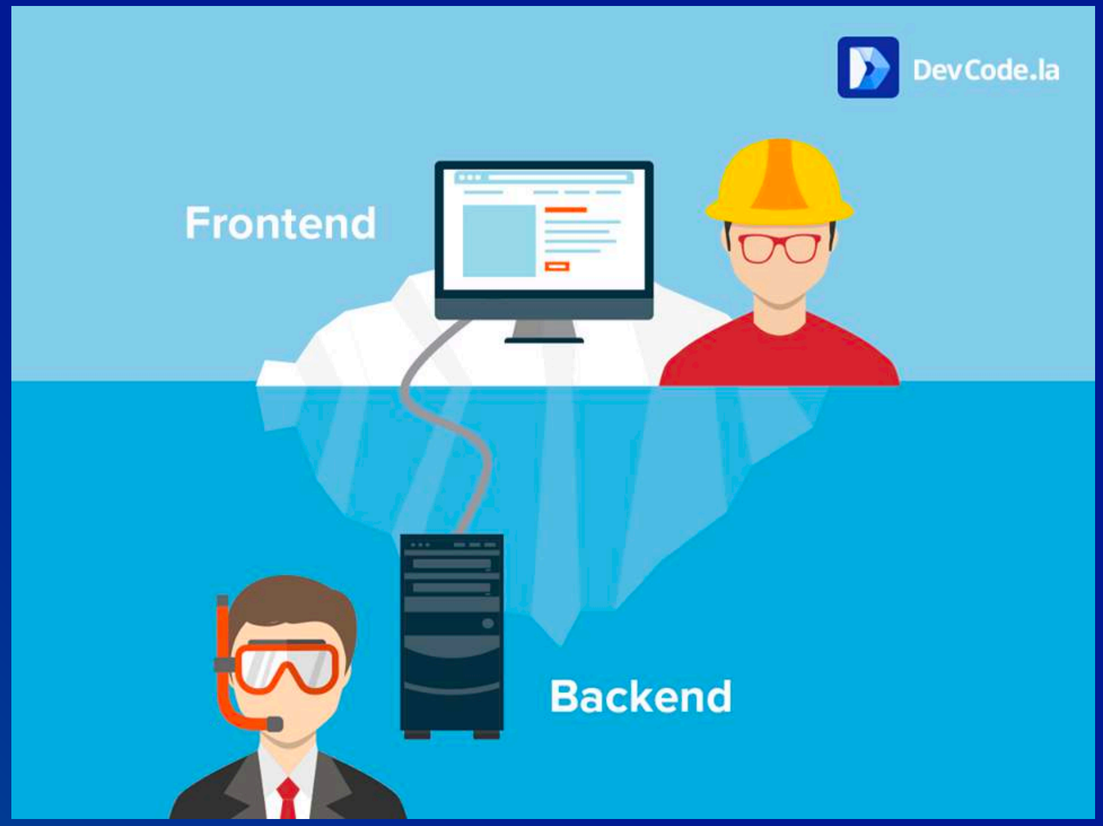
07. Introduction to Backend



Ric Huang / NTUEE

(EE 3035) Web Programming

Recap: Backend Server

- 還記得這幾次的上課與作業我們都是使用 "create-react-app" 這個工具，自動產生整個 React App 的架構，然後利用 "yarn/npm start" 來執行 app.
- 但你有沒有好奇過為什麼我們執行 app 是去打開 "localhost:3000", 而不是用 browser 打開 "index.html"?
- 你有沒有發現只要你更新任何程式碼，都會自動 trigger "localhost:3000" 的更新？

Your First Backend Server

- Basically, 當你執行 "yarn/npm start" 之後，你就啟動了一個 "node.js server", 跑在你的電腦 (i.e. localhost) 上，並且透過 port 3000 來跟外界溝通。

