `Emitter.on`

```
server.on('request', function(req, res) {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World\n');  
});  
  
server.on('listening', function() {  
  console.log('Server running!');  
});
```

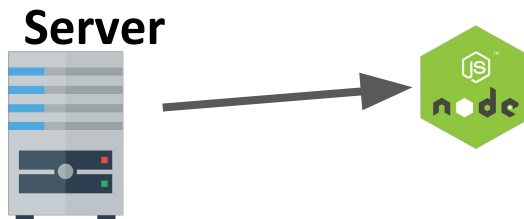
Hàm [on\(\)](#) trong NodeJS tương đương với `addEventListener`.

2.5. Ví dụ (5)

Emitter.on

```
server.on('request', function(req, res) {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World\n');  
});
```

Sự kiện [request](#) sẽ xuất hiện mỗi khi có một HTTP request gửi đến chương trình NodeJS để thực thi.



2.5. Ví dụ (6)

Emitter.on

```
server.on('request', function(req, res) {  
  res.statusCode = 200;  
  res.setHeader('Content-Type', 'text/plain');  
  res.end('Hello World\n');  
});
```

Tham số [req](#) đại diện cho thông tin của request, và [res](#) là nơi ta quyết định phải phản hồi như nào cho client thông qua việc gọi các phương thức của biến đó.

- [statusCode](#): Thiết lập mã HTTP status code.
- [setHeader\(\)](#): Thiết lập HTTP headers.
- [end\(\)](#): Ghi nội dung hồi đáp vào phần body của response và báo server rằng ta đã chuẩn bị xong nội dung hồi đáp.

2.5. Ví dụ (7)

listen() và listening

```
server.on('listening', function() {  
  console.log('Server running!');  
});
```

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

Hàm listen() sẽ đăng ký rằng chương trình chấp nhận mọi thông báo gửi đến **cổng**.

- Sự kiện listening sẽ được xuất hiện khi server phát hiện có gì đó gửi đến cổng.

CÂU HỎI: Cổng là gì?

2.5. Ví dụ (8)

Ports và binding

port: trong ngữ cảnh mạng, đó là một địa chỉ "logic" (trái ngược với địa chỉ vật lý) nơi mà các thông điệp sẽ đi vào (khi thông điệp đã đến với máy chủ)

- Một cổng có giá trị từ 0 đến 65535 (16-bit unsigned integer)

Khi bạn bắt đầu chạy một tiến trình trên server, bạn nhấn với hệ điều hành rằng tiến trình đó sẽ nhận thông báo tại cổng nào, và đó gọi là việc **binding**.

2.5. Ví dụ (10)

Các cổng thường dùng Port

Danh sách các cổng thường dùng cho các mục đích khác nhau:

21: File Transfer Protocol (FTP)

22: Secure Shell (SSH)

23: Telnet remote login service

25: Simple Mail Transfer Protocol (SMTP)

53: Domain Name System (DNS) service

80: Hypertext Transfer Protocol (HTTP) dùng cho World Wide Web

110: Post Office Protocol (POP3)

119: Network News Transfer Protocol (NNTP)

123: Network Time Protocol (NTP)

143: Internet Message Access Protocol (IMAP)

161: Simple Network Management Protocol (SNMP)

194: Internet Relay Chat (IRC)

443: HTTP Secure (HTTPS)

2.5. Ví dụ (11)

Nhiệm vụ của server khi có yêu cầu gửi đến

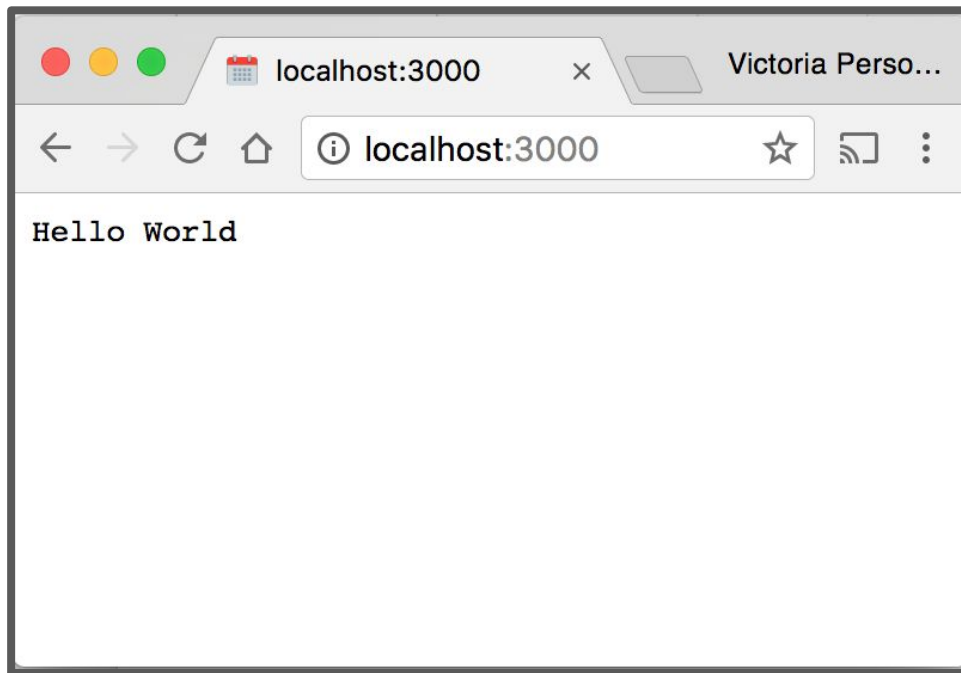
```
server.on('listening', function() {  
  console.log('Server running!');  
});
```

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

Ở đây chương trình sử dụng cổng 3000.

2.5. Ví dụ (14)

Kết quả



2.5. Ví dụ (15)

Node trên servers

Đoạn mã này sẽ trả về
hồi đáp giống hệt nhau
cho mọi yêu cầu .

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

const server = http.createServer();

server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.on('listening', function() {
  console.log('Server running!');
});

server.listen(3000);
```

2.5. Ví dụ (16)

Nạp thư viện HTTP
NodeJS

Khi server nhận
request, gửi lại
"Hello World" dạng
văn bản

Khi server khởi động,
ghi ra log

Bắt đầu lắng nghe
thông báo

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

const server = http.createServer();

server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.on('listening', function() {
  console.log('Server running!');
});

server.listen(3000);
```

2.5. Ví dụ (16)

Node trên servers

Trên thực tế, viết mã NodeJS cho server sẽ rất dài:

- Đăng ký hết tất cả các loại request
- Viết tất cả các hồi đáp cho request
- Sẽ phải rất nhiều dòng mã nguồn cho server

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

http.createServer(function(request, response) {
  var headers = request.headers;
  var method = request.method;
  var url = request.url;
  var body = [];
  request.on('error', function(err) {
    console.error(err);
  }).on('data', function(chunk) {
    body.push(chunk);
  }).on('end', function() {
    body = Buffer.concat(body).toString();
    // BEGINNING OF NEW STUFF

    response.on('error', function(err) {
      console.error(err);
    });

    response.statusCode = 200;
    response.setHeader('Content-Type', 'application/json');
    // Note: the 2 lines above could be replaced with this next one:
    // response.writeHead(200, {'Content-Type': 'application/json'})

    var responseBody = {
      headers: headers,
      method: method,
      url: url,
      body: body
    };

    response.write(JSON.stringify(responseBody));
    response.end();
    // Note: the 2 lines above could be replaced with this next one:
    // response.end(JSON.stringify(responseBody))

    // END OF NEW STUFF
  });
}).listen(8080);
```

Mục lục

1. Tổng quan lập trình Server
2. Tổng quan NodeJS
3. **ExpressJS**
4. Xử lý tham số/trả về
5. Async/await

3. Express JS

3.1. Khái niệm

3.2. Route

3.3. Truy cập file

ExpressJS

Hiện nay, người ta thường có xu hướng dùng thư viện ExpressJS để lập trình NodeJS:

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```

ExpressJS

Tuy vậy, **ExpressJS** không có sẵn trong **NodeJS API**. Nếu không cài gói này, ta sẽ gặp lỗi:

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

