

เอกสารแจกฟรี

เสียต่ายไม่ได้อ่าน

จาวาสคริปต์ฝั่งเซิร์ฟเวอร์

Node.js

ฉบับย่อ

เล่ม 2

โดย แอดมินโฮ ोन้อยออก

ประวัติการแก้ไข

ครั้งที่	วันที่	รายละเอียดการแก้ไข
1	9 ม.ค. 2559	เริ่มสร้าง และเผยแพร่ผลงาน
2	10 ม.ค. 2559	เพิ่มเรื่อง MySQL เข้าไป

ถ้าท่านดาวน์โหลดทิ้งไว้นาน แล้วเพิ่งมาเปิดอ่าน ก็ขอรบกวนให้โหลดใหม่อีกครั้งที่

http://www.patanasongsivilai.com/itebook_form.html

เพื่อผมอัปเดตแก้ไข pdf ตัวใหม่เข้าไป หรือใครไปดาวน์โหลดมาจากที่อื่น

ก็อาจพลาดเวอร์ชันใหม่ล่าสุดได้ครับ

และรบกวนช่วย**กรอกแบบสอบถาม** ตามลิงค์ข้างบนด้วยนะครับ



แอดมินโฮ ोन้อยอก
(จตุรพัชร พัฒนทรงศิริไธ)

9 มกราคม 2559

ถ้าสนใจเกี่ยวกับเพจด้านไอที ก็ติดตามได้ที่ <https://www.facebook.com/programmerthai/>

*EBook เล่มนี้สงวนลิขสิทธิ์ตามกฎหมาย ห้ามมิให้ผู้ใด นำไปเผยแพร่ต่อสาธารณะ เพื่อประโยชน์ในการค้า หรืออื่นๆ โดย
ไม่ได้รับความยินยอมเป็นลายลักษณ์อักษรจากผู้เขียน ผู้ใดละเมิด จะถูกดำเนินคดีตามกฎหมายสูงสุด*

ก่อนจะอ่านหนังสือเล่มนี้

สำหรับใครที่เพิ่งเปิดเล่มนี้ขึ้นมาอ่าน แล้วยังไม่รู้ว่ Node.js คืออะไร มันใช้ทำอะไร รวมทั้งทฤษฎีโน้มนั้นๆ ผมได้อธิบายไว้แล้ว ในหนังสือที่ผมแต่ง 2 เล่ม ได้แก่

- 1) “เสียดายไม่ได้อ่าน จาวาสคริปต์ฝั่งเซิร์ฟเวอร์ (Node.js ฉบับย่อ) เล่ม 1”
- 2) และควรจะอ่านอีกเล่มเสริมคือ “วิธีติดตั้ง Node.js และ npm เบื้องต้น”

(ลิงค์ดาวน์โหลด http://www.patanasongsivilai.com/itebook_form.html)

เกริ่นนำ

หนังสือเล่มนี้จะอธิบายเกี่ยวกับ Node.js โดยจะตะล่อนทัวร์พาทำเวิร์คช็อป (work shop) ง่าย ๆ เพื่อให้เห็นภาพรวมของการใช้งานภาคปฏิบัติ มากกว่าภาคทฤษฎี ซึ่งเนื้อหาคงไม่ได้เจาะลึกรายละเอียดขั้นเทพมากมาย

แต่ก็หวังว่ามันจะช่วยประหยัดเวลาในการเรียนรู้ของคุณ และสามารถช่วยคุณนำไปต่อยอดงานในอนาคตได้

สุดท้ายนี้หากเนื้อหามีอะไรผิดพลาดไป เช่น ให้ข้อมูลผิด สะกดอะไรผิดไป มุมแป้กบ้าง ขำบ้าง หรืออ่านแล้วมึนงไป 7 วัน เป็นต้น ผมก็ขออภัยมา ณ โอกาสนี้ด้วย และถ้าคุณเข้าใจ ไม่เข้าใจยังไง ก็สามารถชี้แนะผมได้ตลอดเวลา

...อีกทั้งผมก็ตั้งใจจะหมั่นอัปเดตเนื้อหา ขึ้นอยู่กับเวลา โอกาส และความสามารถจะอำนวย

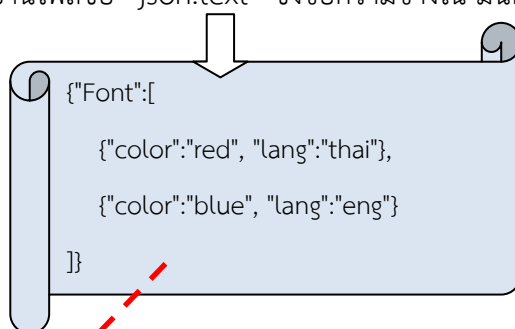
การอ่านและเขียนไฟล์

ในบทนี้จะกล่าวถึงมอดูล fs ซึ่งมีไว้อ่านและเขียนไฟล์ โดยผมจะแนะนำการใช้งานเบื้องต้นเท่านั้น ซึ่งถ้าใครเคยอ่านเล่มแรกของผม ก็จะได้เห็นวิธีใช้งานผ่านตามาแล้ว ด้วยประโยคคำสั่ง

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

วิธีอ่านไฟล์

ในตัวอย่างนี้ ผมจะใช้มอดูล fs อ่านไฟล์ชื่อ "json.text" ซึ่งข้อความข้างใน มันคือชนิดข้อมูลแบบ JSON



โครงสร้างโปรเจกต์จะเป็นดังนี้

C:\testfile

|-- files

|-- json.txt

|--readfile.js

ภายในไฟล์จะเขียนโค้ดดังนี้

```
var fs = require('fs'); // โหลดมอดูล fs
fs.readFile('./files/json.txt', 'utf8', function(err, buffer) {
  if (err) {
    console.log(err); // แสดงข้อความผิดพลาดออกมา
  }
  console.log(buffer.toString()); // แสดงข้อความที่อยู่ในไฟล์ json.txt
});
console.log('Read a file...');
```

อ็อบเจกต์ที่เรียกว่า Buffer

เมื่อพิมพ์คำสั่ง “node readfile.js” ก็จะได้ผลลัพธ์ดังนี้

```
C:\testfile>node readfile.js

Read a file...

{"Font":[
  {"color":"red", "lang":"thai"},
  {"color":"blue", "lang":"eng"}
]}
```

เมื่อย้อนกลับไปโค้ด ค่าอาร์กิวเมนต์ตัวที่สองของ `fs.readFile(..., 'utf8', ...)` จะเป็นการบอกค่า encoding หรือการเข้ารหัสตัวอักษร (ในตัวอย่างนี้คือ 'utf8') ซึ่งเราจะระบุไว้ หรือไม่ระบุไว้เลยก็ได้ ดังโค้ดตัวอย่าง

```
fs.readFile('./files/json.txt', function(err, buffer){ /*...*/ })
```

*** แต่ถ้าไม่ระบุไว้ ...โดยดีฟอลต์แล้ว ค่า encoding จะเป็น null

ถ้าไฟล์ json.txt อยู่ที่ไดเรกทอรีเดียวกันกับ readfile.js ก็ไม่ต้องระบุชื่อพาร ดังโค้ดตัวอย่าง

```
fs.readFile('json.txt', function(err, buffer){ /*...*/ })
```

วิธีเขียนไฟล์

ผมจะสร้างไฟล์ “writefile.js” ในโปรเจกต์เดิม ...เพื่อเอาไว้เขียนข้อความ “I am a programmer” ลงไปในไฟล์ “message.txt” โดยจะมีโครงสร้างโปรเจกต์ดังนี้

```
C:\testfile
```

```
|-- files
```

```
|-- json.txt
```

```
|--readfile.js
```

```
|-- writefile.js
```

ไม่ได้ใช้งานสำหรับตัวอย่างนี้



จะเห็นโค้ดในหน้าถัดไป

```

var fs = require('fs'); // โหลดมอดูล fs
fs.writeFile(
  './files/message.txt',
  'I am a programmer',
  'utf8',
  function (err) {
    if (err) {
      console.log(err); // แสดงข้อความผิดพลาดออกมา
    }
    console.log('It's saved!');
  } // สิ้นสุดฟังก์ชัน
);
console.log('End writing the file');

```

ชื่อไฟล์ที่จะเขียนข้อความลงไป

ข้อความที่จะเขียนลงไป หรือจะใช้เป็นอ็อบเจกต์ Buffer ก็ได้

เข้ารหัส (ระบุหรือไม่ก็ได้)

ในตัวอย่างนี้ เมื่อพิมพ์คำสั่ง “node writefile.js” ก็จะได้ผลลัพธ์ดังนี้

```

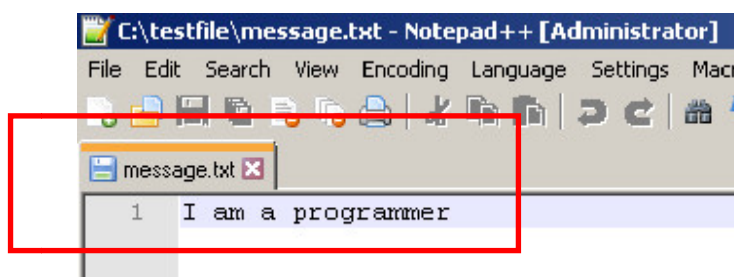
C:\testfile>node writefile.js

End writing the file

It's saved!

```

ถ้าไปเปิดโฟลเดอร์ C:\testfile\files\ ควรจะเห็นชื่อไฟล์ “message.txt” ถูกสร้างขึ้นมา และเมื่อเปิดอ่านก็จะมีข้อความ “I am a programmer” ดังนี้ครับ



เมื่อย้อนกลับไปโค้ดเดิม ในเมธอด `fs.writeFile()` จะมีรายละเอียดของค่าอาร์กิวเมนต์ที่น่าสนใจดังนี้

- ค่าอาร์กิวเมนต์ตัวแรกจะบอกชื่อไฟล์ `./files/message.txt` ซึ่งมันจะถูกสร้างขึ้นใหม่ทุกครั้ง เมื่อเรียกเมธอด `fs.writeFile()` ให้ทำงาน ...แต่ถ้าระบุแค่ชื่อเป็น “message.txt” (ไม่มีพาร) ไฟล์ก็就会被สร้างขึ้นในไดเรกทอรีเดียวกันกับ `writfile.js`
- ค่าอาร์กิวเมนต์ตัวที่ 2 จะเป็นข้อความที่เขียนลงไป ซึ่งในที่นี้คือ “I am a programmer” แต่ขณะเดียวกัน เราก็สามารถระบุเป็นอ็อบเจกต์ Buffer ได้ด้วย
- ค่าอาร์กิวเมนต์ตัวที่ 3 จะบอกค่า encoding เป็น 'utf8' แต่ถ้าไม่ได้ระบุค่านี้ โดยดีฟอลต์แล้วจะเป็น 'utf8'

หมายเหตุ

สำหรับรายละเอียดของ `fs.readFile()` กับ `fs.writeFile()` มันยังมีต่ออีกนะครับ ลองอ่านเพิ่มเติมได้ที่ <https://nodejs.org/api/fs.html>

ไม่เพียงเท่านั้น ในมอดูล `fs` ก็ยังมีเมธอดอื่น ๆ อีกเยอะมาก แต่ผมขอตัดจบก่อน เพราะคิดว่าท่านน่าจะเข้าใจหลักการทำงานไม่ยากแหละ

การใช้งาน Buffer

Buffer ใน Node.js ผมขอให้คำจำกัดความว่า

“Buffer เป็นอ็อบเจกต์ที่เก็บสตริง ซึ่งเราสามารถจัดการมันได้ในระดับไบต์”

สำหรับวิธีสร้าง Buffer ก็สามารถสร้างได้ง่ายจากสตริง ดังตัวอย่าง

```
1. var buf = new Buffer('Hello World');
2. console.log(buf.length);           // 11
3. console.log(buf.toString());       // "Hello World"
```

ในตัวอย่างจะมีรายละเอียดที่น่าสนใจดังนี้

- **บรรทัด 1:** Buffer() จะรับค่าเข้ามาเป็นสตริง ที่เข้ารหัสเป็น UTF-8 (โดยดีฟอลต์) หลังจากนั้นจะใช้โอเปอเรเตอร์ new สร้างอ็อบเจกต์บัฟเฟอร์ขึ้นมา โดยมีตัวแปร buf มาอ้างอิงอีกที
- **บรรทัด 2:** บัฟเฟอร์จะมีพรีอเพอร์ติวชื่อ length (เหมือนที่มีในสตริง) เอาไว้บอกจำนวนตัวอักษร
- **บรรทัด 3:** บัฟเฟอร์จะมีเมธอด toString() เพื่อเอาไว้ใช้แปลงอ็อบเจกต์ให้กลายเป็นสตริงภายหลัง

แต่ทว่า Buffer ก็สามารถระบุค่าอากิวเมนต์ตัวที่สอง เพื่อบอกค่า Encoding ดังตัวอย่าง

```
var buf = new Buffer('Hello World', 'ascii');
console.log(buf.toString()); // "Hello World"
```

สำหรับ Encoding จะมีให้เลือกระบุ ได้แก่ ascii, utf8, utf16le, ucs2, base64, binary และ hex

นอกจากนี้แล้วเมธอด toString() ก็สามารถระบุค่า Encoding ได้เช่นเดียวกัน ดังตัวอย่าง

```
var buf = new Buffer('Hello World', 'ascii');
console.log(buf.toString('ascii')); // "Hello World"
```


ส่วนวิธีการเข้าถึงค่าในบัฟเฟอร์ ก็จะเหมือนเวลาคุณใช้งานอาร์เรย์ ดังตัวอย่าง

```
var buf = new Buffer('Hello World');
console.log(buf[10]); // แสดงตัวเลข 100 ออกมา (ค่า ASCII คือ d ตัวพิมพ์เล็ก)
buf[10] = 90;        // แก้ไขตัวอักษรที่อินเด็กซ์ 10 ให้มีค่าเป็น 90 (ค่า ASCII คือ Z ตัวพิมพ์ใหญ่)
console.log(buf.toString()); // แสดงผล "Hello WorlZ"
```

ในตัวอย่างนี้ตัวแปร buf คือบัฟเฟอร์ ที่เก็บค่าสตริงเป็น "Hello World" โดยเราสามารถเข้าถึงตัวอักษรที่อยู่ในสตริง ด้วยการระบุตำแหน่งอินเด็กซ์ภายในวงเล็บเหลี่ยม (buf[index]) เหมือนเป็นอาร์เรย์ธรรมดาตัวหนึ่ง

อย่างที่บอกในตอนต้น บัฟเฟอร์สามารถจัดการในระดับไบต์ได้ ดังนั้นตอนสร้างบัฟเฟอร์ เราจึงสามารถระบุขนาดให้มีหน่วยเป็นไบต์ได้ แต่ค่าเริ่มต้นของสมาชิกในบัฟเฟอร์ มันจะมีค่าแบบสุ่ม (Random) ดังตัวอย่าง

```
var buf = new Buffer(10); // สร้างบัฟเฟอร์ขนาด 10 ไบต์

// ค่าสมาชิกภายในบัฟเฟอร์ จะถูกกำหนดค่าเริ่มต้นเป็นแบบสุ่ม
console.log(buf);
// แสดงผลลัพธ์ <Buffer 03 00 00 00 04 00 00 00 10 fe>
```

ในตัวอย่างถ้าลองใช้ console.log(buf) แสดงค่าสมาชิกภายในบัฟเฟอร์ออกมา ก็ให้เห็นมันเก็บค่าเริ่มต้นเป็นแบบสุ่ม

บัฟเฟอร์มันยังมีเมธอด slice() เอาไว้ตัดแบ่งสตริงให้เล็กลง ดังตัวอย่าง

```
var buffer = new Buffer('Hello World');
var smaller = buffer.slice(6, 11);
console.log(smaller.toString()); // "World"
console.log(smaller === buffer); // false
```

ในตัวอย่างนี้ประโยค buffer.slice(6, 11); จะตัดแบ่งสตริงตัวเดิม จากอินเด็กซ์ตำแหน่งที่ 6 ไปจนถึงตำแหน่งที่ 11 (ตัวสุดท้าย) แล้วได้เป็นบัฟเฟอร์ตัวใหม่ที่เก็บค่าสตริงเป็น "World"

นอกจากนี้แล้วบัฟเฟอร์ยังมีเมธอด `copy()` เอาไว้ก๊อปปี้ค่าสตริงจากบัฟเฟอร์ตัวหนึ่ง ไปยังบัฟเฟอร์อีกตัว ดังตัวอย่าง

```
var source = new Buffer('Hello World');
var target = new Buffer(6);
var targetStart = 0;
var sourceStart = 6;
var sourceEnd = 11;
source.copy(target, targetStart, sourceStart, sourceEnd);
console.log(target.toString());           // "World"
console.log(target === source);           // false
```

ในตัวอย่างนี้จะก๊อปปี้ค่าสตริงจากตัวแปร `source` โดยเริ่มจากอินเด็กซ์ตำแหน่งที่ 6 จนถึง 11 แล้วนำค่าสตริงที่ได้มาใส่ไว้ในตัวแปร `target` ตั้งแต่อินเด็กซ์ 0

...สำหรับผลการทำงานของตัวอย่างนี้ จะเหมือนกับการใช้เมธอด `slice()` ในตัวอย่างก่อนหน้านี้ทุกประการ

หมายเหตุ

สำหรับรายละเอียดของ Buffer มากกว่านี้ สามารถดูเพิ่มเติมได้ที่

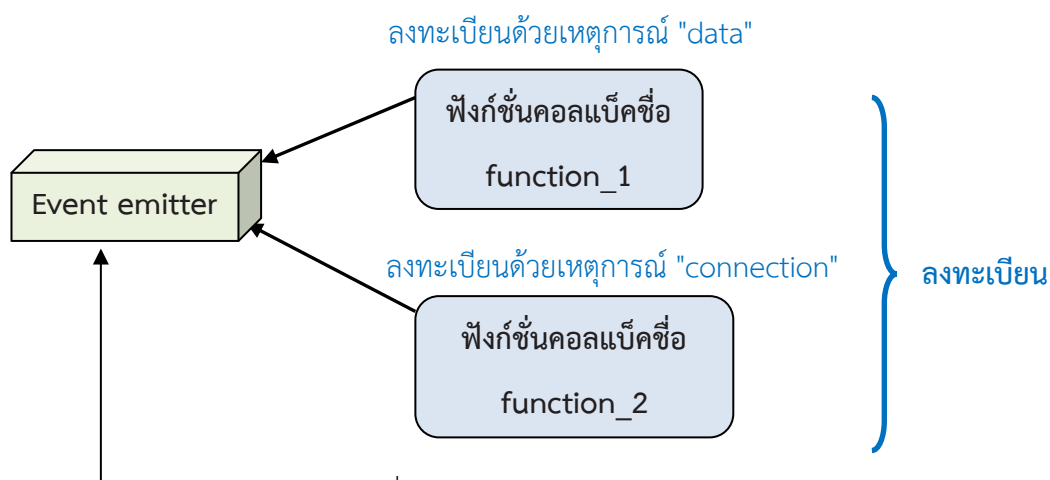
https://nodejs.org/api/buffer.html#buffer_new_buffer_array