- 而伺服器(server)上面的後端服務(backend service)會一直傾聽相關的需求，例如：發現程式碼有更新，就啟動重新編譯的動作，並刷新服務程式

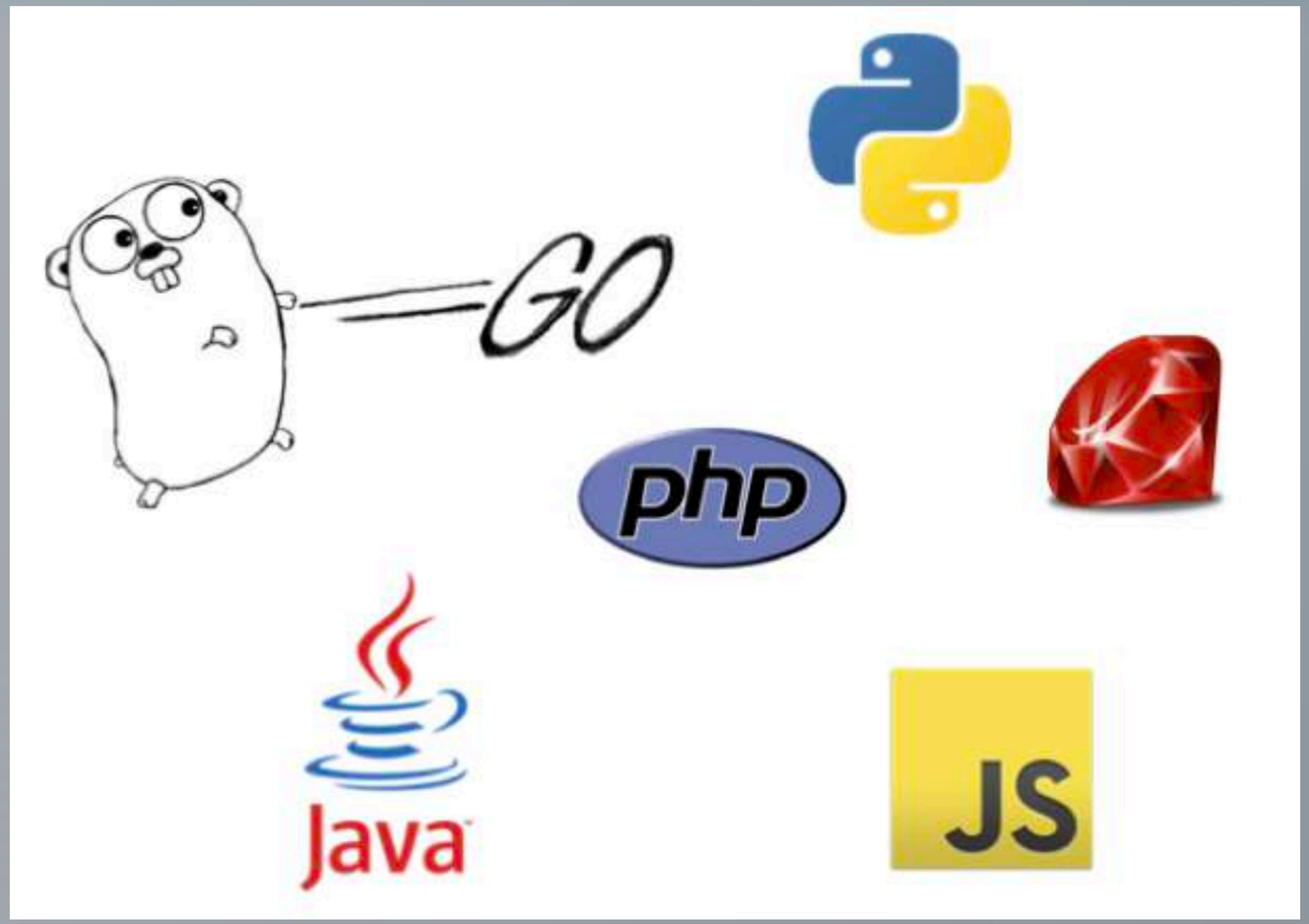
Toward a complete web service

- 一個完整的網路服務應用通常要有一個後台(backend)，並且有資料庫(database)來儲存用戶或者是服務流程中的相關資料
- 而上述的服務通常會放在一個有固定 IP 的伺服器，並且去申請 domain name, 讓客戶端(client)可以透過 URL(e.g. 網址) 來取得服務

Localhost as a local web server

- 開發者在開發網路服務的時候，為了測試方便，會先使用自己的電腦來作為後端的伺服器，這就是為什麼會有在開發的時候 backend 與 frontend 都在同一台機器(i.e. localhost)的現象。等到開發到一定程度之後再來 deploy 到雲端 or 機房的機器上面。