```
module.js:327  
  throw err;  
  ^
```

```
Error: Cannot find module 'express'  
    at Function.Module._resolveFilename
```

Cài ExpressJS qua npm.

3. Express JS

3.1. Khái niệm

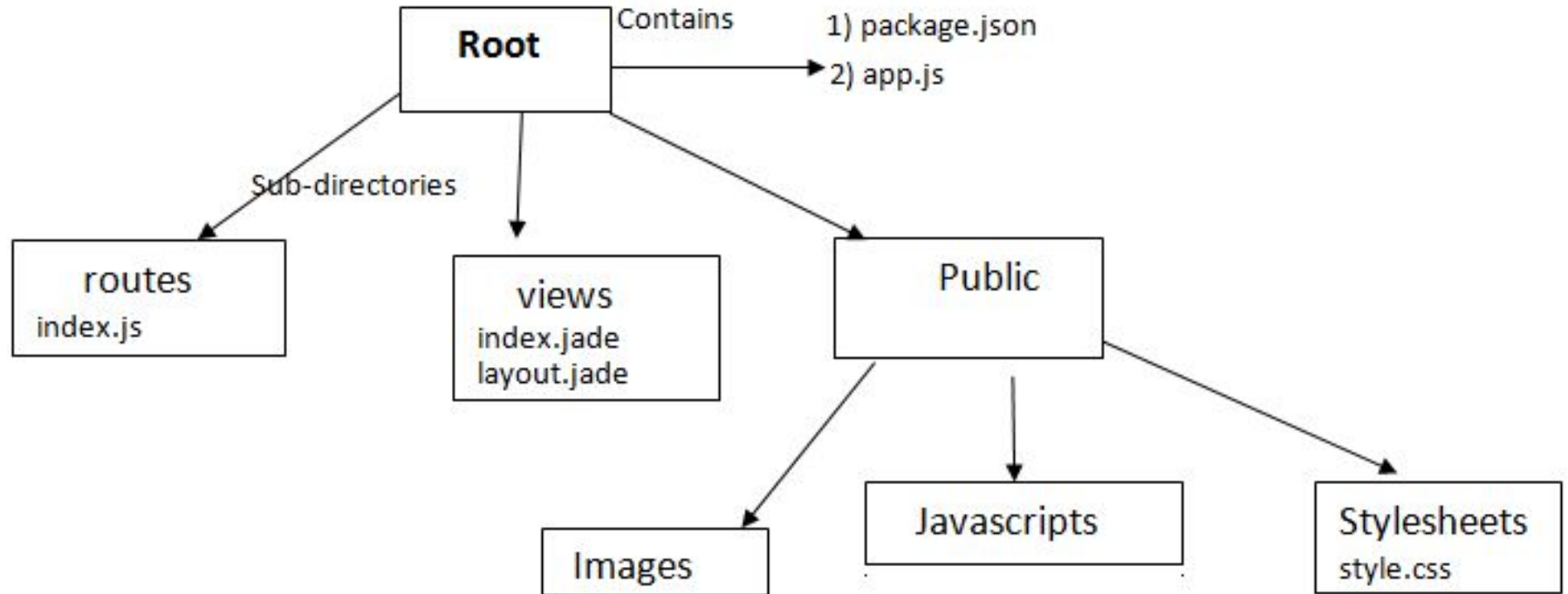
3.2. Route

3.3. Truy cập file

3.1. Khái niệm

- ❑ **ExpressJS** là một framework được xây dựng trên nền tảng của **NodeJS**.
- ❑ Nó cung cấp các tính năng mạnh mẽ để phát triển web hoặc mobile.
- ❑ **ExpressJS** hỗ trợ các giao thức HTTP và tạo ra các RESTful API vô cùng mạnh mẽ, dễ sử dụng.

3.1. Khái niệm (2)



3.1. Khái niệm (3)

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

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

3.1. Khái niệm (4)

Dưới đây là mã cho chương trình bên server mà **KHÔNG DÙNG ExpressJS**:

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

const server = http.createServer();

server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

server.on('listening', function() {
  console.log('Server running!');
});

server.listen(3000);
```

3.1. Khái niệm (5)

Và **CÓ DÙNG Express:**

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

app.get('/', function (req, res) {
  res.send('Hello World!');
})

app.listen(3000, function () {
  console.log('Example app listening on port 3000!');
})
```


3. Express JS

3.1. Khái niệm

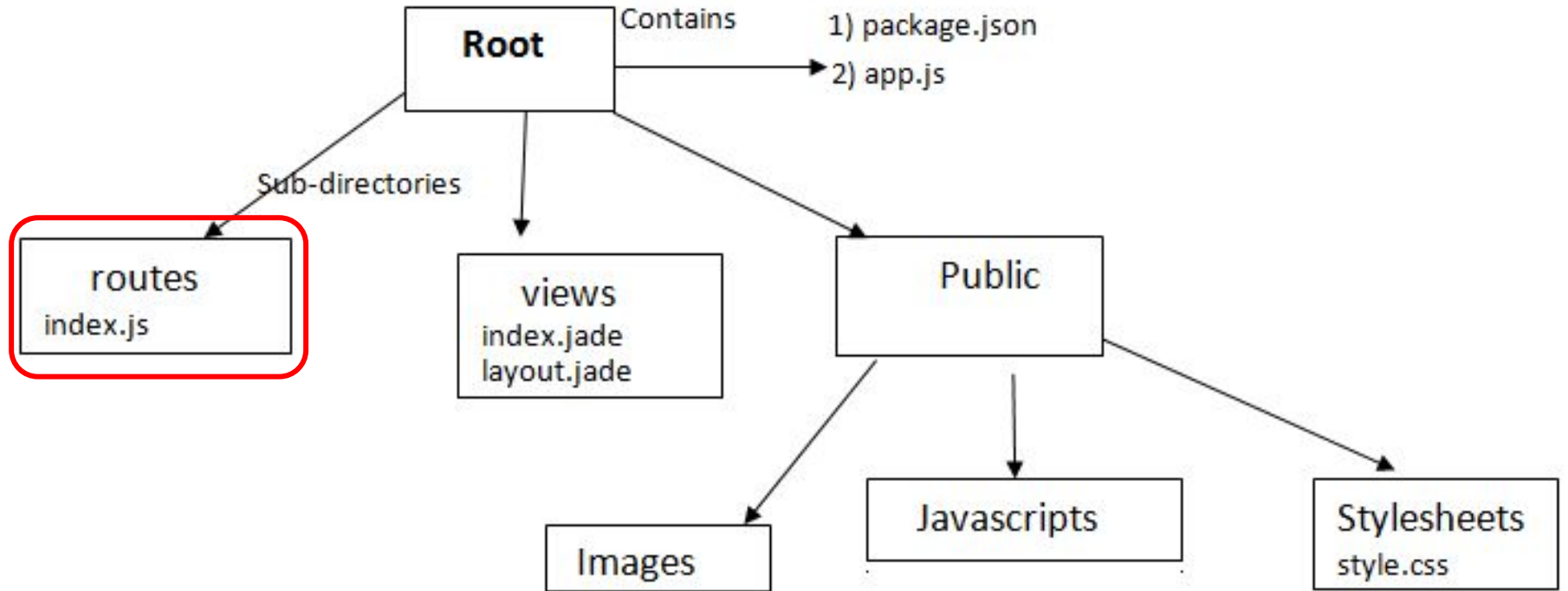
3.2. Route

3.3. Truy cập file

3.2. Route

- ❑ Route là một thành phần cực kỳ quan trọng của một website,
 - nó giúp website biết được người dùng truy cập đến nơi nào của trang web,
 - từ đó cho phép phân ra các đoạn mã nguồn để phản hồi lại một cách thích hợp.
- ❑ Trong ExpressJs, route được tích hợp sẵn và dễ dàng sử dụng.
- ❑ Route hỗ trợ đầy đủ các giao thức GET, PUT, POST, DELETE

3.2. Route



3.2. Route (2)

Chúng ta có thể đặc tả các [route trong ExpressJS](#):

```
app.get('/', function (req, res) {  
  res.send('Main page!');  
});
```

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

```
app.post('/hello', function (req, res) {  
  res.send('POST hello!');  
});
```

3.2. Route (3)

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

Cú pháp chung

`app.method(path, handler)`

- Đặc tả cách server xử lý các ***giao thức*** HTTP requests được gửi đến URL/***path***
- Đoạn mã trên có nghĩa là:
 - Khi nhận được một GET request đến <http://localhost:3000/hello>, hồi đáp với đoạn văn bản "GET hello!"

3.2. Route (4)

```
// respond with "Hello World!" on the homepage
```

```
app.get('/', function (req, res) {
```

```
    res.send('Hello World!');
```

```
});
```

```
// accept POST request on the homepage
```

```
app.post('/', function (req, res) {
```

```
    res.send('Got a POST request');
```

```
});
```

```
// accept PUT request at /user
```

```
app.put('/user', function (req, res) {
```

```
    res.send('Got a PUT request at /user');
```

```
});
```

```
// accept DELETE request at /user
```

```
app.delete('/user', function (req, res) {
```

```
    res.send('Got a DELETE request at /user');
```

```
});
```

3.2. Route (5)

Hoặc gọn hơn sẽ viết là:

```
app.route('/user') .get(function (req, res) {  
    res.send('Hello World!');  
})  
  .post(function (req, res) {  
    res.send('Got a POST request');  
})  
  .put(function (req, res) {  
    res.send('Got a PUT request at /user');  
})  
  .delete(function (req, res) {  
    res.send('Got a DELETE request at /user');  
});
```

3.2. Route (6)

Thông thường chúng ta thường tách các đoạn code route ra thành những file route riêng biệt để tiện quản lý, ví dụ: userRouter.

Tạo file userRouter.js với mã như sau:

```
module.exports = function(app) {  
  app.route('/user')  
    .get(function (req, res) {  
      res.send('Hello World!');  
    })  
    .post(function (req, res) {  
      res.send('Got a POST request');  
    })  
    .put(function (req, res) {  
      res.send('Got a PUT request at /user');  
    })  
    .delete(function (req, res) {  
      res.send('Got a DELETE request at /user');  
    });  
}
```


3.2. Route (7)

Khi đó ta sẽ chỉnh lại file server.js như sau:

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

```
var app = express();
```

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

```
userRouter(app);
```

```
var server = app.listen(3000, function () {
```

```
    var host = server.address().address
```

```
    var port = server.address().port
```

```
    console.log("Ung dung Node.js dang hoat dong tai dia chi: http://%s:%s", host, port)
```

```
});
```

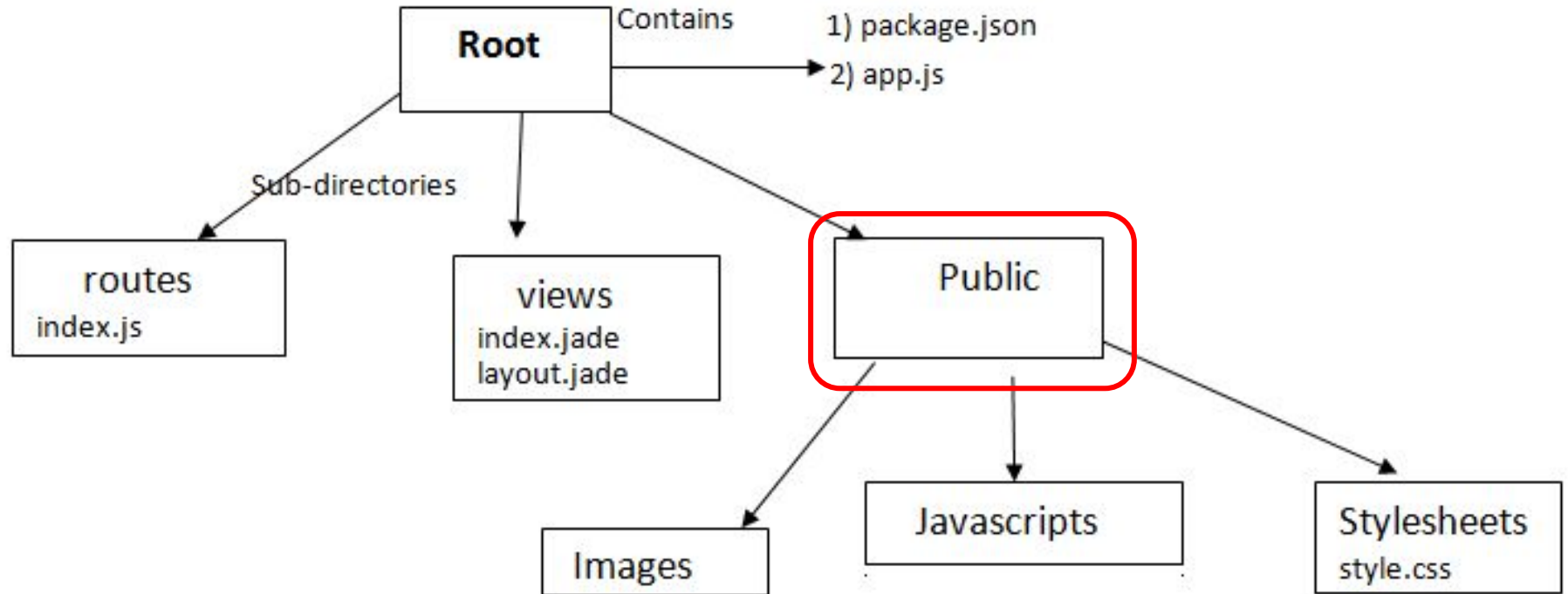
3. Express JS

3.1. Khái niệm

3.2. Route

3.3. Truy cập file

3.3. Truy cập file



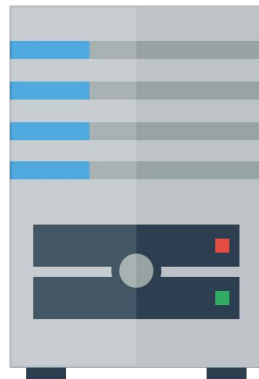
3.3. Truy cập file (2)

Nói chung, khi nhập một địa chỉ URL lên trình duyệt, yêu cầu đó được gửi đến server và thường được coi là một lời gọi **API**.

URL đã tạo ra một **parameterized request**, và web server tự động đưa ra các hồi đáp.

Chẳng hạn như ví dụ sau:

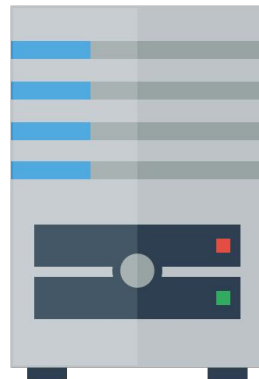
```
app.get('/hello', function (req, res) {  
  res.send('GET hello!');  
});
```



3.3. Truy cập file (3)

Nhưng nhiều lúc chỉ cần trả về một file có nội dung tĩnh cho người dùng, tức trả về **đường dẫn** đến file đó trên server:

- Web server sẽ lấy file đó ra từ trong máy chủ để trả về cho người dùng



Chúng ta có thể cần thiết lập web server trả về các file tĩnh

"statically," tức thay vì coi URL là các lời gọi API, ta coi đó như các yêu cầu truy cập file.

3.3. Truy cập file (4)

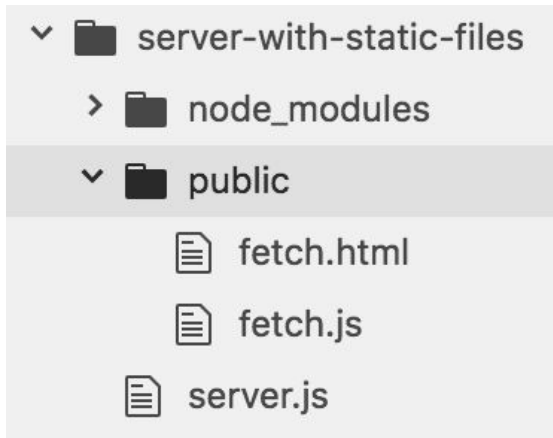
```
const express = require('express');  
const app = express();
```

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

```
app.get('/', function (req, res) {  
  res.send('Main page!');  
});
```

Đoạn mã trên giúp web server trả về được các file nằm trong thư mục 'public'.

3.3. Truy cập file (5)



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

Chẳng hạn

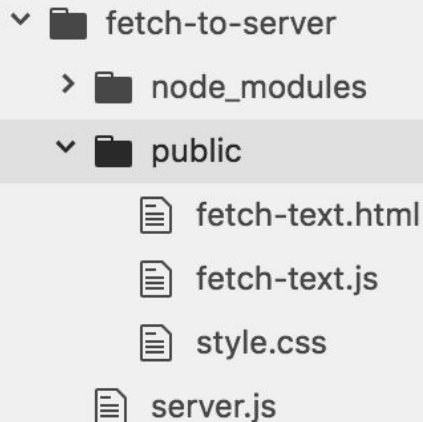
<http://localhost:3000/fetch.html>

<http://localhost:3000/fetch.js>

ExpressJS sẽ tìm kiếm trong danh sách các file ở thư mục tĩnh, vậy tên của thư mục này ("public" trong ví dụ) không phải là một đường dẫn của URL

3.3. Truy cập file (6)

Có thể đặt các tệp **HTML/CSS/JS** vào thư mục **public** đó:



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

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

```
app.get('/', function (req, res) {  
  res.send('Main page!');  
});
```


Mục lục

1. Tổng quan lập trình Server
2. Tổng quan NodeJS
3. ExpressJS
4. **Xử lý tham số/trả về**
5. Async/await

4. Xử lý tham số/trả về

4.1. Route parameters

4.2. Trả về JSON

4.3. Message body

4.4. Query Parameter

4. Xử lý tham số

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

ExpressJS có các đối tượng của lớp [Request](#) và [Response](#):

- req là đối tượng của lớp Request
- res là đối tượng của lớp Response
- [res.send\(\)](#) gửi một HTTP response với nội dung
 - Gửi dữ liệu mặc định là kiểu "text/html"

4. Xử lý tham số

Để gửi dữ liệu từ client lên đến server, chúng ta có nhiều cách khác nhau.

Ví dụ: Spotify Album API

`https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9`

- Phần cuối cùng URL là **tham số** đại diện cho album id, 7aDBFWp72Pz4NZEtVBANi9

Tham số chèn vào URL được gọi là "**route parameter**."

4. Xử lý tham số/trả về

4.1. Route parameters

4.2. Trả về JSON

4.3. Message body

4.4. Query Parameter

4.1. Route parameters

CÂU HỎI: Làm thế nào để đọc route parameters ở phía server?

TL: có thể dùng cú pháp **:*variableName*** trên đường dẫn để gửi tham số dạng route parameter:

```
app.get('/hello/:name', function (req, res) {  
  const routeParams = req.params;  
  const name = routeParams.name;  
  res.send('GET: Hello, ' + name);  
});
```

Server đọc được các route parameters thông qua **req.params**.

4.1. Route parameters (2)