แนวคิด EMITTER PATTERN

อย่างที่ท่านทราบ คอลแบ็คใน Node.js จะถูกเรียกเมื่อเกิดเหตุการณ์เกิดขึ้นมา

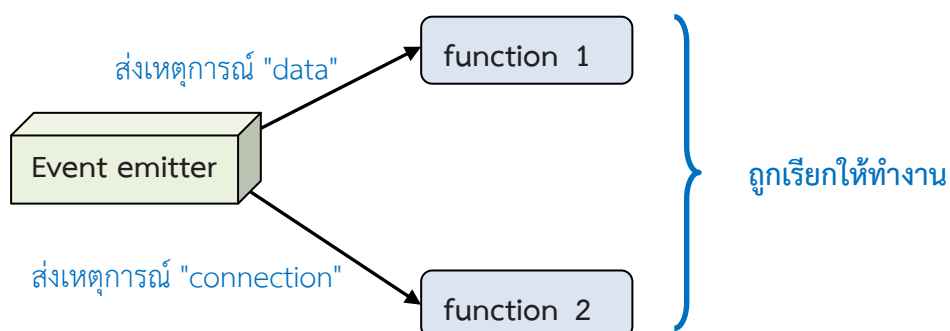
แต่เราก็สามารถสร้างเหตุการณ์ได้ด้วยตัวเอง ด้วยรูปแบบการเขียนโปรแกรมที่เรียกว่า **Emitter pattern** ซึ่งมันคล้าย ๆ กับดีไซน์แพทเทิร์น (Design pattern) ที่เรียกว่า “observer pattern”

...เอานะ ถ้าใครไม่รู้จักดีไซน์แพทเทิร์นดังกล่าว เดี่ยวผมจะอธิบายด้วยรูปภาพต่อไปนี้แล้วกัน



จากรูป Event emitter ก็คืออ็อบเจกต์ ที่มีคอลแบ็คมาลงทะเบียนได้แก่ function_1 กับ function_2 ซึ่งพวกมันจะทำหน้าที่เป็น Listener หรือผู้ฟังเหตุการณ์ที่เกิดขึ้น ได้แก่ "data" กับ "connection"

...สำหรับเหตุการณ์ "data" กับ "connection" เราสามารถตั้งชื่อเป็นสตริงอะไรก็ได้ แล้วเมื่อเกิดเหตุการณ์ดังกล่าวขึ้นมา ตัวคอลแบ็คก็就会被เรียกให้ทำงาน ดังภาพข้างล่าง



ในภาพเมื่อ Event emitter ส่งเหตุการณ์ "data" กับ "connection" ตัวคอลแบ็คได้แก่ function_1 กับ function_2 ก็จะถูกเรียกให้ทำงานตามลำดับ

เพื่อไม่ให้เป็นการเสียเวลา ดูตัวอย่างโค้ดกันดีกว่า ...มันง่ายมากเลย

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

var EventEmitter = new events.EventEmitter();// สร้างอ็อบเจกต์ Event emitter

eventEmitter.on('data', function(){           // ลงทะเบียนคอลแบ็ค ด้วยเหตุการณ์ "data"
  console.log('Listen data');
});

eventEmitter.on('connection', function(){      // ลงทะเบียนคอลแบ็ค ด้วยเหตุการณ์ "connection"
  console.log('Listen connection');
});

eventEmitter.emit('data');                     // ส่งเหตุการณ์ "data"
eventEmitter.emit('connection');               // ส่งเหตุการณ์ "connection"
```

โค้ดในตัวอย่างนี้ มันจะแสดงผลลัพธ์เป็น

Listen data

Listen connection

ในโค้ดจะมีรายละเอียดที่น่าสนใจดังนี้

- เมธอด eventEmitter.on() คือการลงทะเบียนคอลแบ็ค พร้อมทั้งระบุเหตุการณ์ที่จะรับฟัง
- ส่วน eventEmitter.emit() คือการส่งเหตุการณ์ เพื่อเรียกคอลแบ็ค (ที่ลงทะเบียนไว้) ให้ทำงาน

นอกจากนี้แล้ว Event emitter มันยังมีเมธอดอื่นอีก นอกจาก on() กับ emit() ตัวอย่างเช่น

เมธอด	คำอธิบาย
addListener	ใช้งานเหมือนกับเมธอด on
removeEventListener	ลบคอลแบ็คที่ได้ลงทะเบียนไว้
removeAllEventListeners	ลบคอลแบ็คทั้งหมดที่ได้ลงทะเบียนไว้

นอกจากนี้แล้วเราสามารถสร้างคลาสขึ้นมา แล้วสืบทอดมาจาก Event emitter ด้วยเมธอด util.inherits() ดังตัวอย่างโค้ดต่อไปนี้

```
var util = require('util');
var events = require('events');

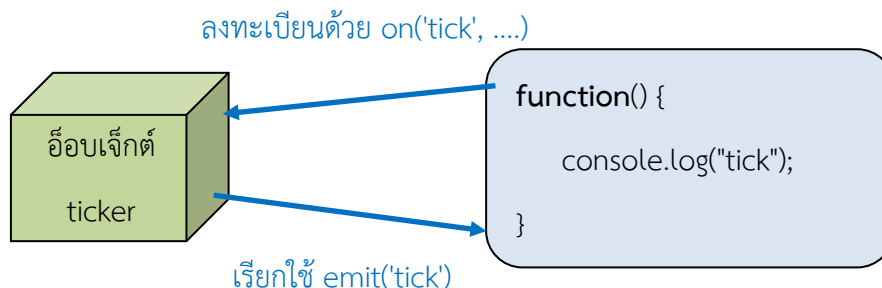
class Ticker{                                // ประกาศคลาสใน ES6
    constructor(){                          // ประกาศคอนสตรัคเตอร์ใน ES6
        let handler = () => this.emit('tick'); // ฟังก์ชันลูกศรที่ชื่อ handler

        // เมื่ออ็อบเจ็กต์ถูกสร้างขึ้นมาจากคลาส Ticker
        // ก็จะทำงานแบบอะซิงโครนัส ด้วยการดีเลย์เรียกฟังก์ชัน handler ทุก ๆ 1 วินาที
        setInterval(handler, 1000);        // 1000 milliseconds
    }
}

// คลาส Ticker สืบทอดมาจาก events.EventEmitter
util.inherits(Ticker, events.EventEmitter);

let ticker = new Ticker();                  // สร้างอ็อบเจ็กต์ด้วยโอเปอเรเตอร์ new
ticker.on('tick', function() {             // ลงทะเบียนคอลแบ็กด้วยเหตุการณ์ 'tick'
    console.log("tick tock");
});
```

ในตัวอย่างนี้ผมเขียนโค้ดด้วย ES6 (เขียน ES5 แบบเดิมก็ได้ แต่ยุ่งยากมาก) ซึ่งการทำงานก็จะได้ตามภาพ



ในภาพก่อนหน้านี้ จะมีรายละเอียดที่น่าสนใจดังนี้

- อ็อบเจกต์ ticker ถูกสร้างมาจากคลาส Ticker ซึ่งสืบทอดมาจาก events.EventEmitter อีกที
- ส่วน ticker ก็ได้ลงทะเบียนคอลแบ็ก เพื่อรับฟังเหตุการณ์เป็น "tick"
- เมื่อ ticker ถูกสร้างขึ้นมา ตัว constructor ของคลาส Ticker ก็ทำงานทันที ด้วยการดีเลย์ หรือนอนหลับไปพักใหญ่ แล้วตื่นขึ้นมาเรียกใช้ this.emit('tick') ทุก ๆ 1 วินาที

เนื่องจากโค้ดนี้ผมใช้ ES6 จึงต้องพิมพ์คำสั่งเป็น “node --use-strict example.js” ซึ่งผลลัพธ์จะแสดงคำว่า “tick tock” ทุก ๆ 1 วินาที ดังนี้

tick tock

tick tock

tick tock

แสดงคำว่า “tick tock” ทุก ๆ 1 วินาที

หมายเหตุ

สำหรับรายละเอียดของ Event Emitter มากกว่า ดูเพิ่มเติมได้ที่

<https://nodejs.org/api/events.html>

บทที่ 4 เว็บแอปพลิเคชันง่าย ๆ

ในบทนี้ ผมจะพาคุณท่องเที่ยวทัวร์ ไปดูวิธีสร้างเว็บแอปพลิเคชันเบื้องต้นด้วย Node.js โดยจะอธิบายเป็นแนวเวิร์คช็อปง่าย ๆ ...เพราะถ้าไม่พาคุณมาดูวิธีสร้างเว็บแอปด้วย Node.js ก็เหมือนเวลาคุณไปเยือนถิ่นอีสาน ไม่กินส้มตำก็แปลก (เกี่ยวกันปะ)

เตรียมโปรเจกต์ให้พร้อมก่อน

ผมจะสร้างโฟลเดอร์ webapp เอาไว้เป็นที่อยู่ของโปรเจกต์ และ cd เข้าไปในไดเรกทอรีดังกล่าว ด้วยคำสั่ง

```
C:\mkdir webapp
```

```
C:\cd webapp
```

ให้พิมพ์คำสั่ง npm ดังต่อไปนี้

```
npm init -y
```

คำสั่งนี้จะสร้างไฟล์ “package.json” ซึ่งจะใช้อธิบายโปรเจกต์ของเรา (metadata) เช่น โปรเจกต์ชื่ออะไร เวอร์ชันอะไร มีแพ็คเกจอะไรติดตั้งลงบ้าง เป็นต้น ไฟล์นี้จะถูกสร้างขึ้นมา แล้วอยู่ในโฟลเดอร์ webapp ดังรูป



The screenshot shows a terminal window with the command `C:\webapp>npm init -y` and the output `Wrote to C:\webapp\package.json:`. Below this, the JSON content of the file is displayed. A callout box points to the JSON content with the text “ข้อความภายในไฟล์ package.json”.

```
<
{
  "name": "webapp",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC"
}
```

หมายเหตุ จริง ๆ แล้ว เราสามารถตัดขั้นตอนการสร้าง package.json ไปเลยก็ได้ สามารถข้ามไปยังหัวข้อถัดไปได้ แต่ที่ผมแนะนำให้สร้างไว้ ก็เพื่อใช้บริหารจัดการโปรเจกต์ในอนาคตได้ง่ายขึ้น (ลองอ่านหนังสือที่ผมเขียนนะครับ “วิธีติดตั้ง Node.js และ npm เบื้องต้น”)

ดาวน์โหลดและติดตั้ง Express

ขั้นตอนต่อไปนี้สำคัญมาก ผมจะต้องดาวน์โหลดและติดตั้งแพ็คเกจ Express เข้ามาเก็บไว้ในโฟลเดอร์ webapp โดยจะให้พิมพ์คำสั่งดังต่อไปนี้

แฟล็ก -- save มันจะอัปเดตไฟล์ package.json ไปด้วยในตัว

```
C:\webapp>npm install express --save
```

ซึ่งถ้าผมแอบเข้าไปเปิดไฟล์ package.json ขึ้นมาดู ก็จะเห็นว่าหน้าตาเปลี่ยนแปลงไป ดังตัวอย่าง



คราวนี้ผมจะลองสำรวจดูโฟลเดอร์ ซึ่งควรจะเห็นโครงสร้างดังนี้

```
C:\webapp
```

```
|-- node_modules\...
```

```
|-- package.json
```

แพ็คเกจต่าง ๆ ที่ดาวน์โหลดมาด้วยคำสั่ง npm มันจะเก็บอยู่ในโฟลเดอร์นี้

คิดว่าเมื่อถึงตอนนี้ คุณอาจสงสัยว่า Express มันคืออะไร ...มันใช่ *อเมริกัน แฮ็กเพรส* หรือไม่หนอ? และมันทำหน้าที่อะไร?

คำตอบ Express มันเป็นเฟรมเวิร์ค (Framework) เอาไว้สร้างเว็บแอปพลิเคชันบน Node.js

อีกอย่าง ...ตอนที่ผมเขียนหนังสือ Express เวอร์ชัน 5 กำลังจะออกมา (ยังเป็นเวอร์ชัน alpha) แต่ในตอนนี้จะใช้ Express 4 ไปพลาง ๆ ก่อนแล้วกัน ซึ่งในอนาคตผมอาจมาปรับปรุงเนื้อหาตามภายหลังได้นะครับ

มาเขียนไฟล์จาวาสคริปต์ เพื่อสร้างเว็บแอปกันเถอะ

ผมจะโหลดมอดูลที่ชื่อ “express” เข้ามา ด้วยประโยคคำสั่งดังตัวอย่าง

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

เราสามารถใช้อ็อบเจกต์ app มารับรีเควสต์ (Request) จากเว็บเบราว์เซอร์เป็นแบบ GET ด้วยโค้ดดังตัวอย่าง

```
app.get('/', function(req, res) {
    res.send('<h1>Hello world </h1>'); // ส่งเรสปอนส์ (Response) ตอบกลับออกไป
});
```

ในตัวอย่างจะเห็นว่า เมธอด app.get() จะรับค่าอาร์กิวเมนต์ตัวแรกคือ path (URL) ส่วนตัวที่สองจะเป็นคอลแบ็ก ซึ่งมีพารามิเตอร์ 2 ตัว ได้แก่

- ตัวแปร req จะรับค่าเป็นอ็อบเจกต์ (ตัวแทนรีเควสต์)
- ตัวแปร res จะรับค่าเป็นอ็อบเจกต์ ซึ่งมีเมธอด send() เอาไว้ส่งเรสปอนส์ตอบกลับไปยังหน้าเว็บเบราว์เซอร์ที่เรียกเข้ามา
- จริง ๆ แล้วคอลแบ็กดังกล่าว มันยังมีพารามิเตอร์ตัวสาม ซึ่งรับค่าเข้ามาเป็นฟังก์ชันคอลแบ็กอีกที เพื่อเรียกต่อไปยัง middleware ตัวถัดไปให้ทำงาน (เดี่ยวจะได้เห็นในบทถัด ๆ ไป)

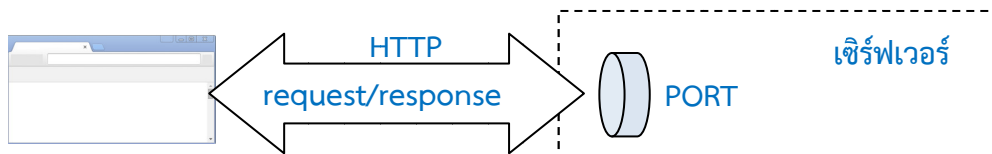
***นอกจากนี้แล้ว app ยังมีเมธอดอื่นอีก ตัวอย่างเช่น app.post(), app.put() และอื่น ๆ อีก ...ซึ่งชื่อเมธอดจะตรงกับชื่อ METHOD ของโปรโตคอล HTTP อีกด้วยนะครับ

ขอเพิ่มเติมเล็กน้อย สำหรับอ็อบเจกต์ res ในตัวอย่างเดิม มันยังมีเมธอดที่น่าสนใจดังนี้

เมธอด	หน้าที่
res.download()	ให้ไคลเอนต์ดาวน์โหลดไฟล์ไปใช้งาน
res.end()	สิ้นสุดการส่งเรสปอนส์
res.json()	ส่งเรสปอนส์เป็นแบบ JSON
res.jsonp()	ส่งเรสปอนส์เป็นแบบ JSONP
res.redirect()	เปลี่ยนทิศทางรีเควสต์ที่ติดต่อเข้ามา
res.render()	สั่งให้เทมเพลต (template engine) แสดงผลบนหน้าเว็บ (จะเห็นการใช้งานในบทถัดไป)

เมธอด	หน้าที่
res.send()	ส่งレスポンス
res.sendFile()	ส่งレスポンスเป็นไฟล์ในรูปแบบ octet stream
res.sendStatus()	ส่งรหัสของレスポンス (Response Code) พร้อมทั้งสตริงอธิบายตัวบอัติของレスポンス

ต่อมาเราจะให้อ็อบเจกต์ app เปิดพอร์ต (port) เพื่อให้เว็บเบราว์เซอร์สามารถติดต่อเข้ามาได้ ดังตัวอย่าง



```
app.listen(8080, function() {
```

```
  console.log('Server running at http://localhost:8080/');
```

```
});
```

โค้ดในตัวอย่างนี้เมธอด listen() จะเปิดพอร์ต 8080 (เป็นอะไรก็ได้) ค้างทิ้งไว้ (วนลูการทำงานค้างไว้) ...ถ้าพูดอีกนัยหนึ่ง โค้ดชุดนี้ทำหน้าที่เป็นเซิร์ฟเวอร์ และถ้าเปิดพอร์ตสำเร็จละก็ ตัวคอลแบ็คของเมธอด ก็จะทำหน้าที่ ด้วยการแสดงข้อความออกทางหน้าคอนโซลเป็น 'Server running at http://localhost:8080/'

จากตัวอย่างที่ยกมาทั้งหมด ผมจะนำมาเขียนใส่ไว้ในไฟล์ชื่อ “app.js” ซึ่งโค้ดจะสั้นมาก ๆ ดังตัวอย่าง

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

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

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

```
  res.send('<h1>Hello world</h1>');
```

```
});
```

```
app.listen(8080, function() {
```

```
  console.log('Server running at http://localhost:8080/');
```

```
});
```

```
console.log("Start server");
```

เมื่อเกิดเหตุการณ์ที่เว็บเบราว์เซอร์ส่งรีควีสต์เข้ามา (ด้วย path /) ตัวคอลแบ็คก็就会被เรียกให้ทำงาน

// http://localhost:8080/

วนลูค้างเพื่อเปิดพอร์ตทิ้งไว้

เมื่อเปิดพอร์ตสำเร็จ คอลแบ็คจะถูกเรียกให้ทำงาน

ในตัวอย่างนี้จะมีโครงสร้างโฟลเดอร์ของโปรเจก ในหน้าถัดไป

```
C:\webapp
```

```
|-- node_modules\...
```

```
|-- package.json
```

```
|-- app.js
```

และเมื่อรันคำสั่ง “node app.js” บนคอมพิวเตอร์ออนไลน์ ก็จะแสดงผลดังต่อไปนี้