```
app.get('/hello/:name', function (req, res) {  
  const routeParams = req.params;  
  const name = routeParams.name;  
  res.send('GET: Hello, ' + name);  
});
```



4.1. Route parameters (3)

Có thể định nghĩa nhiều route parameters trong URL:

```
app.get('/flights/:from-:to', function (req, res) {  
  const routeParams = req.params;  
  const from = routeParams.from;  
  const to = routeParams.to;  
  res.send('GET: Flights from ' + from + ' to ' + to);  
});
```



4. Xử lý tham số/trả về

4.1. Route parameters

4.2. Trả về JSON

4.3. Message body

4.4. Query Parameter

4.2. Trả về JSON

Nếu muốn trả về định dạng JSON, ta có thể dùng hàm `res.json(object)`:

```
app.get('/', function (req, res) {  
  const response = {  
    greeting: 'Hello World!',  
    awesome: true  
  }  
  res.json(response);  
});
```

Tham số gửi vào [res.json\(\)](#) phải là đối tượng JavaScript.

4.2. Trả về JSON (2)

```
function onLookupWord(req, res) {  
  const routeParams = req.params;  
  const word = routeParams.word;  
  
  const key = word.toLowerCase();  
  const definition = englishDictionary[key];  
  
  res.json({  
    word: word,  
    definition: definition  
  });  
}  
app.get('/lookup/:word', onLookupWord);
```

4.2. Trả về JSON (3) - phía client

```
async function onSearch(event) {  
  event.preventDefault();  
  const input = document.querySelector('#word-input');  
  const word = input.value.trim();  
  
  const results = document.querySelector('#results');  
  results.classList.add('hidden');  
  const result = await fetch('/lookup/' + word);  
  const json = await result.json();  
  
  results.classList.remove('hidden');  
  const wordDisplay = results.querySelector('#word');  
  const defDisplay = results.querySelector('#definition');  
  wordDisplay.textContent = json.word;  
  defDisplay.textContent = json.definition;  
}
```

4. Xử lý tham số/trả về

4.1. Route parameters

4.2. Trả về JSON

4.3. Message body

4.4. Query Parameter

4.3. Message body

Client-side:

Đặt JSON vào trong **message body** bằng hàm `fetch()`:

```
const message = {  
  name: 'Victoria',  
  email: '██████████.edu'  
};  
  
const serializedMessage = JSON.stringify(message);  
fetch('/helloemail', { method: 'POST', body: serializedMessage })  
  .then(onResponse)  
  .then(onTextReady);
```

4.3. Message body (2)

Server-side: Xử lý lấy dữ liệu từ message body trong NodeJS/Express:

```
app.post('/helloemail', function (req, res) {  
  let data = '';  
  req.setEncoding('utf8');  
  req.on('data', function(chunk) {  
    data += chunk;  
  });  
  
  req.on('end', function() {  
    const body = JSON.parse(data);  
    const name = body.name;  
    const email = body.email;  
    res.send('POST: Name: ' + name + ', email: ' + email);  
  });  
});
```

Có thể dùng thư viện body-parser

4.3. Message body (3)

Có thể lấy được các thông tin cần thiết từ `req.body`:

```
app.post('/helloparsed', jsonParser, function (req, res) {  
  const body = req.body;  
  const name = body.name;  
  const email = body.email;  
  res.send('POST: Name: ' + name + ', email: ' + email);  
});
```


4.3. Message body (4)

Có thể lấy được các thông tin cần thiết từ `req.body`:

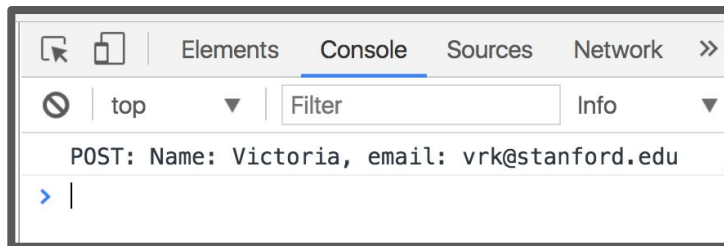
```
app.post('/helloparsed', jsonParser function (req, res) {  
  const body = req.body;  
  const name = body.name;  
  const email = body.email;  
  res.send('POST: Name: ' + name + ', email: ' + email);  
});
```

Chú ý phải sử dụng `jsonParser` làm tham số khi định nghĩa route này.

4.3. Message body (5)

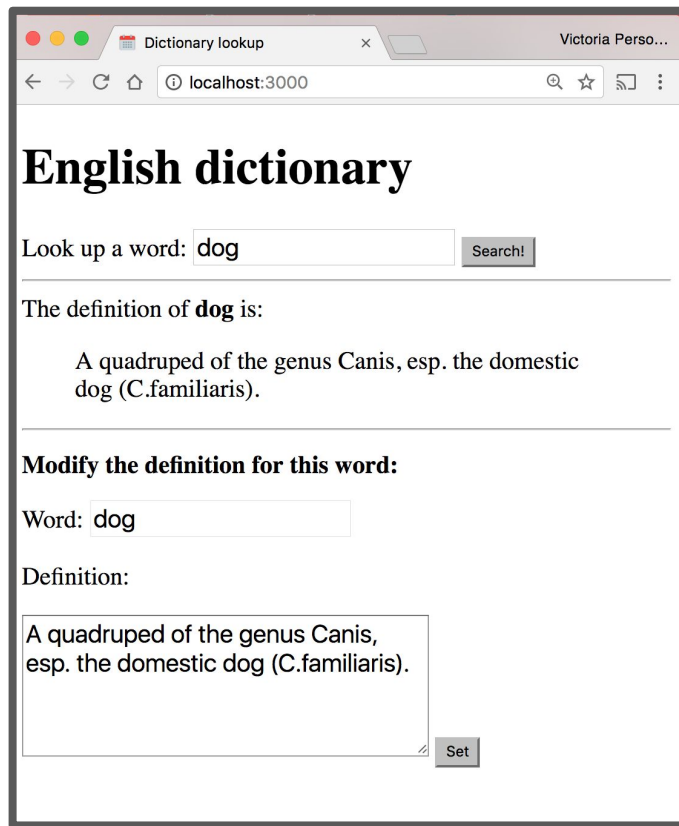
Cuối cùng, ta cần thêm JSON content-type headers trong `fetch()`:

```
const message = {
  name: 'Victoria',
  email: 'vrk@stanford.edu'
};
const fetchOptions = {
  method: 'POST',
  headers: {
    'Accept': 'application/json',
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(message)
};
fetch('/helloparsed', fetchOptions)
  .then(onResponse)
  .then(onTextReady);
```



CÂU HỎI

Làm thế nào để cho phép người dùng có thể gửi đề xuất lên từ điển?



The screenshot shows a web browser window titled "Dictionary lookup" with the address bar displaying "localhost:3000". The page content includes a search bar with the text "Look up a word:" followed by an input field containing "dog" and a "Search!" button. Below this, it states "The definition of **dog** is:" followed by the definition "A quadruped of the genus Canis, esp. the domestic dog (C.familiaris)." There is a section titled "Modify the definition for this word:" with a "Word:" input field containing "dog" and a "Definition:" input field containing the same definition. A "Set" button is located at the bottom right of the definition input field.

Trả lời: server-side

```
async function onSetWord(req, res) {  
  const routeParams = req.params;  
  const word = routeParams.word;  
  const definition = req.body.definition;  
  const key = word.toLowerCase();  
  englishDictionary[key] = definition;  
  await fse.writeFileSync('./dictionary.json', englishDictionary);  
  res.json({ success: true});  
}  
app.post('/set/:word', jsonParser, onSetWord);
```

Trả lời: phía fetch()

```
async function onSet(event) {
  event.preventDefault();
  const setWordInput = results.querySelector('#set-word-input');
  const setDefInput = results.querySelector('#set-def-input');
  const word = setWordInput.value;
  const def = setDefInput.value;

  const message = {
    definition: def
  };
  const fetchOptions = {
    method: 'POST',
    headers: {
      'Accept': 'application/json',
      'Content-Type': 'application/json'
    },
    body: JSON.stringify(message)
  };
  await fetch('/set/' + word, fetchOptions);
}
```

4. Xử lý tham số/trả về

4.1. Route parameters

4.2. Trả về JSON

4.3. Message body

4.4. Query Parameter

4.4. Query parameters

Còn một loại tham số nữa là query

Spotify Search API cũng có giai đoạn từng ưa chuộng sử dụng query parameters:

Example: Spotify Search API

<https://api.spotify.com/v1/search?type=album&q=beyonce>

- Ở đây có hai tham số gửi lên Spotify search:
 - type, với giá trị là album
 - q, với giá trị là beyonce

4.4. Query parameters (2)

CÂU HỎI: Làm thế nào để đọc query parameters trên server?

TL: Sử dụng `req.query`:

```
app.get('/hello', function (req, res) {  
  const queryParams = req.query;  
  const name = queryParams.name;  
  res.send('GET: Hello, ' + name);  
});
```



So sánh

Chúng ta có thể có nhiều cách để gửi tham số lên server:

1. Route parameters
2. GET request với query parameters
(KHÔNG KHUYẾN KHÍCH DÙNG POST với query parameters)
3. POST request với message body

GET và POST

- Dùng **GET** requests để đọc lấy dữ liệu, không phải để ghi dữ liệu
 - Dùng **POST** requests để ghi dữ liệu, không phải để đọc dữ liệu
- Có thể dùng nhiều giao thức HTTP khác:
- **PATCH**: Cập nhật một tài nguyên cụ thể
 - **DELETE**: Xóa đi một tài nguyên cụ thể

Trên kia chỉ là quy ước, tuy vậy nên tuân thủ theo đúng mục đích của các giao thức HTTP vốn đã được chấp thuận trên toàn thế giới.

Route params và Query params

Nói chung nên tuân thủ theo quy tắc (vẫn có Spotify API vi phạm):

- Dùng **route parameters** cho các tham số bắt buộc phải có trong request
- Dùng **query parameters** cho:
 - Tham số tùy chọn
 - Tham số có thể có dấu cách ở trong

Ví dụ: Spotify API

Về Spotify API hầu hết tuân thủ theo các quy ước trên:

<https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9>

- Tham số Album ID là bắt buộc và là một route parameter.

<https://api.spotify.com/v1/search?type=album&q=the%20weeknd&limit=10>

- q là bắt buộc nhưng nó có dấu cách nên cho vào loại query parameter
- limit là tùy chọn nên cho vào loại query parameter
- type là bắt buộc và lại là query parameter (vi phạm quy ước)

Cả hai loại URL kia đều là GET requests

Mục lục

1. Tổng quan lập trình Server
2. Tổng quan NodeJS
3. ExpressJS
4. Xử lý tham số/trả về
5. **Async/await**

5. Async/await

5.1. Xử lý đơn luồng

5.2. Call stack

5.3. Asynchronous và await

5. Async/await

5.1. Xử lý đơn luồng

5.2. Call stack

5.3. Asynchronous và await

5.1. Xử lý đơn luồng

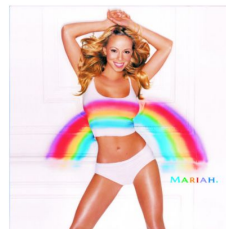
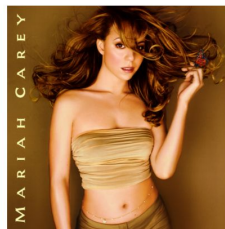
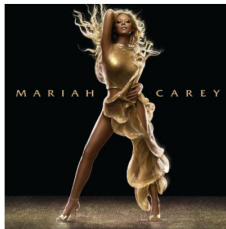
Chúng ta xây dựng trang web hiển thị tất cả các album của ca sĩ Mariah Carey lưu trong [albums.json](#) và cho phép người dùng sắp xếp:

Mariah Carey's albums

By year, descending

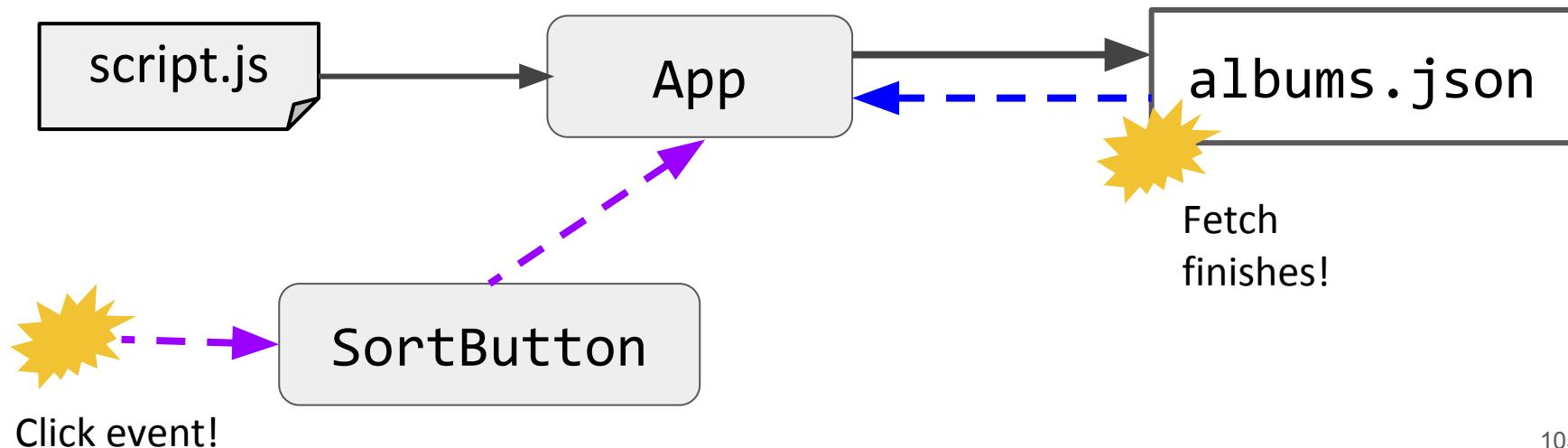
By year, ascending

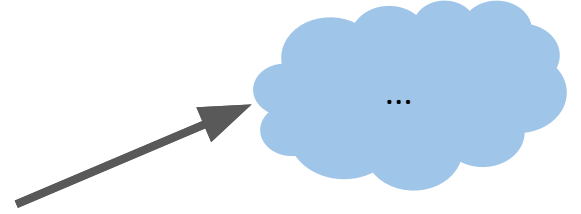
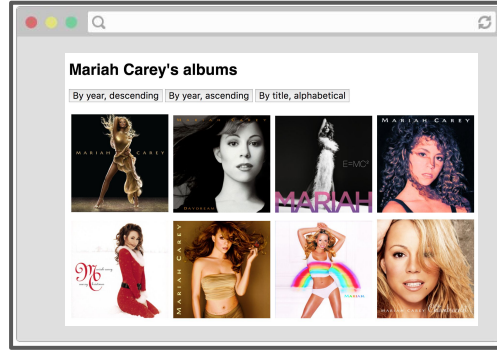
By title, alphabetical



Sự kiện bất đồng bộ

Khi lập trình thì ta giả sử hàm `fetch()` sẽ hoàn thành trước khi người dùng nhấn nút sắp xếp nhưng trong điều kiện mạng kém, điều đó có thể không đúng.





By year, descending

Người dùng nhấn
vào nút sắp xếp



5.1. Xử lý đơn luồng (2)

Vấn đề là như sau:

- Có thể có 2+ sự kiện xảy ra tại các thời điểm không biết trước được, và hai sự kiện này có thể **phụ thuộc** lẫn nhau



Giải pháp

Chúng ta có thể ngăn chặn việc đó xảy ra bằng cách:

- Vô hiệu hóa các nút cho đến khi JSON tải về thành công
- Hoặc không hiển thị các nút cho đến khi JSON tải về thành công
- Hoặc không hiển thị TOÀN BỘ giao diện cho đến khi JSON tải về thành công
- Nhưng như vậy, vẫn có thể xảy ra trường hợp khi xâu JSON đã tải về hết, thì các bức ảnh của album vẫn cần nhiều thời gian để hiện lên được hết

Image loading

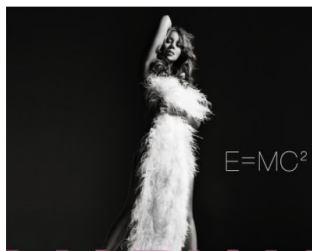
Có khả năng xảy ra như này

Mariah Carey's albums

By year, descending


By year, ascending

By title, alphabetical




Lúc đang hiển thị ảnh của album (hàm `_onJsonReady` chưa kết thúc) thì người dùng nhấn nút sắp xếp

Liệu có vừa sắp xếp vừa hiển thị các ảnh đang được tải về?




```
loadAlbums() {  
  fetch(JSON_PATH)  
    .then(this._onResponse)  
    .then(this._onJsonReady);  
}
```

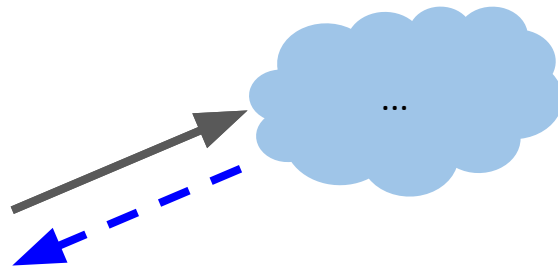
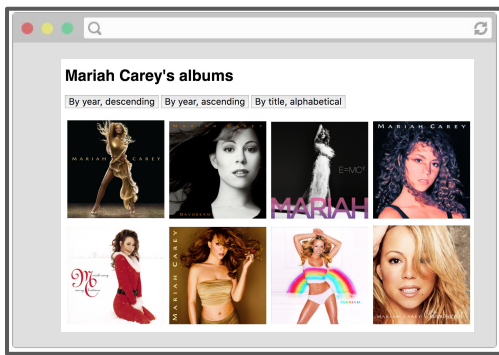


```
_onJsonReady(json) {  
  this.albumInfo = json.albums;  
  this._renderAlbums();  
}
```

```
_onResponse(response) {  
  return response.json();  
}
```



```
_sortAlbums(sortFunction) {  
  this.albumInfo.sort(sortFunction);  
  this._renderAlbums();  
}
```



Câu trả lời là KHÔNG, vì
JavaScript là ngôn ngữ đơn
luồng.