```
c:\webapp>node app.js
```

```
Start server
```

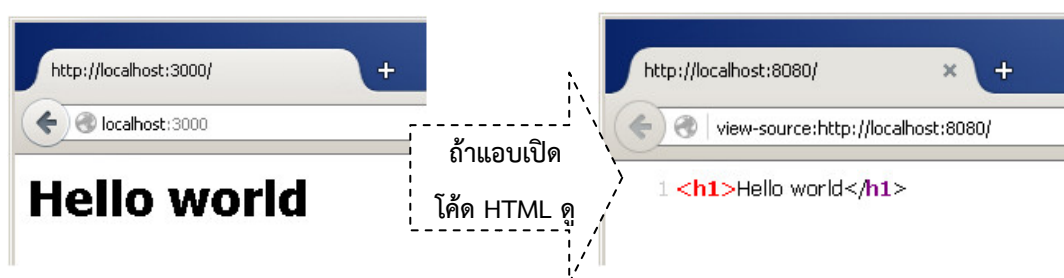
```
Server running at http://localhost:8080/
```

ทำตัวเป็นเซิร์ฟเวอร์

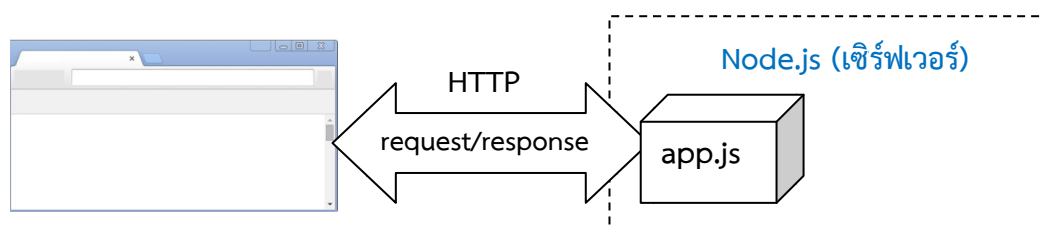
เปิดพอร์ต 8080 ค้างไว้

Node.js ยังไม่จบการทำงาน

หลังจากนั้นจะให้เปิดเว็บเบราว์เซอร์ขึ้นมา แล้วก็ให้กรอก URL เป็น <http://localhost:8080/> ซึ่งจะเห็นหน้าจอ มันแสดงข้อความว่า “Hello world” ดังภาพ

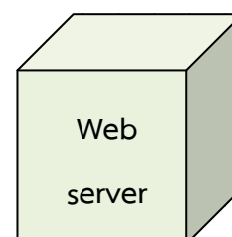


จากตัวอย่างที่ผมทำให้ดู จะเห็นว่าแค่เขียน app.js ไม่กี่บรรทัด คุณก็ได้เว็บแอปพลิเคชันง่าย ๆ แล้วครับ



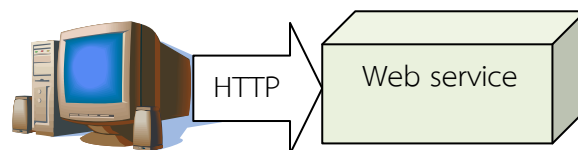
สำหรับตัวอย่างนี้ ยังไม่มีลูกเล่นอะไรที่ซับซ้อน แต่อย่างน้อยคุณคงจะเห็นว่า แค่เราพิมพ์คำสั่งบนคอมพิวเตอร์ออนไลน์เพื่อรันไฟล์จาวาสคริปต์ขนาดเพียงไม่กี่ไบต์ มันก็ทำงานเป็นเซิร์ฟเวอร์ได้แล้ว

แต่ถ้าคุณใช้บางภาษาสคริปต์เช่น PHP, ASP หรือ JSP เป็นต้น ก็จะต้องติดตั้งซอฟต์แวร์ที่เป็นเว็บเซิร์ฟเวอร์ (Web server) ขึ้นมาก่อน ถึงจะรันสคริปต์ได้ แต่ Node.js ไม่ต้องทำ เพราะมันเบาวิว หรือ lightweight กว่านั่นเองครับ



ขอแทรกด้วย RESTful

ในหัวข้อนี้ผมจะพาทำเว็บเซอร์วิส (Web Service) ...ซึ่งชื่อของมันขึ้นต้นด้วยเว็บก็จริง แต่มันไม่ใช่เว็บไซต์นะ



คำว่าเว็บเซอร์วิสในมุมผู้เขียน มันคือเซอร์วิสฝั่งเซิร์ฟเวอร์ที่ให้บริการอะไรสักอย่าง โดยที่ไคลเอนต์เวลาติดต่อกับฝั่งเซิร์ฟเวอร์ ทั้งคู่จะไม่แคร์ว่าอยู่บนฮาร์ดแวร์ หรือ OS อะไร (ไม่สนใจแพลตฟอร์ม) และไม่สนใจว่าทั้งสองฝั่งใช้ภาษาโปรแกรมอะไรในการคุยกัน แต่ทั้งนี้เวลาไคลเอนต์ขอใช้บริการจากเซอร์วิส ข้อมูลจะต้องวิ่งอยู่บนโปรโตคอลของเว็บไซต์คือ HTML ...ด้วยเหตุนี้เราถึงเรียกมันว่า เว็บเซอร์วิส

ถึงอย่างไรก็ตามฝั่งเซิร์ฟเวอร์ผู้ให้บริการ กับฝั่งไคลเอนต์ จะต้องตกลงกันก่อนว่า จะให้คุยกันด้วยรูปแบบไหน (วิ่งอยู่บน HTML นั่นแหละ) ซึ่งในปัจจุบันจะนิยมใช้เป็น SOAP หรือ RESTful (เรียกสั้น ๆ ว่า REST)

แต่ในบทนี้จะกล่าวถึงเว็บเซอร์วิสแบบ REST อย่างเดียว ซึ่งสามารถส่งข้อมูลได้เป็น ข้อความ, JSON และ XML ...และเพื่อไม่ให้เป็นการเสียเวลา ก็จะทำให้ดูตัวอย่างในหัวข้อถัดไปแล้วกันนะ

เตรียมไฟล์ JSON ตัวอย่าง

```
{
  "Font": [
    {
      "color": "red",
      "lang": "thai"
    },
    {
      "color": "blue",
      "lang": "eng"
    }
  ]
}
```

ผมจะบันทึกไฟล์นี้อยู่ในโปรเจกต์เดิมเป็นชื่อ “font.json”

C:\webapp

```
|-- node_modules\...
|-- package.json
|-- app.js
|-- font.json
```

แก้ไขไฟล์ app.js

ในตัวอย่างนี้ ผมได้นำไฟล์ app.js ของตัวอย่างเดิม มาแก้ไขเสียใหม่ดังนี้

```
var express = require('express');
var app = express();
var fs = require("fs");

app.get('/listFont', function(req, res) {    // http://ipaddress:8081/listFont

    fs.readFile( __dirname + "/" + "font.json", function (err, data) {

        if(err){

            console.error('Error read a file:', err.stack);

        }

        res.end( data );

    }); // ลื่นสุดคอลแบ็ค

});

app.listen(8081, function() {

    console.log('Server running at http://localhost:8081/');

});

console.log("Start server");
```

ตัวแปร `__dirname` จะมีขีดล่างสองอันวางติดกัน
ซึ่งจะแทนที่ด้วยชื่อไดเรกทอรี ซึ่งไฟล์ app.js มันอาศัยอยู่

อ่านเนื้อหาในไฟล์ font.json
แล้วส่ง data (Buffer) กลับออกไปหาไคลเอนต์

เปลี่ยนพอร์ตจาก 8080 เป็น 8081 (ตั้งเป็นอะไรก็ได้)

เมื่อนำไฟล์ app.js มารันจะได้ผลลัพธ์ดังนี้

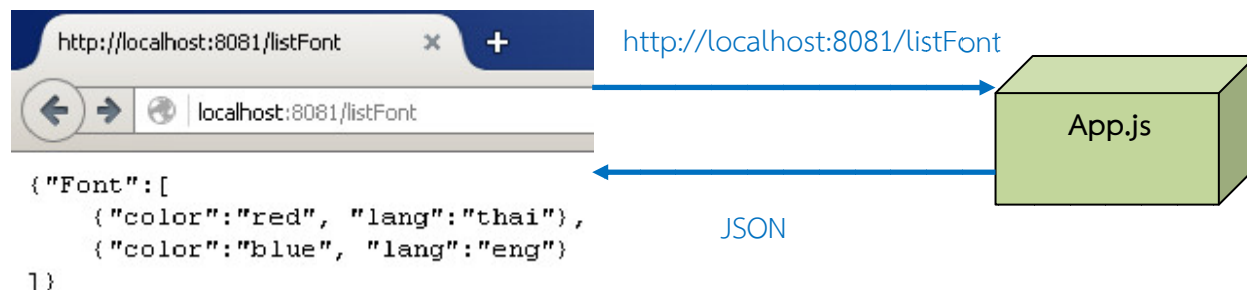
```
C:\code\webapp>node app.js

Start server

Server running at http://localhost:8081/
```

เมื่อไคลเอนต์ติดต่อเข้ามาด้วยโปรโตคอล http (แบบ GET) เป็น <http://localhost:8081/listFont> ...ฝั่งเซิร์ฟเวอร์ก็จะตอบกลับออกไป ด้วยข้อมูล JSON (อ่านมาจากไฟล์ font.json)

...โดยเราสามารถทดสอบการทำงานอย่างง่าย ๆ ด้วยการเปิดหน้าเว็บขึ้นมา แล้วกรอก URL ก็จะได้เห็นข้อความจากไฟล์ font.json แสดงโชว์ออกมาดังภาพ



มีเรื่องที่ผมต้องมาส์มอยนิทหนึ่ง

- สำหรับใครที่ยังไม่รู้จัก REST ...เราไม่จำเป็นต้องใช้เว็บเบราว์เซอร์ส่งรีเควสต์ แต่สามารถเขียนโปรแกรมทำตัวเป็นไคลเอนต์ขึ้นมา แล้วส่งรีเควสต์ไปขอใช้บริการจากเซิร์ฟเวอร์ก็ได้
- สำหรับ REST ในตัวอย่างนี้จะใช้ METHOD เป็น GET แต่จริง ๆ แล้วยังสามารถใช้เป็น PUT, DELETE, POST ได้อีกด้วย
- ในตัวอย่างนี้ใช้ REST กับมอดูล Express ...แต่จริง ๆ แล้ว ยังมีมอดูลทางเลือกอื่นที่ใช้แทนได้

สำหรับเรื่องเว็บเซอร์วิสแบบ REST ผมก็จะแสดงง่าย ๆ ผ่านการใช้ Express ให้เห็นเป็นไอเดียเท่านั้นก่อนครับ

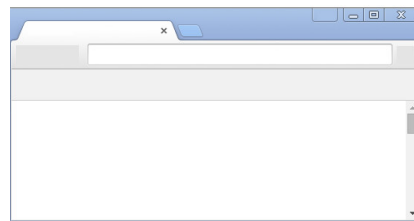
หมายเหตุ

สำหรับรายละเอียดของ express มากกว่านี้ ถ้าสนใจก็อ่านเพิ่มเติมได้ที่

<http://expressjs.com/en/4x/api.html>

สร้างเว็บแอปพลิเคชันแบบเร่งด่วน

บทก่อนหน้านี้ผมได้แนะนำ Express แต่ผมจะให้คุณลิ้มมันไปก่อน เพราะวิธีนั้นมันลวกทุ้งเกินไป จึงแนะนำให้ใช้มอดูล express-generator ซึ่งมีเจ้าของเดียวกันกับเอ็นเทร็กซ์ กับโทนาฟ (ตึง ๆ โป๊ะ) ...ล้อเล่นนะครับ เจ้าของเดียวกับ express ซึ่งมันสร้างเว็บได้ง่ายกว่า



สำหรับวิธีนี้ ผมก็ดัดแปลงมาจากลิงค์นี้แหละ <http://expressjs.com/en/starter/generator.html> และคุณไม่ต้องคิดมาก ทำตามขั้นตอนต่อไปนี่เลย ...ซึ่งมันจะสร้างเว็บได้รวดเร็ว สมกับที่มันชื่อ express ที่แปลว่า “เร่งด่วน” แต่ทว่าเราต้องพิมพ์คำสั่งเยอะหน่อย แหะ ๆ ๆ

ขั้นตอนที่ 1

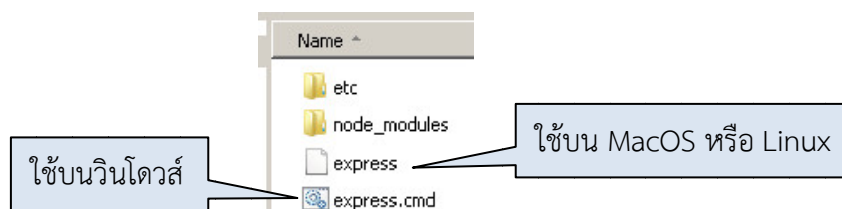
ติดตั้งมอดูล express-generator ด้วยคำสั่งต่อไปนี้

```
npm install express-generator -g
```

เมื่อใช้แฟล็กเป็น -g จะติดตั้งมอดูลไว้ที่ไดเรกทอรีซึ่งได้ติดตั้ง npm เอาไว้

ก่อนจะไปขั้นตอนถัดไป ก็อยากให้คุณลองเปิดไดเรกทอรีดังต่อไปนี้

- ในวินโดวส์ก็ให้ไปที่ “C:\Users\username\AppData\Roaming\npm”
(เมื่อ username คือชื่อโฟลเดอร์ของผู้ใช้งานบนเครื่อง)
- แต่ถ้าเป็น MacOS หรือ Linux ก็ให้ไปที่ “/usr/local/share/npm”



จากภาพที่เห็นจะเป็นไฟล์ express (ใช้บน MacOS หรือ Linux) กับ express.cmd (ใช้บนวินโดวส์) ซึ่งจะอยู่ในไดเรกทอรีดังที่กล่าวมา โดยมันจะใช้เป็นตัวรันคำสั่ง “express” ที่จะได้เห็นในขั้นตอนถัดไป

ขั้นตอนที่ 2

จะให้พิมพ์คำสั่งแบบนี้ครับ

express myapp

ชื่อโปรเจกต์เรา จะสร้างเป็นชื่ออะไรก็ได้

ซึ่งคำสั่งนี้จะสร้างโฟลเดอร์ myapp โดยดูผลการทำงาน จากหน้าจอข้างล่างเลยครับ

C:\>express myapp

```
create : myapp
create : myapp/package.json
create : myapp/app.js
create : myapp/public
create : myapp/public/javascripts
create : myapp/public/images
create : myapp/public/stylesheets
create : myapp/public/stylesheets/style.css
create : myapp/routes
create : myapp/routes/index.js
create : myapp/routes/users.js
create : myapp/views
create : myapp/views/index.jade
create : myapp/views/layout.jade
create : myapp/views/error.jade
create : myapp/bin
create : myapp/bin/www
```

สร้างมาให้เราใช้งานเสร็จสรรพ

install dependencies:

> cd myapp && npm install

พิมพ์คำสั่งนี้ในขั้นตอนที่ 3

run the app:

> SET DEBUG=myapp:* & npm start

พิมพ์คำสั่งนี้ในขั้นตอนที่ 4

ขั้นตอนที่ 3

จะให้ cd เข้าไปในโฟลเดอร์ myapp ที่ถูกสร้างขึ้นมา (ในขั้นตอนที่ 2)

```
C:\>cd myapp
```

```
C:\myapp>
```

ซึ่งมันจะมีโครงสร้างโปรเจกต์ดังต่อไปนี้



ไฟล์ที่เห็นทั้งหมด เดี่ยวจะอธิบายภายหลังนะครับ ...แต่จะให้ลองเปิดดูไฟล์ package.json ขึ้นมาก่อน ซึ่งจะเห็นชื่อ dependencies ต่าง ๆ แต่ทว่ามันยังไม่ถูกดาวน์โหลดมาติดตั้งให้ครับ

ด้วยเหตุนี้เราจึงต้องติดตั้งด้วยมือตนเอง ด้วยคำสั่งข้างล่าง (เสียเวลารอนานหน่อย)

```
C:\myapp>npm install
```

ขั้นตอนที่ 4

ขั้นตอนนี้จะเป็นการรันไฟล์ "www" หรือสั่งให้เว็บแอปพลิเคชันทำงาน ก็ให้ใช้คำสั่ง

```
set DEBUG=myapp:* & npm start
```

ซึ่งจะให้ผลลัพธ์ดังนี้

เอาไว้ debug

```
C:\myapp>set DEBUG=myapp:* & npm start
```

```
> myapp@0.0.0 start C:\myapp
```

```
> node ./bin/www
```

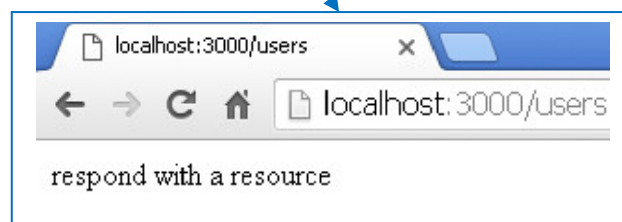
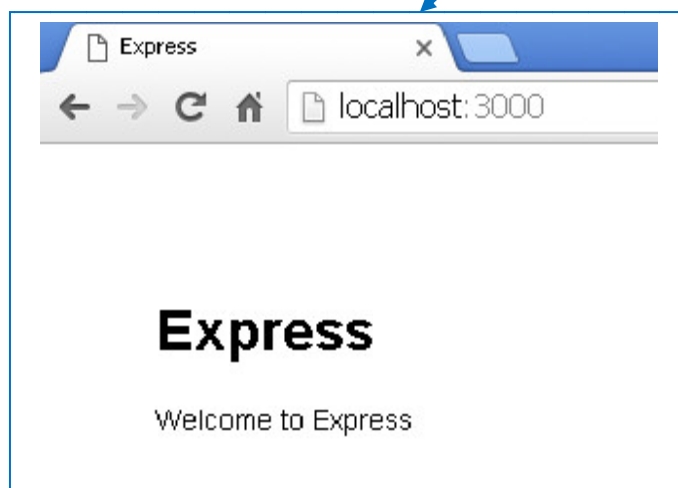
```
myapp:server Listening on port 3000 +0ms
```

แต่ถ้าคุณใช้งานบน MacOS หรือ Linux ก็ให้ใช้คำสั่งเหล่านี้แทน

```
$ DEBUG=myapp:* npm start
```

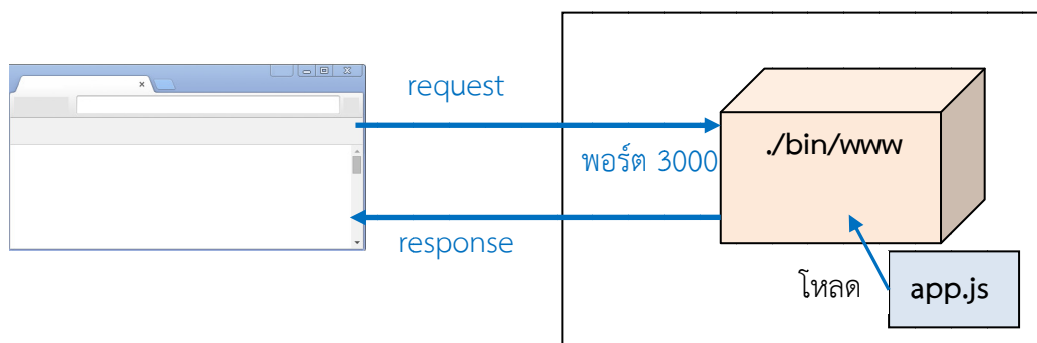
ขั้นตอนที่ 5

ลองเปิดเว็บเบราว์เซอร์เพื่อกรอก URL เป็น <http://localhost:3000> กับ <http://localhost:3000/users> ก็จะได้ผลลัพธ์ดังภาพ



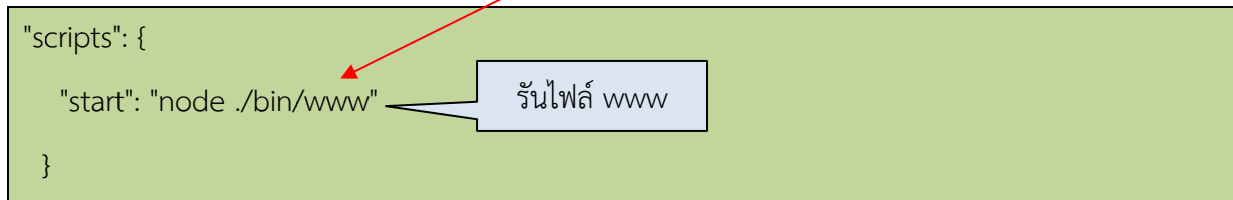
ดูการทำงานของโปรเจค

ในหัวข้อนี้ ผมจะพามาดูการทำงานของโปรเจค **myapp** กันดีกว่า แต่ผมจะอธิบายด้วยรูปภาพแล้วกันนะ

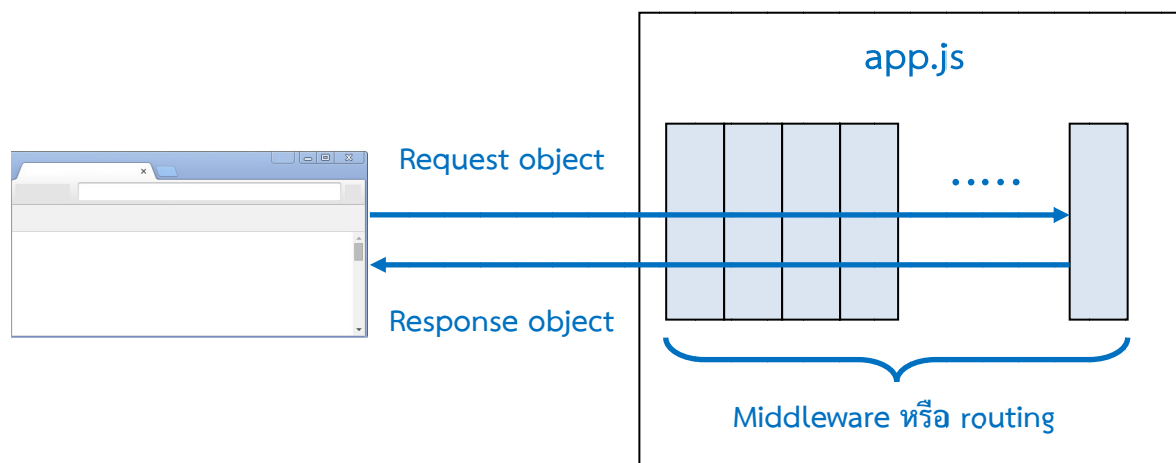


สำหรับไฟล์ `./bin/www` มันจะทำงานเป็นเซิร์ฟเวอร์ (เปิดพอร์ต 3000) ซึ่งเบื้องหลังมันจะโหลดมอดูล `app.js` เข้ามาอีก

***ในไฟล์ `package.json` ถ้าเราไปเปิดดู มันจะสั่งรันโปรเจคด้วยคำสั่งดังต่อไปนี้



สำหรับไฟล์ `app.js` มันเป็นมอดูลตัวหนึ่ง ที่ใช้กำหนดค่าต่าง ๆ ให้กับเว็บแอปพลิเคชัน ตามภาพตัวอย่างข้างล่าง



ต้องอธิบายอย่างนั้นะครับ ... Express มันเป็นเว็บเฟรมเวิร์ค ซึ่งทำหน้าที่หลัก 2 อย่างคือ routing กับ middleware และผมจะอธิบายภาพในหน้าที่แล้ว ในแง่ของการโค้ดดังนี้

- Middleware มันคือกล่องสีฟ้าในรูป ที่นำมาวางต่อกันเรื่อย ๆ เพื่อจัดการ Request object กับ Response object
- ส่วน Routing ก็คือ middleware นั้นแหละ ...แต่จะทำงานเฉพาะอย่างคือ “เลือกเส้นทาง” หมายความว่า มันจะเป็นตัวเลือกว่า ถ้ารีเควสต์ URL เข้ามาเป็นอะไร ก็จะเลือกหน้าเว็บไปแสดงผลบนเว็บเบราว์เซอร์ (เดี๋ยวจะได้เห็นตัวอย่างในหัวข้อถัด ๆ ไป ...รอซักครู)

สำหรับการกำหนด middleware ก็จะมีรูปแบบการเขียนโค้ดดังนี้

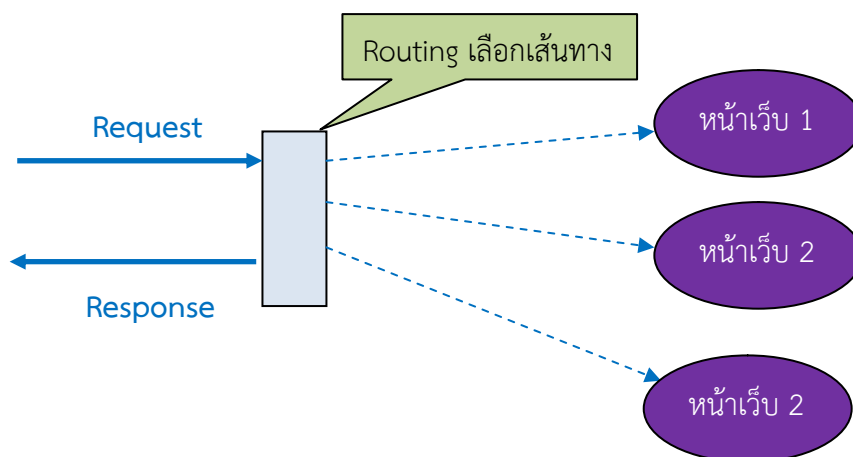
```
app.use(middleware_1);
```

middleware_1 มันคือคอลแบ็ค ซึ่งทำหน้าที่เป็น middleware

ส่วนการกำหนด routing ก็จะมีรูปแบบการเขียนโค้ดดังนี้

```
app.use("urlPath", middleware_2);
```

- โดยที่ middleware_2 มันคือคอลแบ็ค ซึ่งทำหน้าที่เป็น routing
- ส่วน "urlPath" ก็คือรีเควสต์ (พารามิเตอร์ของ URL) ที่วิ่งเข้ามา



ถ้าเราเปิดไฟล์ app.js ขึ้นมาดู ...คุณก็จะเห็นโค้ดตามภาพหน้าถัดไป

```

16
17 // uncomment after placing your favicon in /public
18 //app.use(favicon(path.join(__dirname, 'public', 'favicon.ico')));
19 app.use(logger('dev'));
20 app.use(bodyParser.json());
21 app.use(bodyParser.urlencoded({ extended: false }));
22 app.use(cookieParser());
23 app.use(express.static(path.join(__dirname, 'public')));
24
25 app.use('/', routes);
26 app.use('/users', users);
27
28 // catch 404 and forward to error handler
29 app.use(function(req, res, next) {
30   var err = new Error('Not Found');
31   err.status = 404;
32   next(err);
33 });
34
35 // error handlers
36
37 // development error handler
38 // will print stacktrace
39 if (app.get('env') === 'development') {
40   app.use(function(err, req, res, next) {
41     res.status(err.status || 500);
42     res.render('error', {
43       message: err.message,
44       error: err
45     });
46   });
47 }
48
49 // production error handler
50 // no stacktraces leaked to user
51 app.use(function(err, req, res, next) {
52   res.status(err.status || 500);
53   res.render('error', {
54     message: err.message,
55     error: {}
56   });
57 });

```

Middleware

Middleware ที่เป็น Routing

Middleware ที่จัดการ error

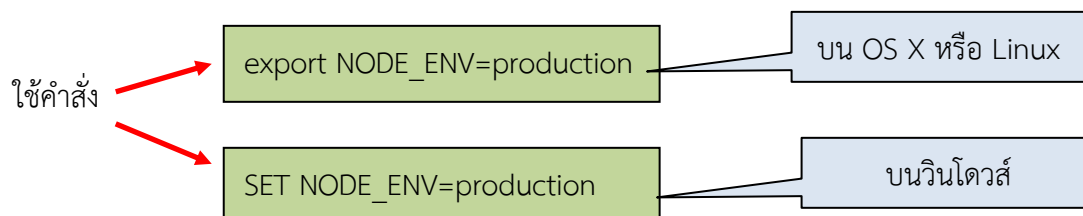
จากภาพที่ตีกรอบจุดไข่ปลาสีแดง หมายเลข 1,2,3,4,5 มันคือ middleware ที่มีรายละเอียดที่น่าสนใจดังนี้

- หมายเลข 1 คือ middleware ที่ถูกติดตั้งมาให้เป็นค่าดีฟอลต์ของ Express (ขอไม่ลงรายละเอียด)
- หมายเลข 2 คือ middleware ที่ทำตัวเป็น Routing
- หมายเลข 3 คือ middleware เอาไว้จัดการ error เมื่อหาไฟล์ html มันพบ (สถานะ error เป็น 404)
- หมายเลข 4 คือ middleware เอาไว้จัดการ error ที่มีสถานะเป็น 500
- หมายเลข 5 คือ middleware เอาไว้จัดการ error ที่มีสถานะเป็น 500

แต่ทั้งนี้เราต้องเลือกว่า จะให้ middleware หมายเลข 4 หรือ 5 ตัวไหนมันทำงาน เพราะว่า

- Middleware หมายเลข 4 จะใช้งานในโหมด development
- Middleware หมายเลข 5 จะใช้งานในโหมด production

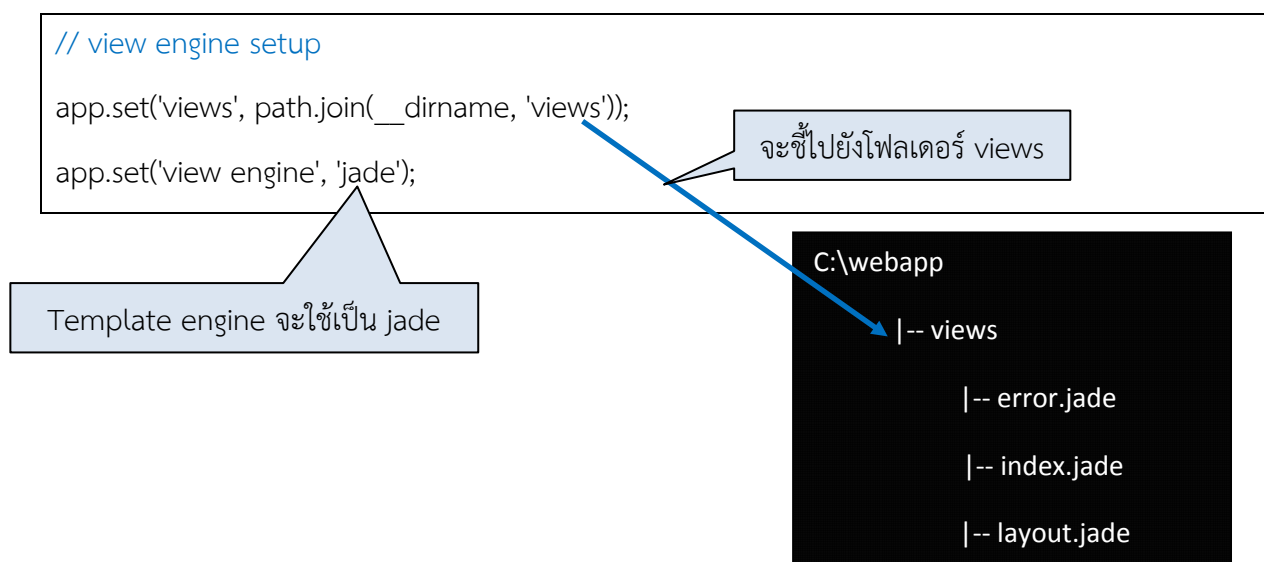
การใช้งานแบบ production ก็คือการกำหนดค่าในตัวแปร NODE_ENV ซึ่งต้องทำก่อนรันโปรเจค ดังนี้



***แต่ถ้าเราไม่ได้กำหนดค่าอะไรไว้เลย ก็จะหมายถึง development โดยดีฟอลต์

Template engine

การใช้งาน Express เราสามารถกำหนด template engine ...ซึ่งมันเป็นกลไก เอาไว้ใช้แสดงผลบนหน้าเว็บเบราว์เซอร์ และในไฟล์ app.js ก็จะมีโค้ด 2 บรรทัด ดังนี้



ในตัวอย่างนี้จะใช้ template engine เป็น Jade ซึ่งจะใช้ไฟล์นามสกุล *.jade เป็นตัวแสดงผลสำหรับไฟล์ *.jade ภายหลังจะถูก render หรือแปลงร่างกลายเป็น HTML บนเว็บเบราว์เซอร์อีกที

และเมื่อดูโค้ดใน app.js ก็จะมีการกำหนดค่า routing เป็นดังนี้

```
var routes = require('./routes/index');
```

```
var users = require('./routes/users');
```

```
.....
```

```
.....
```

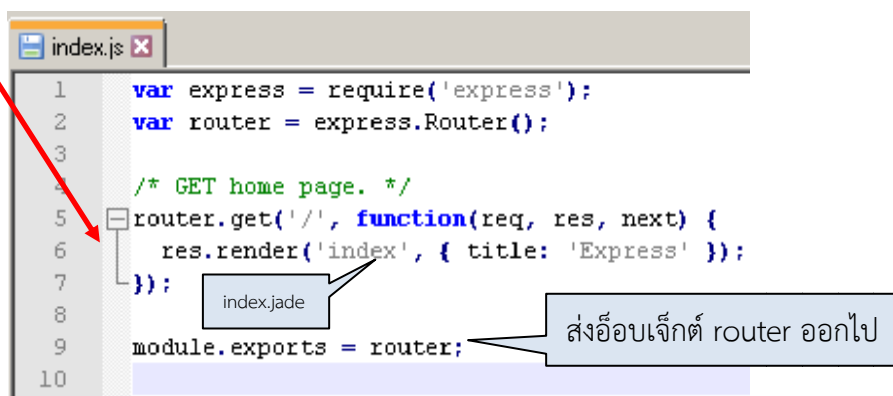
```
app.use('/', routes);
```

```
app.use('/users', users);
```

กำหนด routing

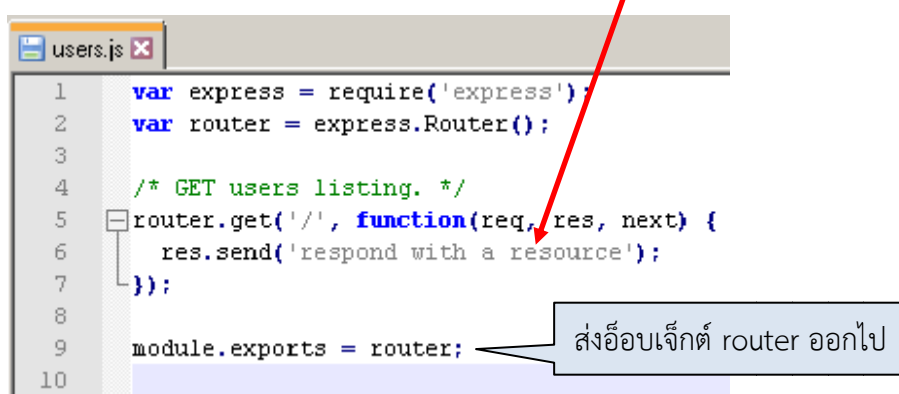
มอดูล routes กับ users มันจะโหลดมาจากไฟล์ ./routes/index.js และ ./routes/users.js และถูกนำไปกำหนดให้เป็น routing ในเมธอด app.use() ...โดยระบุพารามิเตอร์เป็น / กับ /users ตามลำดับ

โดยที่ไฟล์ index.js จะเป็นตัวเลือกเส้นทาง เพื่อ render ไฟล์ index.jade ให้แสดงผลบนหน้าเว็บเบราว์เซอร์



```
1 var express = require('express');
2 var router = express.Router();
3
4 /* GET home page. */
5 router.get('/', function(req, res, next) {
6   res.render('index', { title: 'Express' });
7 });
8
9 module.exports = router;
```

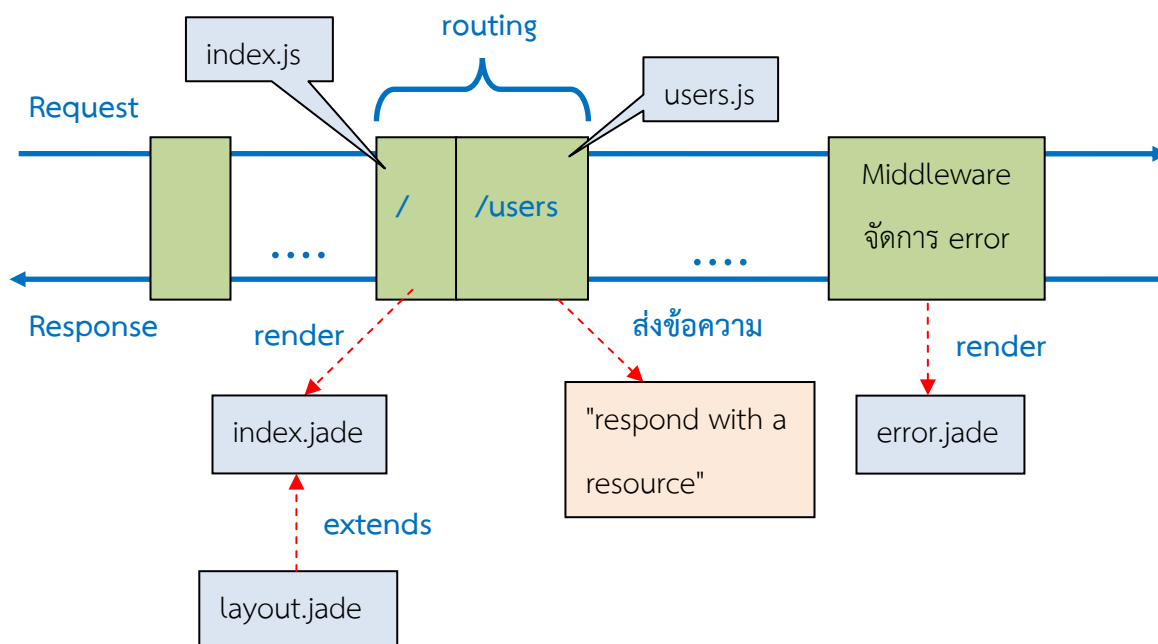
ส่วนไฟล์ users.js จะส่งข้อความกลับไปหาเว็บเบราว์เซอร์โดยตรง เป็น 'respond with a resource'