```
_sortAlbums(sortFunction) {  
  this.albumInfo.sort(sortFunction);  
  this._renderAlbums();  
}
```

```
_onJsonReady(json) {  
  this.albumInfo = json.albums;  
  this._renderAlbums();  
}
```

Đơn luồng trong JS

- Tạo ra lớp Album cho mọi album trong tệp JSON
- Với mỗi album, tạo ra đối tượng DOM image


```
_renderAlbums() {  
  const albumContainer = document.querySelector('#album-container');  
  albumContainer.innerHTML = '';  
  for (const info of this.albumInfo) {  
    const album = new Album(albumContainer, info.url);  
  }  
}
```

CÂU HỎI: Do JavaScript là ngôn ngữ đơn luồng, thì liệu có nghĩa là các bức ảnh sẽ được tải lần lượt từng cái một?

```
class Album {  
  constructor(albumContainer, imageUrl) {  
    // Same as document.createElement('img');  
    const image = new Image();  
    image.src = imageUrl;  
    albumContainer.append(image);  
  }  
}
```


Network tab của Chrome

Khi nhìn vào Network tab của Chrome, ta sẽ thấy vài bức ảnh cùng được tải đồng thời:



Name	Status	Type	Initiator	Size	Time	Waterfall	
0638f0ddf70003cb94b43aa5e4004d85...	200	jpeg	Other	4.0 KB	13.25 s		
bca35d49f6033324d2518656531c9a89...	200	jpeg	Other	4.0 KB	13.25 s		
82f13700dfa78fa877a8cdec725ad552c...	200	jpeg	Other	451 B	13.25 s		
676275b41e19de3048fddf72937ec0db...	200	jpeg	Other	2.7 KB	13.25 s		
2424877af9fa273690b688462c5afbad6...	200	jpeg	Other	452 B	13.25 s		
dca82bd9c1ccae90b09972027a408068...	200	jpeg	Other	453 B	557 ms		
0638f0ddf70003cb94b43aa5e4004d85...	200	jpeg	Other	454 B	696 ms		
bca35d49f6033324d2518656531c9a89...	200	jpeg	Other	451 B	790 ms		
82f13700dfa78fa877a8cdec725ad552c...	200	jpeg	Other	451 B	Pending		
676275b41e19de3048fddf72937ec0db...	200	jpeg	Other	450 B	Pending		
2424877af9fa273690b688462c5afbad6...	200	jpeg	Other	452 B	Pending		

CÂU HỎI: Do JavaScript là ngôn ngữ đơn luồng, thì tại sao các bức ảnh được tải song song?

5. Async/await

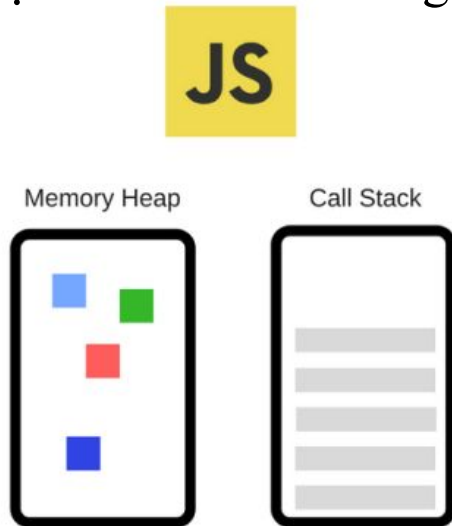
5.1. Xử lý đơn luồng

5.2. Call stack

5.3. Asynchronous và await

5.2. Call stack

- Memory Heap: cấp phát bộ nhớ sẽ diễn ra ở đây.
- Call Stack: cấu trúc dữ liệu nơi chứa các lời gọi hàm khi code được thực thi.



5.2. Call stack (2)

- JavaScript là một ngôn ngữ đơn luồng, chỉ có một Call Stack.
 - *Call Stack là một cấu trúc dữ liệu dạng ngăn xếp (stack) dùng để chứa thông tin về hoạt động của chương trình máy tính trong lúc thực thi.*
- Khi debug từng dòng lệnh, các IDE sẽ cung cấp luôn một giao diện để chúng ta xem call stack hiện tại.
- Nôm na là khi bạn debug/step đến một function A, thì A sẽ được **push** (on top) vào call stack. Sau khi A thực thi xong và trả về kết quả, A sẽ bị **pop** ra khỏi stack.

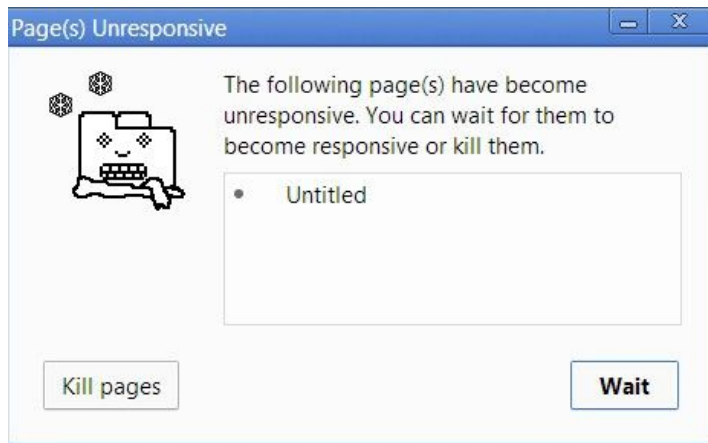
5.2. Call stack (3)

- Nhìn chung, viết code đơn luồng (single-threaded) thường đơn giản hơn khi KHÔNG quan tâm tới lập trình đa luồng (multi-threaded).
- Nhưng bù lại single-threaded cũng rất hạn chế, do JS chỉ có một Call Stack, chuyện gì sẽ xảy ra khi mã nguồn phải xử lý các tác vụ nặng?
 - Như xử lý ảnh chẳng hạn.

```
function resize() {  
    console.log("starts resizing the image.. It may take 1 hour.");  
    // user1: oops, okay lets wait  
    // user2: or maybe I will go home and back on tomorrow  
}
```

5.2. Call stack (4)

- Khi hàm **resize()** đi vào trong Call Stack và bắt đầu thực thi, trình duyệt sẽ chặn các công việc khác trong tab này,
- Các tác vụ khác kể cả **render** phải chờ => gây ảnh hưởng đến trải nghiệm.
- Các trình duyệt hiện đại, như Chrome sẽ cảnh báo này lên khi bị bắt chờ quá lâu.




5.2. Call stack (5)

- Dùng các hàm asynchronous callback, là các hàm không gây chặn (non-blocking) các tác vụ khác.

```
function main() {  
    console.log("Hi!"); //sẽ in "Hi!" ra đầu tiên.  
    setTimeout(function timeout() {  
        console.log("There!");  
    }, 5000); //được gọi với một async callback là timeout  
    console.log("Welcome to loupe!"); //Khởi lệnh (*)  
}  
main();
```

- Trình duyệt không chờ 5s mới thực thi (*) mà in ra "Welcome to loupe!" ngay sau "Hi!". Rồi sau đó một lúc thì "There!" mới xuất hiện.

Call stack + setTimeout



```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call Stack

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
→ console.log('Point A');  
  setTimeout(onTimerDone, 3000);  
  console.log('Point B');
```

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}
```

→

```
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

`console.log('Point A');`

(global function)


Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

(global function)

Call stack + setTimeout



```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call Stack

setTimeout(...);

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```



(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}
```



```
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

console.log('Point B');

(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```



(global function)

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```


Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

onTimerDone()

Call stack + setTimeout

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call Stack

console.log('Point C');

onTimerDone()

Call stack + setTimeout

```
function onTimerDone() {  
  console.log('Point C');  
  → const h1 = document.querySelector('h1');  
    h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call Stack

onTimerDone()

Call stack + setTimeout

```
function onTimerDone() {  
  console.log('Point C');  
  → const h1 = document.querySelector('h1');  
    h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call Stack

querySelector('h1');

onTimerDone()

Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

onTimerDone()

Call stack + setTimeout

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
} →  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call Stack

onTimerDone()


Call stack + setTimeout

Call Stack

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```

Call stack + setTimeout

```
function onTimerDone() {  
  console.log('Point C');  
  const h1 = document.querySelector('h1');  
  h1.textContent = 'loaded';  
}  
  
console.log('Point A');  
setTimeout(onTimerDone, 3000);  
console.log('Point B');
```



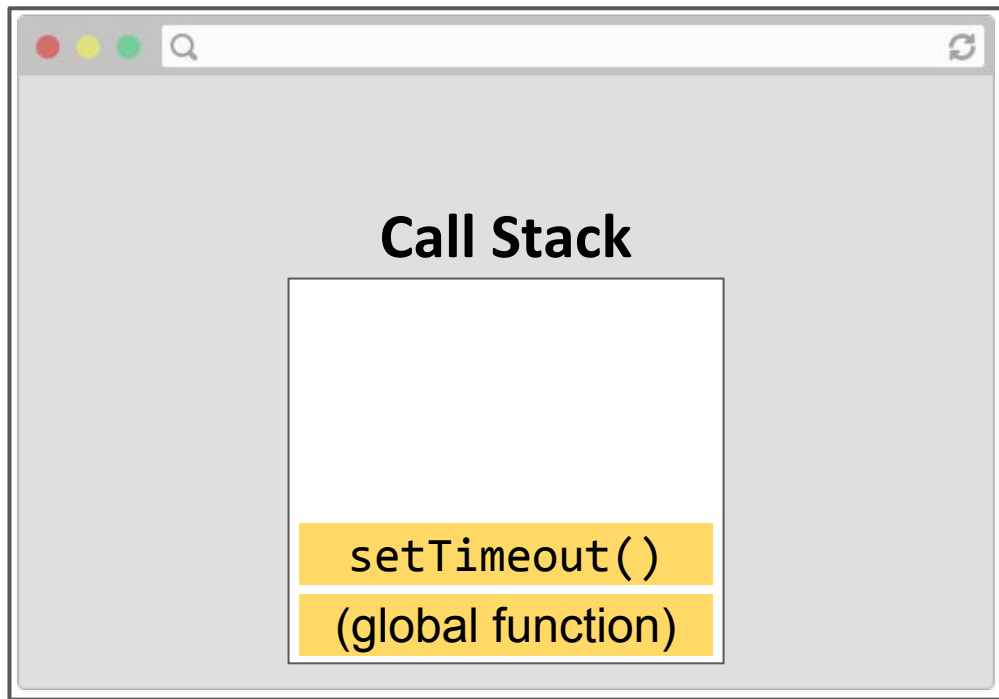
Cái gì đã đảm bảo rằng hàm
onTimerDone được thực thi sau một
khoảng thời gian

Call Stack

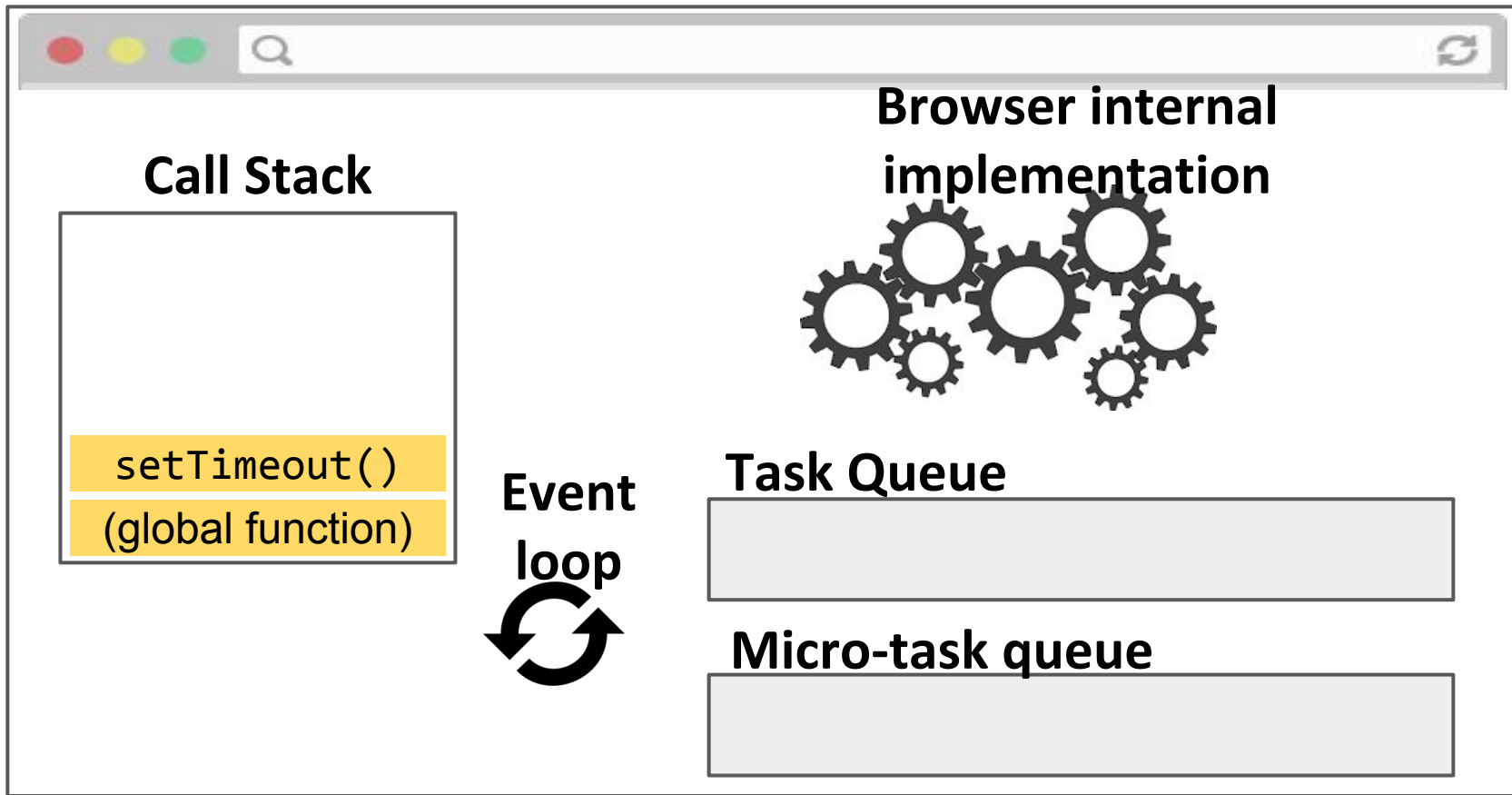
setTimeout(...);

(global function)

5.2. Call stack (6)

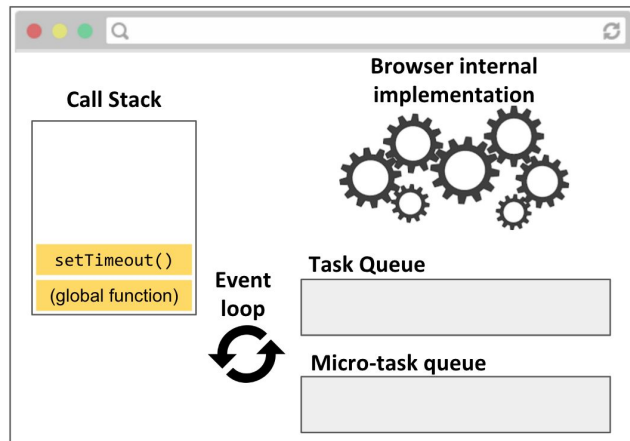


Trình duyệt đã hỗ trợ điều này



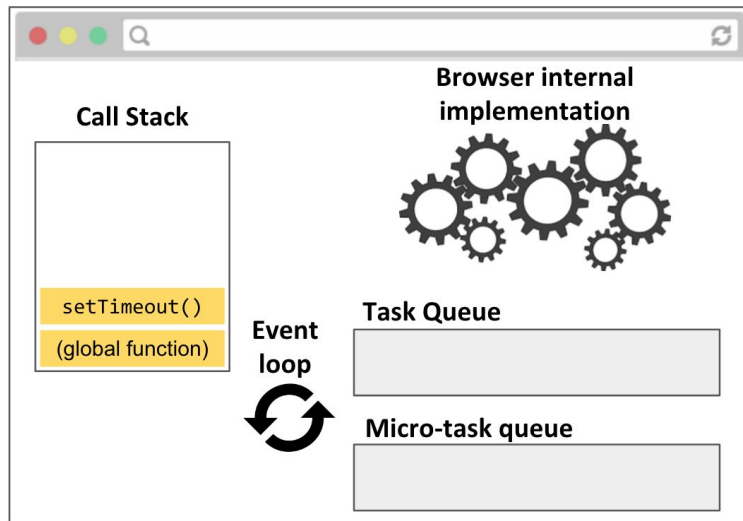
Trình duyệt có hai hàng đợi: Task và Micro-task

5.2. Call stack (7)



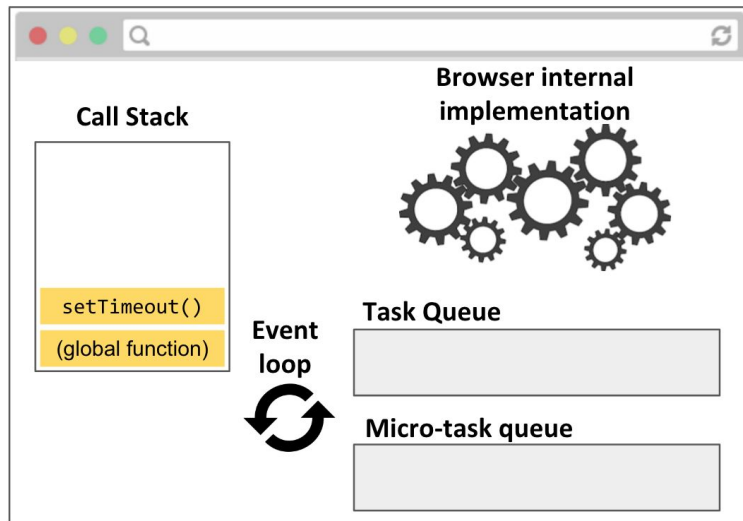
- **Call stack:** là của JavaScript runtime. Nơi chứa các JavaScript commands, functions cần thực thi.
- **Browser internal implementation:** cài đặt bằng C++ để thực thi các mã JavaScript, vd. `setTimeout`, `element.classList.add('style')`, v.v..