```
1 var express = require('express');
2 var router = express.Router();
3
4 /* GET users listing. */
5 router.get('/', function(req, res, next) {
6   res.send('respond with a resource');
7 });
8
9 module.exports = router;
```

แต่ถ้าเกิด error ขึ้นมา ก็จะมีตัว Middleware มา render ไฟล์ error.jade (ให้ดูโค้ด app.js อีกทีนะคะ)

สำหรับ index.js กับ users.js เวลากำหนดให้เป็น routing นั้น ...ผมจะอธิบายด้วยภาพข้างล่างแล้วกัน



จากรูป สามารถอธิบายได้โดยย่อดังนี้

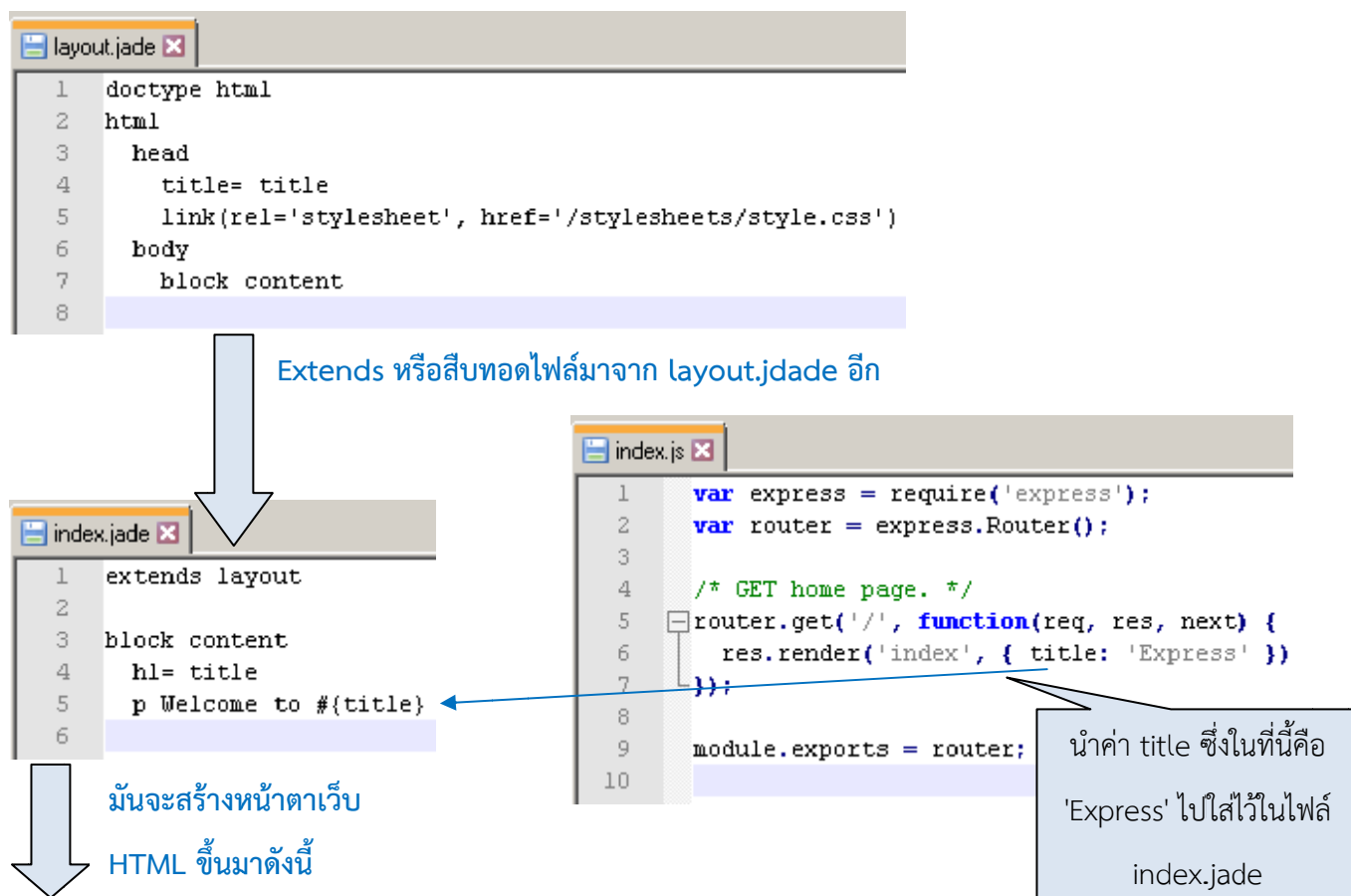
- ถ้ารีเควสเข้ามาเป็น URL <http://ipaddress:3000> ก็จะใช้ index.jade เป็นตัวแสดงผลหน้าเว็บ
- แต่ถ้าเป็น <http://ipaddress:3000/users> ก็ส่งข้อความ "respond with a resource" กลับไป
- แต่ถ้าเกิด error ขึ้นมา ก็จะใช้ error.jade เป็นตัวแสดงผลหน้าเว็บ

จะขอย้อนกลับไปโค้ดในไฟล์ index.js กับ users.js มันจะเรียกใช้อ็อบเจกต์เราเตอร์ (Router) ดังนี้

```
var router = express.Router(); // เป็นอ็อบเจกต์เราเตอร์
....
router.get('path', function(req, res, next) {
  /*... โค้ดเรา ...*/
});
```

ในตัวอย่างนี้ router นอกจากจะมีเมธอด get แล้ว ...มันยังมีเมธอด post, delete และอื่น ๆ ตามชื่อ METHOD ของโปรโตคอล HTTP อีกด้วย

เมื่อผมเปิดไฟล์ layout.jade กับ index.jade ขึ้นมาดู ก็จะเห็นว่าทั้ง 2 ไฟล์มีความสัมพันธ์กันดังนี้



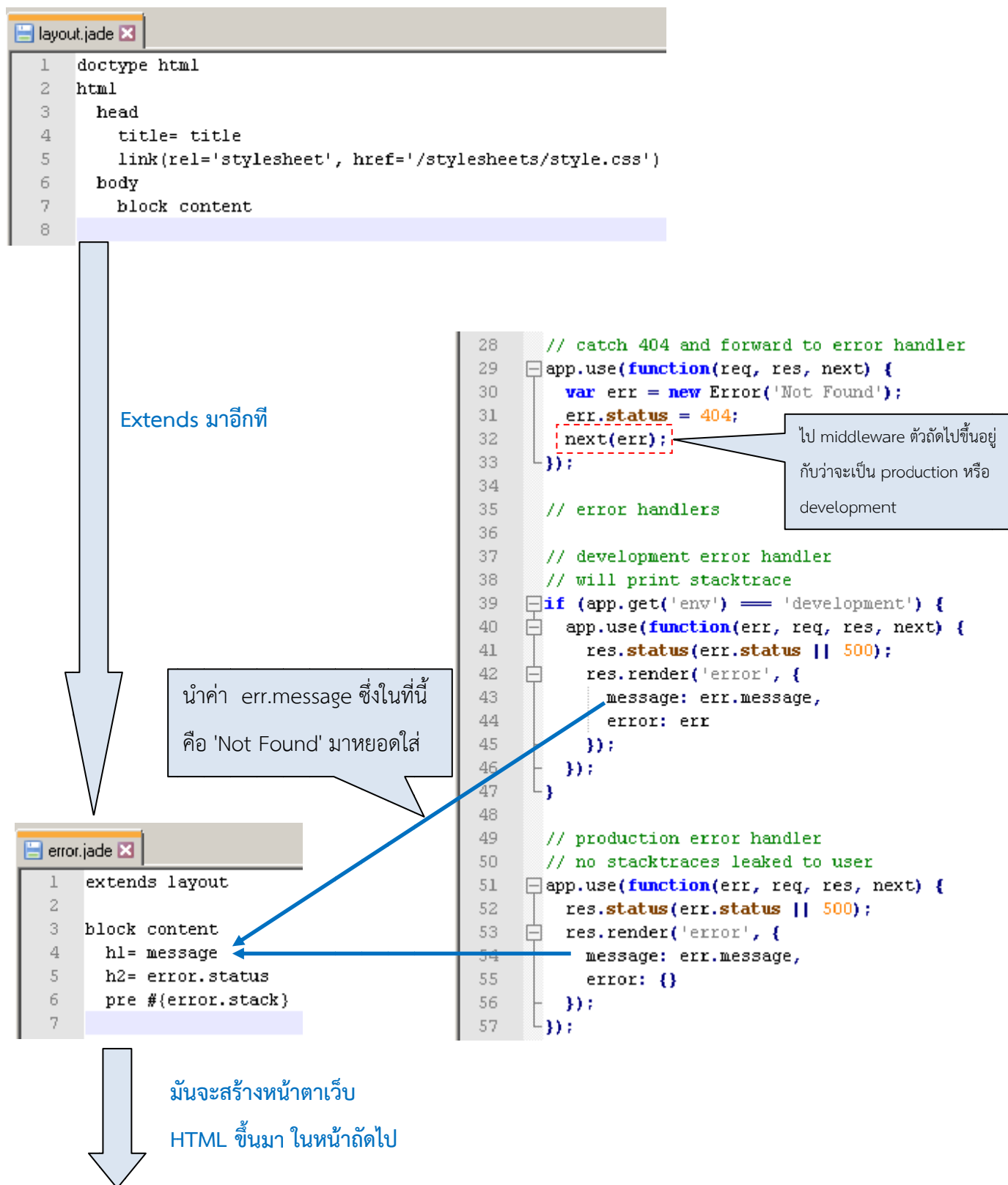
```

<!DOCTYPE html>
<html>
<head>
  <title>Express</title>
  <link rel="stylesheet" href="/stylesheets/style.css">
</head>
<body>
  <h1>Express</h1>
  <p>Welcome to Express</p>
</body>
</html>

```

ผมคิดว่า ถ้าใครเคยเขียน PHP, ASP และ JSP มาก่อน มันจะเหมือนเขียนสคริปต์แทรกเข้าไปในไฟล์ HTML แต่บังเอิญ Jade ...มันดันสร้างไวยากรณ์ขึ้นมาใหม่ โดยปราศจากแท็ก HTML ให้เกะกะนะครับ

ถ้าสมมติเรากรอก URL ผิดเป็น <http://localhost:3000/ที่ไม่มีบุ๊ก-ที่ไหนมีทุกซ์> ตัวไฟล์ error.jade ก็จะทำให้หน้าแสดงผลแทน ...เนื่องจากหาหน้าเว็บไม่พบ (error 400) ซึ่งไฟล์ error.jade ก็จะมีการทำงานตามภาพ



```

<!DOCTYPE html>

<html>

<head>

  <title></title>

  <link rel="stylesheet" href="/stylesheets/style.css">

</head>

<body>

  <h1>Not Found</h1>

  <h2></h2>

  <pre></pre>

</body>

</html>

```

Jade มันก็ยังมีคำสั่งต่าง ๆ มากมาย เช่น if else และคำสั่งอื่น ๆ ไม่ต่างอะไรกับภาษาสคริปต์ทั่ว ๆ ไป ดังตัวอย่าง

```

doctype html
html(lang="en")
  head
    title= pageTitle
    script(type='text/javascript').
      if (foo) {
        bar(1 + 5)
      }
  body
    h1 Jade - node template engine
    #container.col
      if youAreUsingJade
        p You are amazing
      else
        p Get on it!
      p.
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.

```

หน้าตาเว็บ HTML

```

<!DOCTYPE html>
<html lang="en">
  <head>
    <title>Jade</title>
    <script type="text/javascript">
      if (foo) {
        bar(1 + 5)
      }
    </script>
  </head>
  <body>
    <h1>Jade - node template engine</h1>
    <div id="container" class="col">
      <p>You are amazing</p>
      <p>
        Jade is a terse and simple
        templating language with a
        strong focus on performance
        and powerful features.
      </p>
    </div>
  </body>
</html>

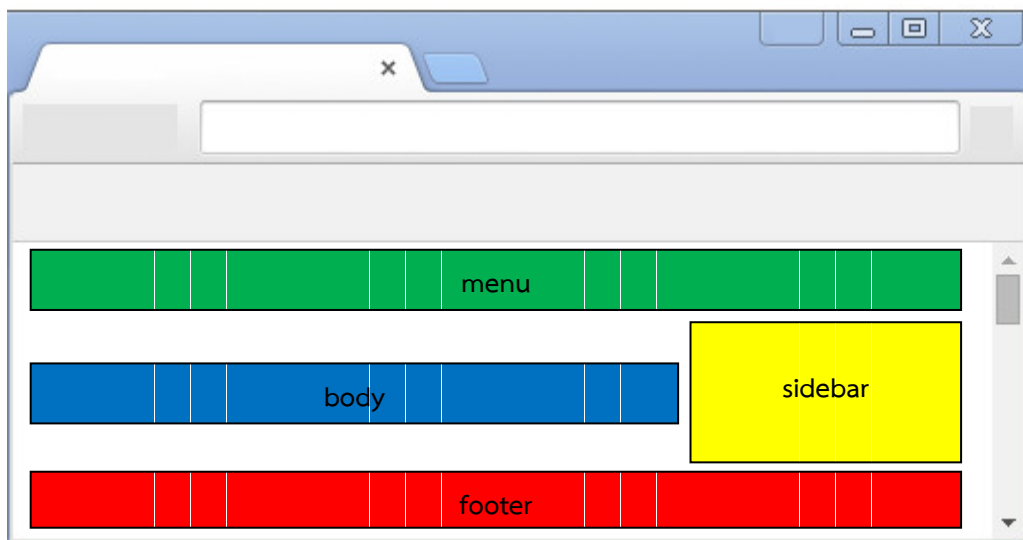
```

หมายเหตุ โค้ดที่เห็นนี้ นำมาจากเว็บไซต์หลักของเขา <http://jade-lang.com/> ลองศึกษาเพิ่มเติมได้ครับ

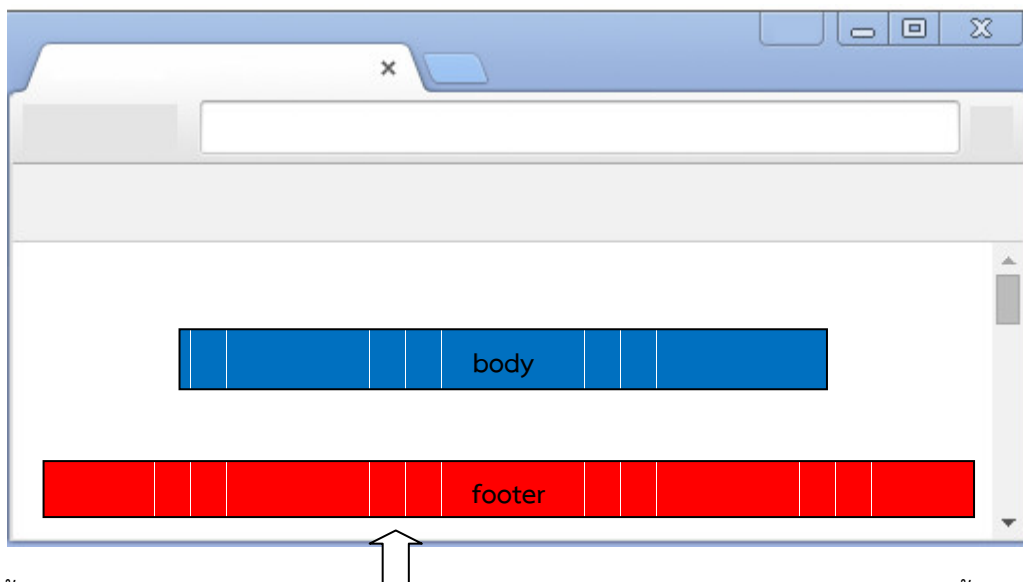
เทมเพลตหน้าเว็บ

ในบทนี้จะพูดถึงมอดูล EJS ซึ่งมีไว้แบ่งหน้าจออกเป็นส่วน ๆ โดยแต่ละชิ้นส่วนอาจเรียกว่าคอมโพเนนต์ (Component) โดยแต่ละคอมโพเนนต์สามารถถูก reuse หรือนำกลับมาใช้งานใหม่ได้

...โดย EJS ก็คือ template engine เหมือนกับ Jade แต่มันจะเป็นคนละคอนเซปต์กัน ดังภาพ



ในภาพนี้ หน้าเว็บจะประกอบไปด้วยคอมโพเนนต์ ได้แก่ menu, body, footer และ sidebar



ในภาพนี้ หน้าเว็บจะประกอบไปด้วยคอมโพเนนต์ body และ footer (นำกลับมาใช้งานใหม่อีกครั้ง)

หน้าเว็บที่ผมยกมาให้ดู มันจะเป็นโครงร่าง หรือเทมเพลต (Template) ที่ถูกประกอบขึ้นมาจากคอมโพเนนต์ต่าง ๆ ...และเพื่อไม่ให้เป็นการเสียเวลา ก็จะให้คุณดูโค้ดที่ใช้แสดงผลหน้าเว็บในหน้าถัดไป

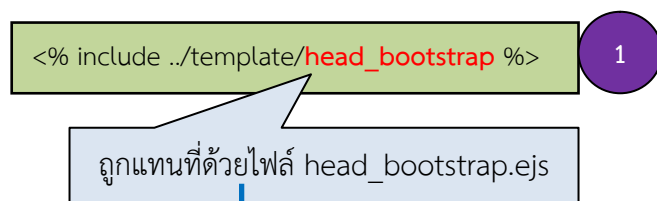
```

<!-- views/pages/home.ejs -->

<!DOCTYPE html>
<html lang="en">
<head>
  <% include ../template/head_bootstrap %>
</head>
<body class="container">
  <header><% include ../template/menu %></header>
  <main>
    <div class="jumbotron">
      <h1>โปรแกรมเมอร์ไทย</h1>
      <p>ในใจนะเจ็บปวด</p>
    </div>
  </main>
  <footer><% include ../template/footer %></footer>
</body>
</html>

```

ในตัวอย่างนี้ไฟล์ “home.ejs” จะเขียนเป็นโครงร่างหน้าเว็บเอาไว้ก่อน โดยมันจะมีนามสกุล .ejs ...และคุณน่าจะเห็นแท็ก <% %> ที่ตีกรอบสีแดงไข่ม้อยครับ ...พวกมันจะถูกแทนที่ด้วยแท็ก HTML แล้วกลายเป็นหน้าเว็บที่สมบูรณ์ภายหลัง โดยผมจะอธิบายไค้ดหมายเลข 1, 2, 3 ตามลำดับดังนี้



```

<!-- views/partials/head_bootstrap.ejs -->

```

```

<meta charset="UTF-8">

```

```

<title>ตัวอย่างการใช้ EJS</title>

```

```

<!-- CSS (load bootstrap from a CDN) -->

```

```

<!-- Latest compiled and minified CSS -->

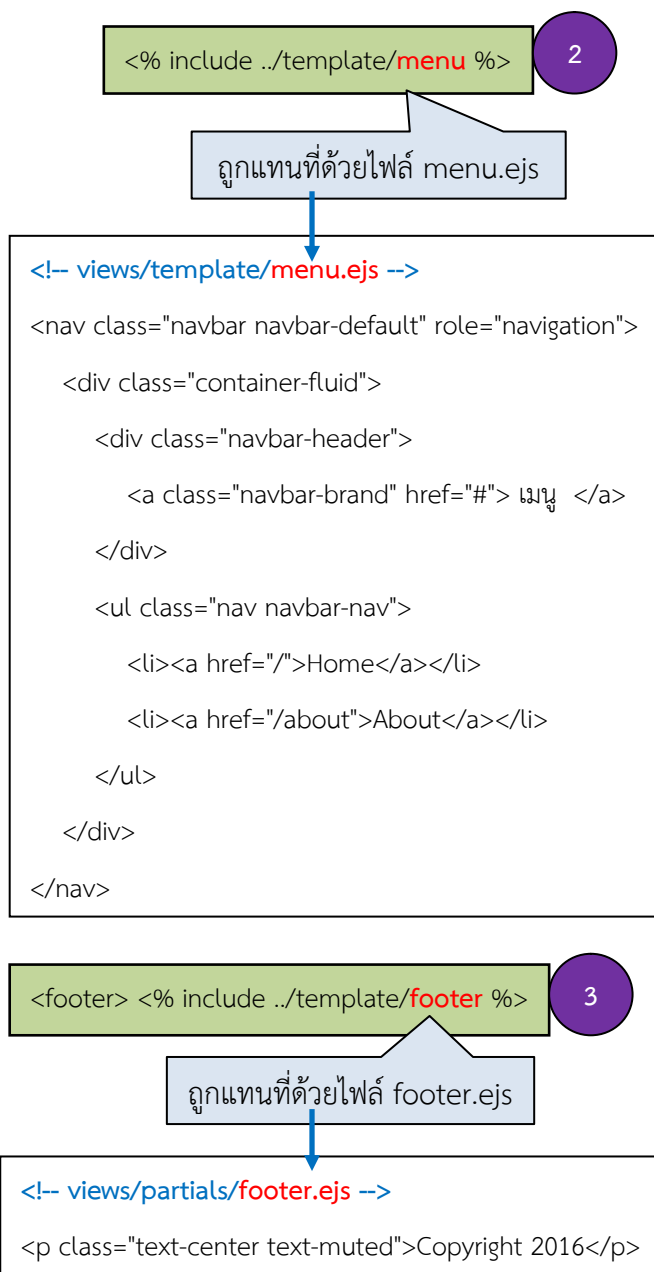
```

```

<link rel="stylesheet" href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">

```

ใช้ bootstrap ซึ่งเป็น CSS เฟรมเวิร์คยอดนิยมตัวหนึ่งสำหรับเอาไว้แสดงผลหน้าเว็บแบบ responsive ซึ่งตอนที่เขียนหนังสือยังเป็นเวอร์ชัน 3 (เวอร์ชัน 4 เป็น alpha)



ไฟล์ *.ejs ที่ผมยกตัวอย่างมาทั้งหมด มันจะถูก render เพื่อแสดงผลบนหน้าเว็บเบราว์เซอร์ แบบเดียวกับไฟล์ *.jad ของ Jade ...แต่โครงสร้างไฟล์ยังเป็น HTML ไม่ได้สร้างไวยากรณ์ใหม่ขึ้นมาเหมือน Jade

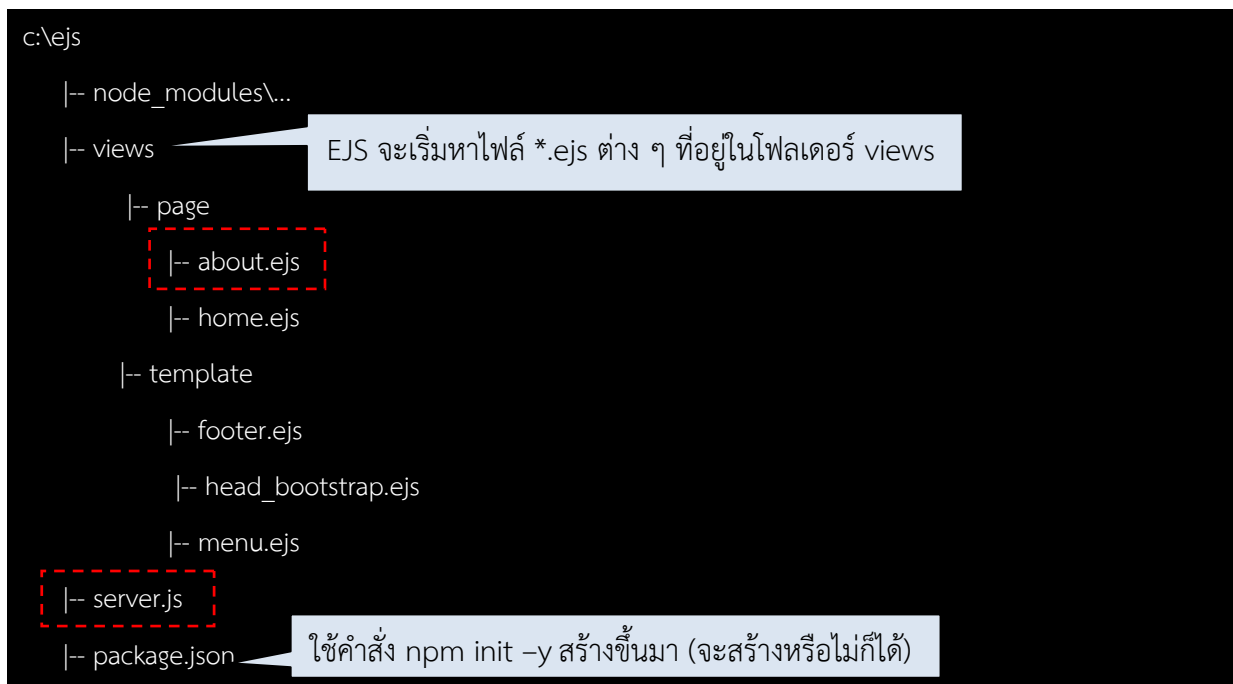
ใช้งานมอดูล EJS

ในตัวอย่างนี้ผมจะสร้างโปรเจกต์แยกออกมา (ชื่อโฟลเดอร์ ejjs) และก็ติดตั้งมอดูล EJS ด้วยคำสั่งดังนี้

```
npm install ejjs-locals -save
```

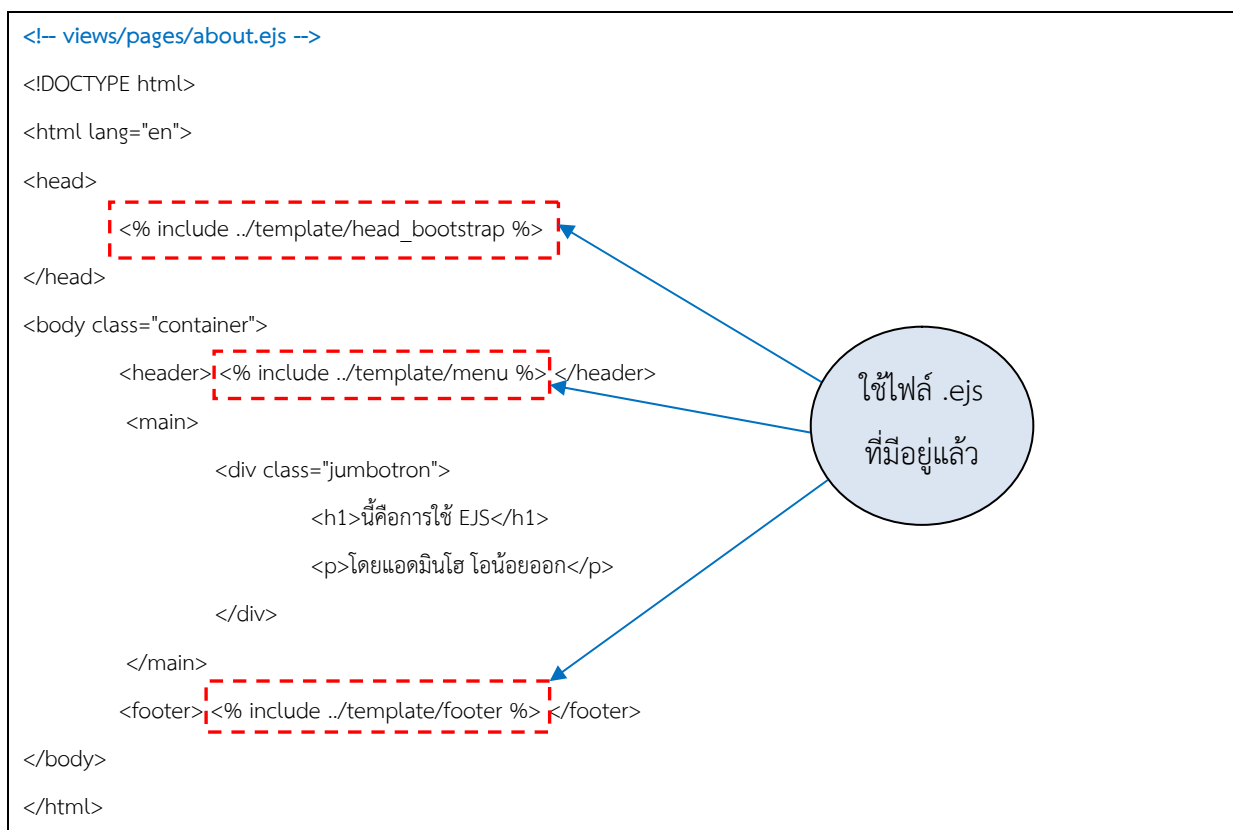
*** ปัจจุบัน EJS จะเป็นเวอร์ชัน 2 แต่ทว่าได้ติดตั้ง ejjs-locals เวอร์ชัน 1 แทน เพราะทดลองใช้แล้วมันโอเคดี

ในตัวอย่างนี้ จะมีโครงสร้างโปรเจกต์ดังตัวอย่าง



ในตัวอย่างนี้ไฟล์ home.ejs, footer.ejs, head_bootstrap.ejs และ menu.ejs จะนำมาจากหัวชื่อก่อนหน้านี้แหละ ...ส่วนไฟล์ about.ejs กับ server.js จะสร้างเพิ่มขึ้นมาใหม่เอี่ยมอ่อง ซึ่งจะเห็นได้ดังต่อไปนี้ ...

ผมจะสร้างไฟล์ “about.ejs” ดังนี้



ไฟล์ about.ejs ก็มีหลักการทำงานเดียวกันกับ home.ejs ซึ่งพวกมันจะเป็นหน้าที่ร่างโครงขึ้นมาก่อน พอถึงเวลาประมวลผลจริงๆ ก็จะนำไฟล์ *.ejs มาหยอดใส่ ให้กลายเป็นหน้าเว็บ HTML ที่สมบูรณ์ภายหลัง

ต่อมาผมจะสร้างไฟล์ server.js ซึ่งโค้ดข้างในจะเป็นการใช้งานมอดูล Express ร่วมกับ EJS ดังต่อไปนี้

```
var express = require('express');
var engine = require('ejs-locals'); // ในตัวอย่างนี้ผมจะใช้มอดูล ejs-locals ไม่ได้โหลดมอดูล ejs โดยตรงนะครับ
var app = express();

app.engine('ejs', engine);
app.set('view engine', 'ejs');
```

บอก Express ว่าจะใช้ EJS เป็น Template engine

```
app.get('/', function(req, res) { // ติดต่อเข้ามาด้วย http://ipaddress:8080/
  res.render('page/home'); // นำไฟล์ home.js ไปแสดงผล
});

app.get('/about', function(req, res) { // ติดต่อเข้ามาด้วย http://ipaddress:8080/about
  res.render('page/about'); // นำไฟล์ about.js ไปแสดงผล
});

app.listen(8080, function() { // ทำตัวเป็นเซิร์ฟเวอร์ และเปิดพอร์ต 8080
  console.log('Server running at http://localhost:8080/');
});

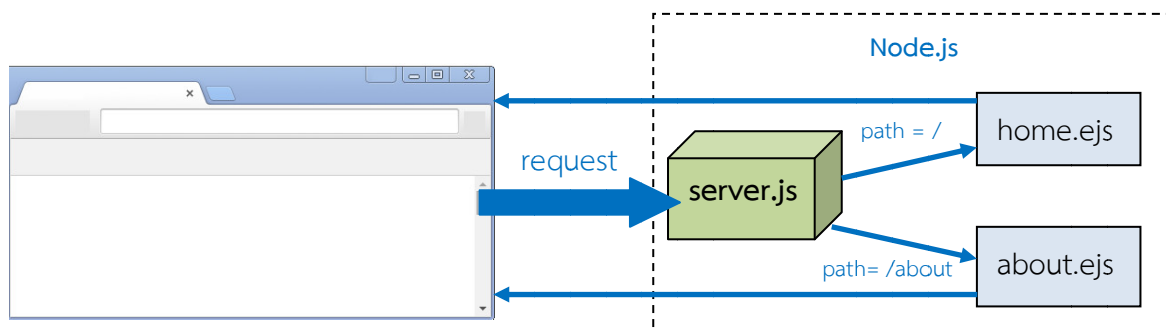
console.log("Using EJS example");
```

ไฟล์ server.js จะทำตัวเป็นเซิร์ฟเวอร์และเปิดพอร์ต 8080 เมื่อเว็บเบราว์เซอร์ติดต่อเข้ามาด้วยพาธ ได้แก่ / หรือ /about ก็ sẽนำไฟล์ home.js หรือ about.js ไปแสดงผลหน้าเว็บเบราว์เซอร์ ...หรือพูดอีกนัยหนึ่ง server.js มันทำตัวเป็นทั้งเซิร์ฟเวอร์ และเราเตอร์เพื่อเลือกไฟล์ (*.ejs) มาแสดงผล

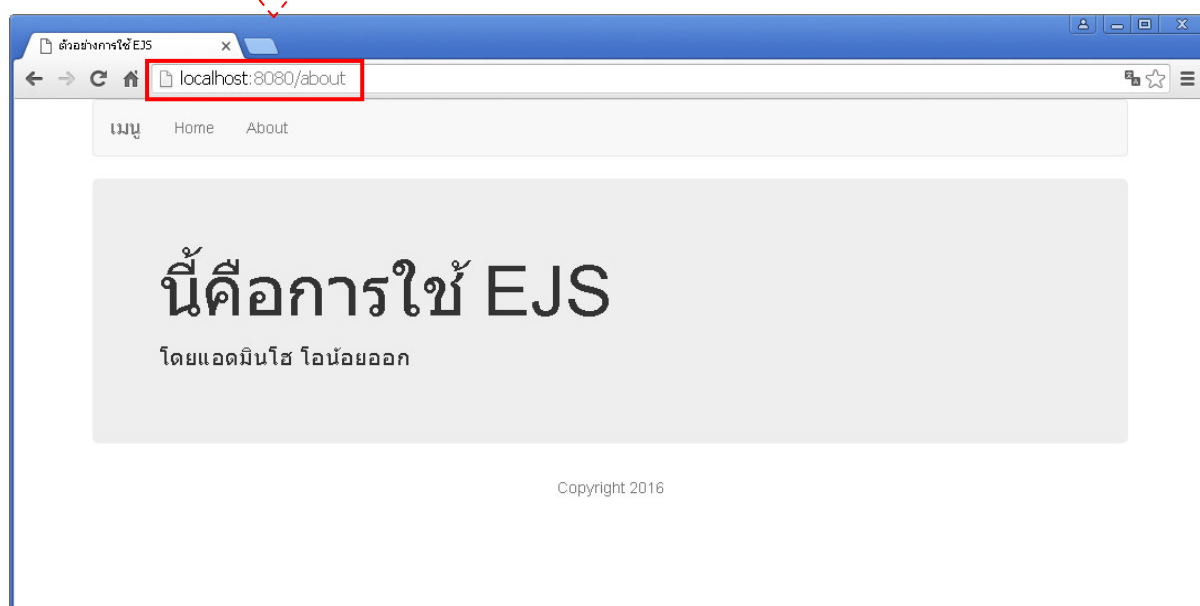
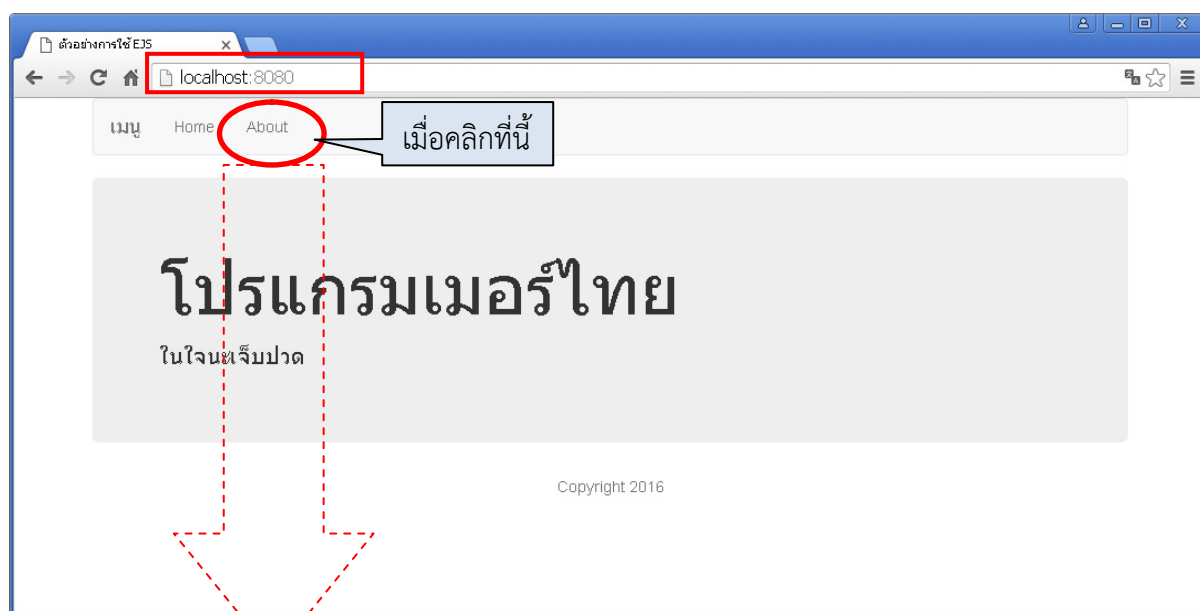
แต่ก่อนอื่นอย่าลืมรันคำสั่ง “node server.js” เพื่อให้เว็บแอปทำงาน ดังนี้