5.2. Call stack (8)



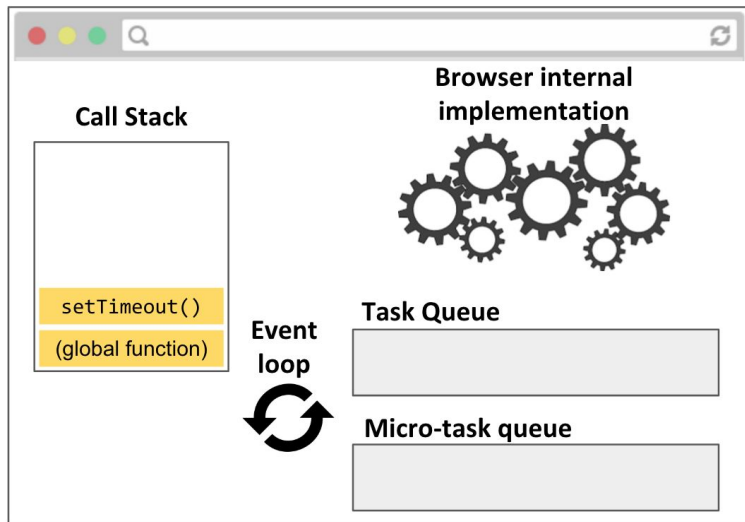
- **Task Queue:** Nơi trình duyệt lưu trữ danh sách các hàm cần callback như là `setTimeout` hoặc `addEventListener`. Nhiệm vụ callback chúng được cho vào hàng đợi Task

5.2. Call stack (9)



- **Micro-task Queue:** Nơi lưu trữ các Promise, là các nhiệm vụ đặc biệt có độ ưu tiên cao hơn các nhiệm vụ trong hàng đợi Task

5.2. Call stack (10)



Event loop: thực thi các nhiệm vụ trong các hàng đợi.

- Khi call stack trống, event loop lấy ra nhiệm vụ tiếp theo trong các hàng đợi và cho vào call stack.
- Nhiệm vụ trong Micro-task có độ ưu tiên cao hơn

5. Async/await

5.1. Xử lý đơn luồng

5.2. Call stack

5.3. Asynchronous và await

5.3. Asynchronous và await

- Là cơ chế cho phép khai báo các hàm để chúng được xử lý bất đồng bộ
- Một khi hàm - được khai báo **async** - (nó vốn cần nhiều thời gian) được gọi thì sẽ được cho vào ngăn xếp **Micro-Task**
 - Việc gọi phải thêm từ khóa **await** trước lời gọi hàm
- Javascript cũng cung cấp các hàm async có sẵn của nó



5.3. Asynchronous và await (2)

Có hai loại hàm bất đồng bộ:

1. Được Javascript cung cấp sẵn

- Chẳng hạn: **addEventListener('click')**. Hàm này khi được thực thi sẽ nằm trong hàng đợi task.

2. Cần được khai báo bất đồng bộ (do người dùng tự định nghĩa)

- Chẳng hạn: **fetch()**. Hàm này có thể thực thi đồng bộ, nhưng tốt hơn hết là nên khai báo nó là bất đồng bộ, nhất là khi các mã nguồn bên trong cần nhiều thời gian

5.3. Asynchronous và await (3)

Một phiên bản đồng bộ của hàm `fetch()` có thể như sau:

```
// THIS CODE DOESN'T WORK
```

```
const response = fetch('albums.json');  
const json = response.json();  
console.log(json);
```

Mã nguồn trên rất đơn giản, dễ viết, dễ đọc!!

Tuy vậy hàm đồng bộ `fetch()` sẽ làm chậm hoạt động của trình duyệt khi tải tài nguyên về, làm giảm trải nghiệm của người dùng.

5.3. Asynchronous và await (4)

- Bằng cách thêm các từ khóa khai báo `async` và `await`, kết quả sẽ có sự khác biệt

// But this code does work:

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

5.3. Asynchronous và await (5)

- Một hàm được khai báo là bất đồng bộ có các đặc tính sau:
 - Nó sẽ hoạt động giống như một hàm bình thường nếu bạn không đặt khai báo await trước lời gọi hàm.
- Biểu thức await có dạng:
 - await **Promise**

5.3. Asynchronous và await (6)

- Một hàm được khai báo là bất đồng bộ có các đặc tính sau:
 - Nếu có biểu thức *await*, việc thực thi hàm sẽ tạm dừng cho đến khi thực thi xong ***Promise*** trong biểu thức *await*.
 - Lưu ý: Trình duyệt không bị chặn; nó sẽ tiếp tục thực thi JavaScript khi hàm bất đồng bộ bị tạm dừng.
- Sau đó, khi ***Promise*** được hoàn thành, việc thực thi ở hàm chứa *await* ***Promise*** vẫn tiếp tục.
- Biểu thức *await* trả về giá trị được lưu trữ trong ***Promise***.

```
function onJsonReady(json) {  
  console.log(json);  
}  
function onResponse(response) {  
  return response.json();  
}  
fetch('albums.json')  
  .then(onResponse)  
  .then(onJsonReady);
```

Các phương thức có
màu tím sẽ trả về các
Promise.

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
}  
loadJson();
```

5.3. Asynchronous và await (7)

```
    async function loadJson() {  
        ➡ const response = await fetch('albums.json');  
        const json = await response.json();  
        console.log(json);  
➡ }  
loadJson();
```

Khi thực thi đến câu lệnh chứa `await`, có hai điều xảy ra:

1. `fetch('albums.json');` được thực thi
2. Quá trình thực thi `loadJson` sẽ được tạm dừng cho đến khi `fetch('albums.json');` hoàn thành.

5.3. Asynchronous và await (7)

```
async function loadJson() {  
    ➡ const response = await fetch('albums.json');  
    const json = await response.json();  
    console.log(json);  
}  
loadJson();  
console.log('after loadJson');
```

Trong chờ đợi hàm loadJson hoàn thành, nếu một nút được nhấn vào và có trình xử lý sự kiện được gọi, JS sẽ thực thi trình xử lý sự kiện này.

Cách xử lý bất đồng bộ với **.then()**

```
function onResponse(response) {  
    return response.json();  
}  
fetch('albums.json')  
    .then(onResponse)
```

Khi `fetch()` hoàn thành, nó sẽ gọi `onResponse`, với tham số là biến `response`. Giá trị trả về của `fetch()` là một Promise được **lưu trữ** trong đối tượng `response`.

Cách xử lý bất đồng bộ với **.then()**

```
function onJsonReady(jsObj) {  
    console.log(jsObj);  
}  
function onResponse(response) {  
    return response.json();  
}  
fetch('albums.json')  
    .then(onResponse)  
    .then(onJsonReady);
```

- Giá trị trả về của `json()` là một Promise được **lưu trữ** trong đối tượng **`jsObj`**.

Khi hàm `json()` hoàn thành, nó sẽ gọi hàm `onJsonReady` như một callback, với tham số là **`jsObj`**.

Trả về của hàm async

CÂU HỎI: Chuyện gì sẽ xảy ra nếu ta trả về giá trị ở hàm async?

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
  return true;  
}  
loadJson();
```

Trả về của hàm async

Trả lời: hàm async sẽ luôn được trả về một Promise

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
  return true;  
}  
loadJson();
```

Đơn giản là **JavaScript engine** sẽ tạo ra đối tượng của lớp **Promise** mà nó bao bọc (wrap) giá trị trả về

Trả về của hàm async

```
function loadJsonDone(value) {  
  console.log('loadJson complete!');  
  // Prints "value: true"  
  console.log('value: ' + value);  
}
```

```
async function loadJson() {  
  const response = await fetch('albums.json');  
  const json = await response.json();  
  console.log(json);  
  return true;  
}  
loadJson().then(loadJsonDone)  
console.log('after loadJson');
```

Có thể chạy thử để
kiểm tra kết quả

Khi nào thì hàm được khai báo `async`

- Phương thức khởi dựng không được khai báo `async`
- Ta có thể truyền các hàm `async` làm tham số cho các hàm:
 - `addEventListener` (Browser)
 - `on` (NodeJS)
 - `get/put/delete/v.v..` (ExpressJS)
 - Bất kỳ hàm nào mà nhận hàm khác làm tham số