```
c:\ejs>node server.js
Using EJS example
Server running at http://localhost:8080/
```

ซึ่งภาพรวมของการทำงานในโปรเจกต์นี้ จะเป็นดังภาพในหน้าถัดไป



ถ้ากรอก URL บนเว็บเบราว์เซอร์เป็น <http://localhost:8080/> ก็จะเห็นหน้าจอการทำงานดังนี้

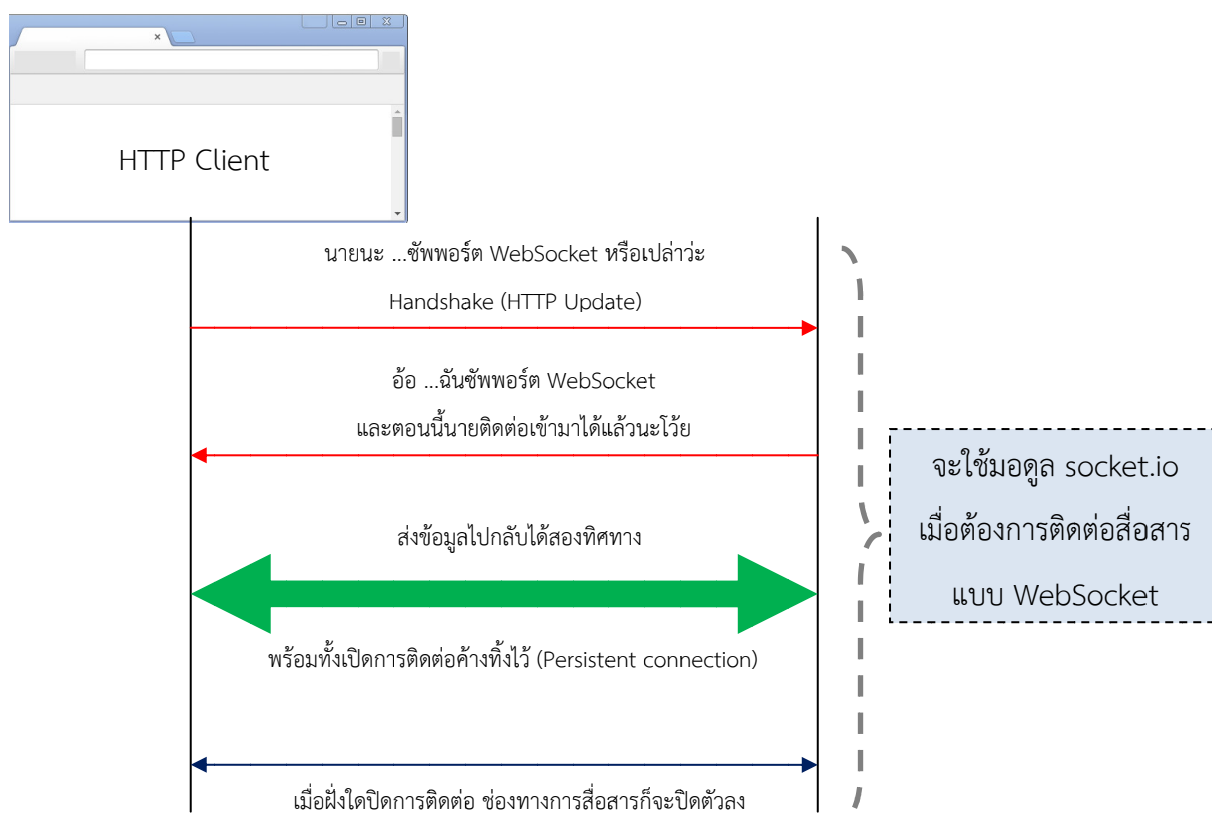


สำหรับ EJS ที่เห็นในบทนี้ มันเป็นแค่หนึ่งในทางเลือก ที่ใช้เป็น Template engine ซึ่งจริง ๆ แล้วยังมีอีกหลายตัว ซึ่งแต่ละอันก็จะมีคอนเซ็ปต์ที่แตกต่างกันไป

เว็บแอปพลิเคชันแบบเรียลไทม์

ในบทนี้จะแสดงตัวอย่างการสร้างเว็บแอปพลิเคชันแบบเรียลไทม์ ด้วยการใช้ WebSocket ซึ่งเป็นโปรโตคอลที่ใช้ติดต่อสื่อสารระหว่างไคลเอนต์ (HTTP Client) และเซิร์ฟเวอร์ (HTTP Server)

โดย WebSocket จะมีช่องทางการติดต่อสื่อสารแบบ full-duplex (ติดต่อสื่อสารได้ 2 ทิศทางบน socket อันเดียว) ส่วนข้อมูลที่ส่งหากันก็จะวิ่งอยู่บนโปรโตคอล TCP อีกที ตามรูปข้างล่าง (มันไม่ใช่โปรโตคอล HTTP นะครับ อย่าสับสนกัน)



WebSocket ผมขอเปรียบเทียบเหมือนกับท่อที่เปิดค้างไว้ เพื่อเชื่อมต่อระหว่างไคลเอนต์ (เว็บเบราว์เซอร์) กับเซิร์ฟเวอร์ และเราก็ใช้เจ้าท่อนี้แหละ ส่งข้อมูลไปกลับหากันได้ 2 ทิศทาง ไม่ว่าจะเป็นไคลเอนต์ส่งไปหาเซิร์ฟเวอร์ หรือเซิร์ฟเวอร์ส่งมาหาไคลเอนต์ ด้วยท่อส่งเพียงอันเดียว แต่ถ้าฝั่งใดปิดท่อก่อนก็จะคุยกันไม่ได้

อยากให้คุณลองนึกถึงโปรโตคอล HTML เวลาที่หน้าเว็บส่งรีควีสต์ไปหาเซิร์ฟเวอร์ หน้าเว็บจะต้องกระพริบทีหนึ่ง (หรือ Refresh ตัวเอง) แล้วถึงนำข้อมูลจากเซิร์ฟเวอร์ มาแสดงผลบนหน้าเว็บ

แต่ถ้าใช้ WebSocket เราสามารถติดต่อแลกเปลี่ยนข้อมูลกับเซิร์ฟเวอร์ โดยไม่ต้อง refresh หน้าเว็บ จริง ๆ แล้วมันจะคล้ายกับ AJAX แต่จะมีประสิทธิภาพดีกว่าเยอะ (คำว่า AJAX น่าจะเข้าใจกันอยู่แล้วเนอะ)

*** ถ้าใครใช้ HTML5 มาก่อน ก็น่าจะเคยเห็นการใช้ WebSocket มาแล้ว ซึ่งมีหลักการเดียวกันแหละครับ 😊

มอดูล socket.io

การใช้งาน WebScket ใน Node.js สามารถใช้ socket.io แต่เราต้องติดตั้งมันก่อนด้วยคำสั่ง

```
npm install socket.io --save
```

หลังจากนี้ต่อไป ผมจะอธิบายการเขียนโค้ดทีละขั้นตอนแล้วกัน ดังรายละเอียดต่อไปนี้

ขั้นที่ 1

เมื่อติดตั้งมอดูลเสร็จเรียบร้อยแล้ว ก็จะทำให้เขียนโค้ดจำลองเซิร์ฟเวอร์ และเปิดพอร์ตขึ้นมา (เป็นอะไรก็ได้) เพื่อใช้ติดต่อสื่อสารแบบ WebSocket ดังนี้

```
var io = require('socket.io').listen(8080); // จะจำลองเซิร์ฟเวอร์ และเปิดพอร์ต 8080 ค้างไว้
```

ขั้นที่ 2

ผมจะเพิ่มซอร์สโค้ดในฝั่งไคลเอนต์ (ในไฟล์ HTML) กับฝั่งเซิร์ฟเวอร์ ดังนี้

// ฝั่งไคลเอนต์

```
io.connect('http://localhost:8080');
```

WebSocket

// ฝั่งเซิร์ฟเวอร์

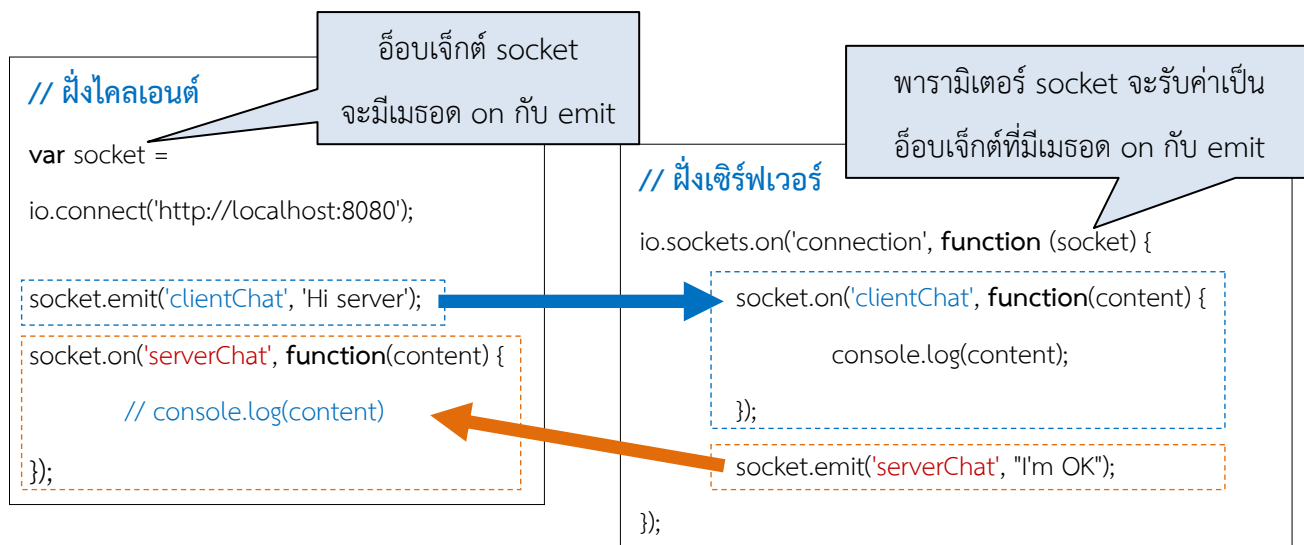
```
io.sockets.on('connection', function (socket) {
  // เมื่อไคลเอนต์ติดต่อเข้ามา
  // โค้ดส่วนนี้จะทำงาน
});
```

เกิดเหตุการณ์ 'connection'

จากโค้ดที่ยกมาให้ดู เวลาไคลเอนต์ติดต่อเข้ามาเป็น URL ธรรมดา ได้แก่ <http://localhost:8080> ...เมื่อนั้นคอลแบ็กของเมธอด on() ในฝั่งเซิร์ฟเวอร์ ก็จะถูกเรียกให้ทำงาน

ขั้นตอนที่ 3

ผมจะแสดงให้เห็นว่า เวลาไคลเอนต์กับเซิร์ฟเวอร์ส่งข้อมูลไปมาหาสู่กัน จะมีวิธีเขียนโค้ดดังต่อไปนี้



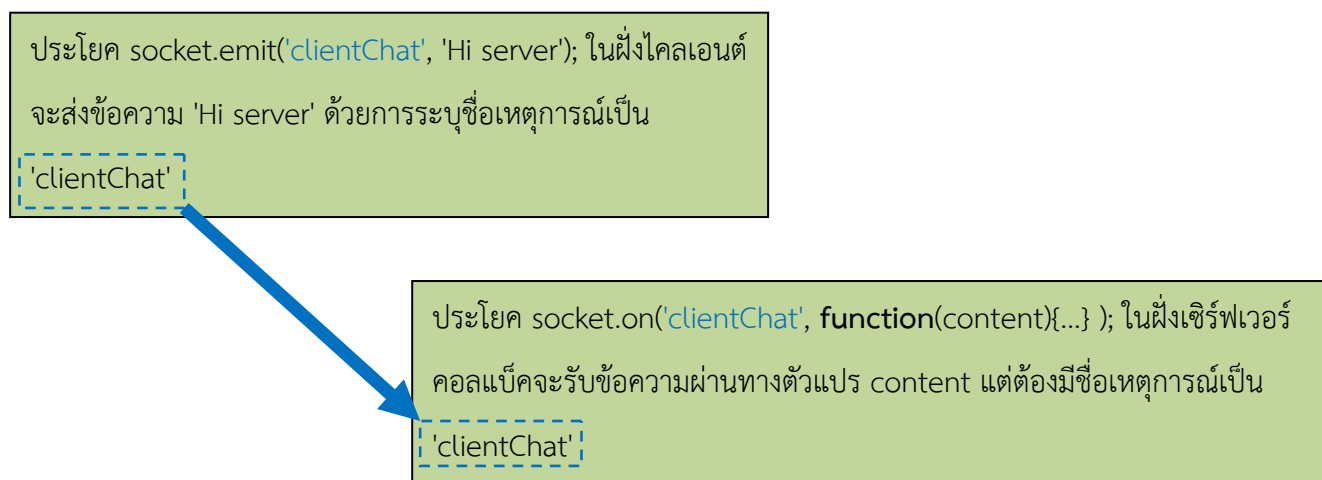
ต้องเข้าใจอย่างนั้นนะครับ โมดูล socket.io จะใช้กลไกติดต่อสื่อสารเป็นแบบ message-oriented หรือก็คือ ส่งเป็นข้อความธรรมดาได้เลย โดยเบื้องหลังมันจะแอบใช้ WebSocket อีกทีหนึ่ง (สบายแฮะ)

จากโค้ดตัวอย่างดังกล่าว เวลาส่งข้อความและรับข้อความ ก็สามารถทำได้ดังนี้

- การส่งข้อความไปให้อีกฝั่งหนึ่ง จะใช้เมธอด emit()
- ส่วนอีกฝั่งเมื่อรับข้อความ ก็จะใช้เมธอด on()

คอนเซปต์คล้าย ๆ
Emitter pattern

แต่ทั้งนี้ข้อความที่ส่งหากัน คุณจะต้องระบุชื่อเหตุการณ์เป็นอะไรก็ได้ด้วยนะ หรืออาจมองว่าเป็นการติดฉลาก เข้าไปได้ ดังตัวอย่าง



ขณะเดียวกันเหตุการณ์ "serverChat" ก็จะมีหลักเกณฑ์ทำงานเดียวกัน แต่มันจะทำงานกลับด้านกัน ...ก็คือ ฝั่งเซิร์ฟเวอร์จะเป็นคนส่งข้อความมาหาไคลเอนต์เอง

ขั้นตอนที่ 4

จากขั้นตอนที่ 1 – 3 ผมจะรวบยอดเขียนเป็นโค้ดในฝั่งไคลเอนต์ (HTML) กับฝั่งเซิร์ฟเวอร์ ดังนี้

```
<!-- ไฟล์ index.html -->
<!DOCTYPE html>
<html lang="en">
<head> <meta charset="utf-8">
      <title>Socket.IO example</title>
      <script src="http://localhost:8080/socket.io/socket.io.js"></script>
</head>
<body>
  <h1><div id="div1"/> </h1>
  <script>
    var div = document.querySelector('#div1') ;
    var socket = io.connect("http://localhost:8080"); // ติดต่อกับเซิร์ฟเวอร์ ด้วยโปรโตคอล WebSocket
    socket.emit('clientChat', 'Hi server');           // ส่งข้อความไปให้เซิร์ฟเวอร์
    socket.on('serverChat', function(content) {       // รับข้อความมาจากเซิร์ฟเวอร์
      div.innerHTML = content; // นำข้อความที่ได้รับมาแทรกลงในแท็ก <div id="div1"/>
    });
  </script>
</body>
</html>
```

ต้องระบุว่าจะใช้ socket.io ในไฟล์ HTML

```
// ไฟล์ server.js
var io = require('socket.io').listen(8080); // จะจำลองเซิร์ฟเวอร์ และเปิดพอร์ต 8080 ค้างไว้ (เป็น WebSocket)
io.sockets.on('connection', function (socket) { // เมื่อหน้าเว็บติดต่อเข้ามาครั้งแรก คอลแบ็กก็就会被เรียกให้ทำงาน
  socket.on('clientChat', function(content) { // รับข้อความมาจากหน้าเว็บ
    console.log(content);
  });
  socket.emit('serverChat', "I'm OK"); // ส่งข้อความไปให้หน้าเว็บ
});
console.log('Server running at http://localhost:8080/');
```

ไฟล์ทั้งสองตัวที่ยกมาให้ดู (index.html กับ server.js) จะอยู่ในโปรเจกต์ที่มีโครงสร้างดังนี้

```
C:\websocket
```

```
|-- node_modules\...
```

```
|-- package.json
```

```
|-- index.html
```

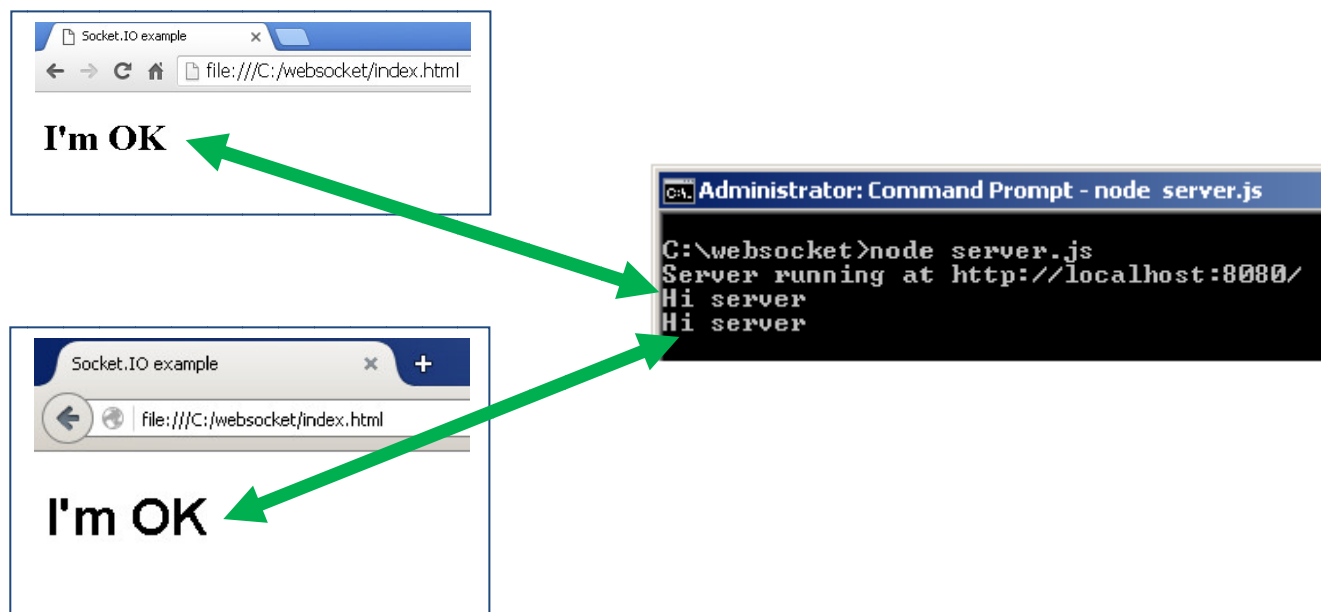
```
|-- server.js
```

เมื่อพิมพ์คำสั่ง “node server.js” ก็จะได้แสดงผลดังต่อไปนี้

```
C:\websocket>node server.js
```

```
Server running at http://localhost:8080/
```

หลังจากนั้นให้ดับเบิลคลิกไฟล์ [C:\websocket\index.html](file:///C:/websocket/index.html) เพื่อเปิดมันขึ้นมา ซึ่งคุณจะได้เห็นหน้าจอบริบทกับหน้าจอบริบทในฝั่งเซิร์ฟเวอร์ ดังนี้



ในภาพจะเป็นการเปิดไฟล์ index.html จำนวนถึง 2 ครั้ง ซึ่งจะหมายถึง ...มีไคลเอนต์สองตัวติดต่อไปยังเซิร์ฟเวอร์ เพื่อแลกเปลี่ยนข้อความ ได้แก่ “Hi server” กับ “I’m OK” ซึ่งกันและกัน

แต่ถ้าเรากรอก URL เป็น <http://localhost:8080/> บนเว็บเบราว์เซอร์ แทนการดับเบิลคลิกไฟล์ จะทำไม่ได้ นะครับ เพราะเซิร์ฟเวอร์มันเปิดพอร์ต 8080 เป็นโปรโตคอล WebSocket ...แต่เราก็มีวิธีแก้ดังหน้าถัดไป

ตัวอย่างโค้ดต่อไปนี้จะนำไฟล์ server.js มาแก้ไขเสียใหม่ ด้วยการผสมผสานวิธีใช้งานมอดูล Express ร่วมกับ socket.io และเพื่อไม่ให้เป็นการเสียเวลา ก็ให้คุณดูโค้ดดีกว่าจะได้เข้าใจไปเลย ดังตัวอย่าง

```
// ไฟล์ server.js

var fs = require('fs');    // มอดูล fs

var app = require('express');
var server = require('http').createServer(app);
var io = require('socket.io')(server);
server.listen(8080);      // จะจำลองเซิร์ฟเวอร์ ด้วยการเปิดพอร์ต 8080

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

    fs.readFile(__dirname + '/index.html',    // อ่านไฟล์ index.html

        function(err, data) {

            if (err) {

                res.writeHead(500);

                return res.end('Error loading index.html');

            }

            res.writeHead(200);
            res.end(data);

        } // สิ้นสุดคอลแบ็ค

    ); // สิ้นสุด readFile()

});

io.sockets.on('connection', function (socket) {

    socket.on('clientChat', function(content) {

        console.log(content);

    });

    socket.emit('serverChat', "I'm OK");

});

console.log('Server running at http://localhost:8080/');
```

เคล็ดกระบวนท่าผสมรวม Express เข้ากับ socket.io ให้กลายเป็นหนึ่ง

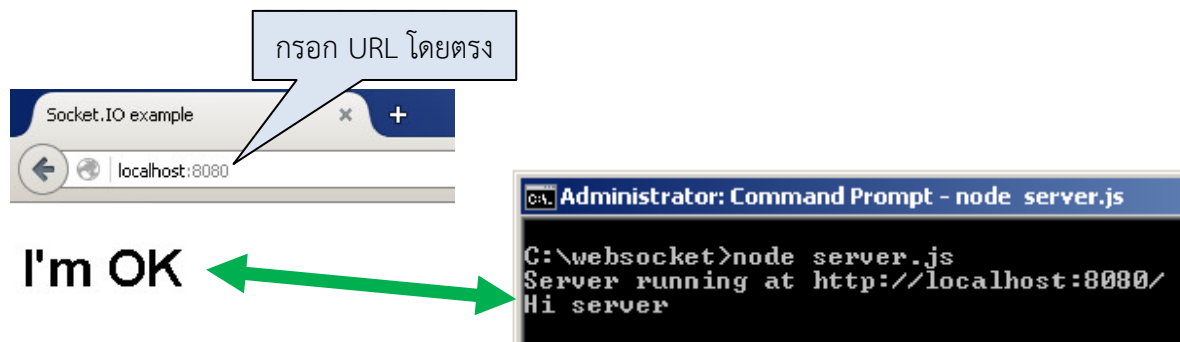
อ่านเนื้อหาในไฟล์ index.html แล้วส่ง data (Buffer) กลับออกไปหาไคลเอนต์

โค้ดเหมือนตัวอย่างที่แล้วเป๊ะ

ในตัวอย่างนี้คอนเซ็ปท์ก็คือ จะเปิดพอร์ต 8080 ค้างเอาไว้ และแทนที่คุณจะเปิดไฟล์ index.html (ไฟล์เดิม) ก็เปลี่ยนไปเปิดเว็บเบราว์เซอร์ แล้วกรอก URL เป็น <http://localhost:8080/> ไม่เพียงเท่านั้นพอร์ตนี้ยังรองรับโปรโตคอล WebSocket ได้อีกด้วย

แต่ทว่าโค้ดใน server.js มันซับซ้อนอยู่พอควร เพราะว่า...

เมื่อเว็บเบราว์เซอร์ส่งรีควีสต์มายังเซิร์ฟเวอร์ ใน server.js จะต้องอ่านไฟล์ index.html แล้วส่งกลับไปยังหน้าเว็บ ...หลังจากนั้นหน้าเว็บจะติดต่อมาหาเซิร์ฟเวอร์ใหม่อีกครั้งด้วย WebSocket โดยผลการทำงานก็จะเป็นดังภาพข้างล่าง



หมายเหตุ

สำหรับรายละเอียดของ socket.io มากกว่านี้ ถ้าสนใจก็อ่านเพิ่มเติมได้ที่

<http://socket.io/>

ติดต่อบานข้อมูล MySQL

ในบทนี้ผมจะพาเที่ยวชม วิธีใช้ Node.js ติดต่อบานข้อมูลเป็น MySQL ...คือถ้าไม่พาคุณมาลองทำ มันเหมือนกินอาหารไม่ใส่เครื่องปรุง เหมือนขาดอะไรไปจริง ๆ นะ เพราะการติดต่อบานข้อมูล เป็นเรื่องปกติที่นักพัฒนาซอฟต์แวร์ มักจะเจอในชีวิตจริง



...เอาละแต่ก่อนอื่นเราต้องติดตั้งมอดูล mysql ขึ้นมาก่อน ด้วยคำสั่ง

```
npm install mysql --save
```

*** สำหรับวิธีติดตั้ง MySQL ก็ให้ไปดูวิธีติดตั้งได้ที่ www.patanasongsivilai.com/itebook_form.html

โค้ดต่อไปนี้จะใช้ติดต่อบานข้อมูลเป็น MySQL

```
var mysql = require('mysql'); // โหลดมอดูล mysql
var connection = mysql.createConnection({
  host : 'localhost',
  user : 'me',
  password : 'secret',
  database : 'my_db'
});
```

ระบุค่าคอนฟิกที่ต้องใช้ติดต่อบานข้อมูล
ขึ้นอยู่กับว่าเราตั้งค่าเป็นอะไร

โค้ดข้างบนคิดว่าคุณน่าจะเข้าใจไม่ยาก ขอแค่เราระบุค่าคอนฟิก (Configuration) ที่เอาไว้ติดต่อบานข้อมูลให้ถูกต้อง หลังจากนั้นเราก็สามารถใช้คำสั่ง SQL ติดต่อบานข้อมูลได้ตามปกติ โดยมีโครงสร้างการเขียนโค้ดหน้าตาประมาณนี้ (ง่าย ๆ ไม่ยากครับ)

```
connection.connect(); // เชื่อมต่อบานข้อมูล
connection.query('SELECT 1 + 1 AS example'); // เขียนคำสั่ง SQL เข้าไป
connection.end(); // ยกเลิกการติดต่อบานข้อมูล
```

โค้ดในหน้าก่อนหน้านี้ เราสามารถระบุคอลแบ็คได้ด้วย ซึ่งถ้านำโค้ดที่กล่าวมาทั้งหมด มาเขียนแบบเต็ม ๆ ยศ ก็จะได้ดังตัวอย่างต่อไปนี้

```

var mysql = require('mysql');
var connection = mysql.createConnection({
  host : 'localhost',
  user : 'root',
  password : '123456'
  //,database : 'nodedb'
});

connection.connect(function(err) {
  if (err) {
    console.error('Error connecting:', err.stack);
    return;
  }

  console.log('Connected as id', connection.threadId);
});

connection.query('SELECT 1 + 1 AS example', function(err, rows, fields) {
  if (err) throw err;
  console.log('The example is:', rows[0].example);
  // สิ้นสุดคอลแบ็ค
});

connection.end(function(err) {
  console.log('The connection is terminated now');
});

console.log('Test MySQL');

```

แล้วแต่ค่าคอนฟิก

ถ้าเรียกเมธอด `connection.query()` แล้ว ตัวเมธอด `connection.connect()` ก็จะถูกเรียกให้ทำงานโดยอัตโนมัติ

สมมติว่าโค้ดชุดนี้ บันทึกไว้เป็นไฟล์ชื่อ “testdb.js” โดยจะมีโครงสร้างโฟลเดอร์ประมาณนี้

C:\database

|-- node_modules\...

|-- package.json

|-- testdb.js

เมื่อผมพิมพ์คำสั่งเป็น “node testdb.js” ก็จะได้ผลลัพธ์ดังต่อไปนี้

```
c:\database>node testdb.js
Test MySQL
Connected as id 2
The example is: 2
The connection is terminated now
```

มีข้อสังเกตนอกจาก connection.end() จะใช้สิ้นสุดการเชื่อมต่อฐานข้อมูลได้แล้ว มันก็ยังสามารถใช้เมธอด...

```
connection.destroy(); // ยกเลิกการติดต่อฐานข้อมูลได้เช่นกัน
```

เมธอดนี้สามารถยกเลิกการติดต่อฐานข้อมูลได้เหมือนกัน แต่มันจะยกเลิกการติดต่อได้ทันทีในระดับ socket อีกทั้งยังจะไม่มีคอลแบ็กให้เรียกทำงานอีกด้วย

ลองสร้างฐานข้อมูล

ในหัวข้อก่อนหน้านี้ เป็นแค่การลองเล่นมอดูล mysql อย่างง่ายเท่านั้น แต่ในตัวอย่างต่อไปนี้ ผมจะลองสร้างฐานข้อมูลชื่อ “nodedb” ซึ่งจะมีโค้ดประมาณนี้

```
connection.query('CREATE DATABASE nodedb');
```

ต่อมาผมจะสร้างตารางชื่อ “test_table” ก็เขียนโค้ดหน้าต่างดังนี้

test_table	
id	content
1	Node.js
2	Easy

```
connection.query(`CREATE TABLE test_table
(id INT(11) AUTO_INCREMENT,
content VARCHAR(255),
PRIMARY KEY(id))`
);
connection.query('INSERT INTO test_table (content) VALUES ("Node.js")');
connection.query('INSERT INTO test_table (content) VALUES ("Easy")');
```

ให้ใช้เทมเพลตสตริง
เพราะจะเขียนข้อความหลายบรรทัด
ได้ง่ายกว่าสตริงธรรมดา

จากโค้ดที่อธิบายมาทั้งหมด ผมจะเขียนไว้อยู่ในไฟล์ “createTable.js” โดยมีหน้าตาดังต่อไปนี้

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

var connection = mysql.createConnection({
  host : 'localhost',
  user : 'root',
  password : '123456'
  //,database : 'nodedb'
});

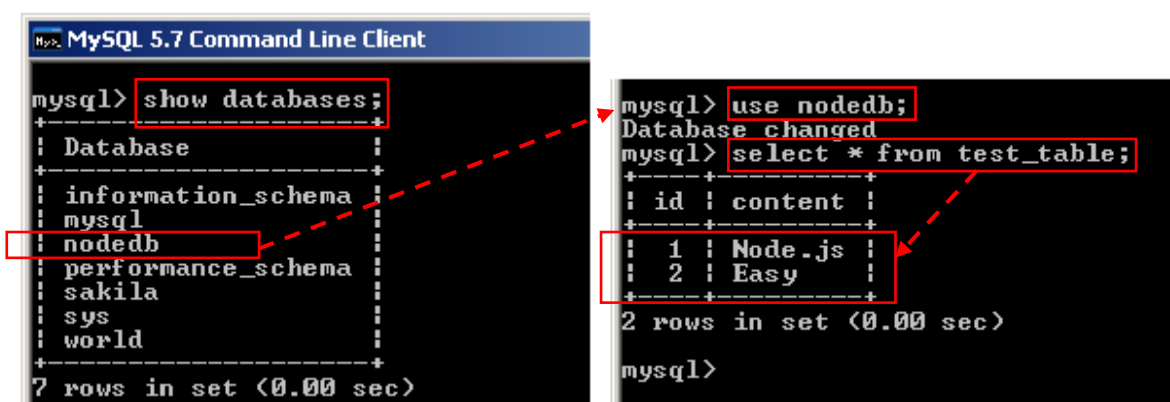
// connection.query('DROP DATABASE nodedb'); // เมื่อต้องการลบฐานข้อมูล
connection.query('CREATE DATABASE nodedb'); // สร้างฐานข้อมูลชื่อ nodedb

connection.query('USE nodedb');

//connection.query('DROP TABLE test_table'); // เมื่อต้องการลบตาราง
connection.query('CREATE TABLE test_table
  (id INT(11) AUTO_INCREMENT,
  content VARCHAR(255),
  PRIMARY KEY(id))'
);

connection.query('INSERT INTO test_table (content) VALUES ("Node.js");');
connection.query('INSERT INTO test_table (content) VALUES ("Easy");');
connection.end();
```

ถ้าผมรันคำสั่ง “node createTable.js” บนคอมพิวเตอร์เครื่องเรียบร้อยแล้ว เมื่อไปเปิดคอมพิวเตอร์ของ MySQL ก็จะมีตาราง test_table อยู่ในฐานข้อมูล nodedb ตามรูปข้างล่าง



ต่อมาผมจะเขียนโค้ด เพื่อคิวรีข้อมูลในตาราง test_table มาแสดงผล ซึ่งจะเห็นในหน้าถัดไป ...ขอรับ

```

var mysql = require('mysql');
var connection = mysql.createConnection({
  host : 'localhost',
  user : 'root',
  password : '123456',
  ,database : 'nodedb'
});

connection.query('SELECT * FROM test_table', function(err, rows, fields) {
  if (err) throw err;
  for(let r of rows) {
    console.log('id = ${r.id} ,content = ${r.content}' ); // เข้าถึงแต่ละคอมลัมน์
  } // สิ้นสุดประโยค for
});

connection.end();
console.log('Test MySQL');

```

เพิ่มค่าเข้ามา

คิวรี

เมื่อคิวรีทำงานเสร็จแล้ว
คอลแบ็คก็就会被เรียกให้ทำงาน

// เข้าถึงแต่ละคอมลัมน์

// สิ้นสุดประโยค for

ในตัวอย่างนี้พารามิเตอร์ `rows` ของคอลแบ็ค มันคืออาร์เรย์ (Array) ที่เก็บสมาชิกเป็นอ็อบเจกต์ (แถวในตาราง `test_table`) โดยอ็อบเจกต์ดังกล่าวจะมีชื่อคือ `rows` (ชื่อพรีอเพอร์เตอร์) ตรงกับชื่อคอลัมน์ในตาราง ตามรูป

test_table	
id	content
1	Node.js
2	Easy

`rows[0] = { id=1, content='Node.js'}`

`rows[1] = { id=2, content='Easy'}`

ส่วนประโยค `for(let r of rows){...}` ในโค้ด มันเป็นประโยค `for ...of` ใน ES6 ซึ่งจะไวนลูปเข้าถึงสมาชิกทุกตัวของอาร์เรย์ โดยแต่ละรอบที่มันเข้าถึง จะนำสมาชิก (เป็นอ็อบเจกต์) มากำหนดค่าให้กับตัวแปร `r`

และถ้าสมมติโค้ดนี้บันทึกเป็นไฟล์ `"query.js"` เมื่อรันคำสั่งเป็น `"node --use-strict query.js"` จะได้ดังนี้

```
c:\database>node --use-strict query.js
```

```
Test MySQL
```

```
id = 1 ,content = Node.js
```

```
id = 2 ,content = Easy
```

สำหรับในตัวอย่างที่กล่าวมาในหัวข้อนี้ทั้งหมด ก็จะมีโครงสร้างโฟลเดอร์ประมาณนี้ครับ

```
C:\database
```

```
|-- node_modules\...
```

```
|-- package.json
```

```
|-- testdb.js
```

```
|-- createTable.js
```

```
|-- query.js
```

ส่งท้าย

สำหรับเล่มนี้ผมแค่แสดงวิธีประยุกต์ใช้งาน Node.js เบื้องต้นเท่านั้น เพราะจริง ๆ แล้ว มันยังมีลูกเล่นอื่นอีก เยอะเลย ...แต่ก็หวังว่าน่าจะเพียงพอ ให้นำมันไปใช้ต่อยอดงานของคุณในอนาคตได้

อ้างอิง

หนังสือ

[1] Pedro Teixeira, “Professional Node.js Building Javascript Based Scalable Software”, John Wiley & Sons, Inc., 2013.

เอกสารจากเว็บไซต์ เข้าถึงล่าสุด 9 ธ.ค. 2559

- [1] <https://en.wikipedia.org/wiki/Node.js>
- [2] <https://nodejs.org/en/>
- [3] <http://expressjs.com/>
- [4] <http://getbootstrap.com/>
- [5] <http://ejs.co/>
- [6] <http://expressjs.com/en/starter/generator.html>
- [7] <http://kikobeats.com/synchronously-asynchronous/>
- [8] <http://jade-lang.com/>
- [9] <http://socket.io/>
- [10] <https://github.com/felixge/node-mysql>