基本上任何可以跑在伺服器(電腦)的程式語言
都可以用來實現後端的 server

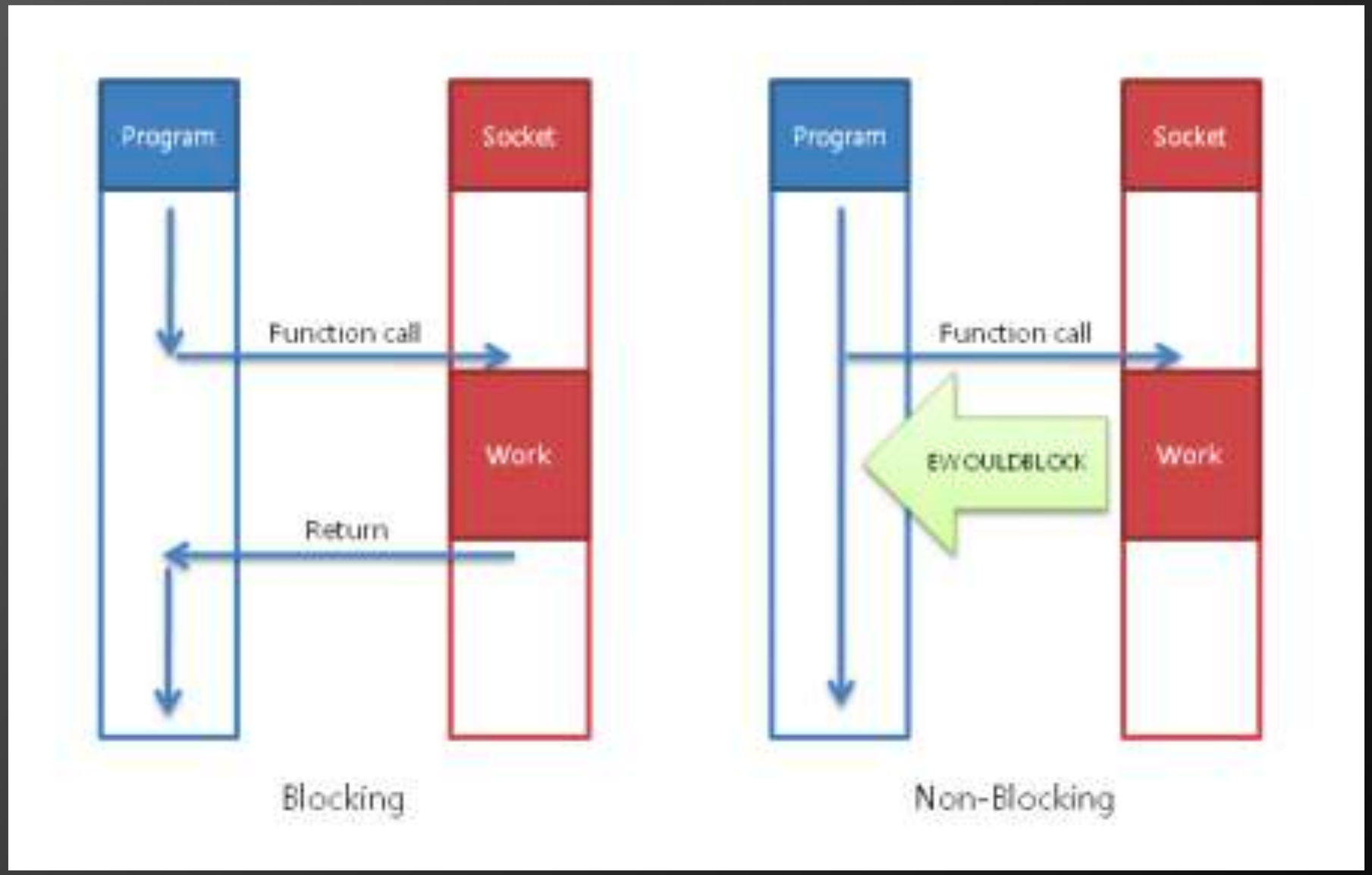


在我們開始介紹後端技術、
框架、以及系統整合之前
我們先介紹一些有關於後端
的基礎知識

Background Knowledges about Backend

- Blocking vs. Non-Blocking
- Synchronous vs. Asynchronous
- Promise, Async/Await
- HTTP
- Middleware

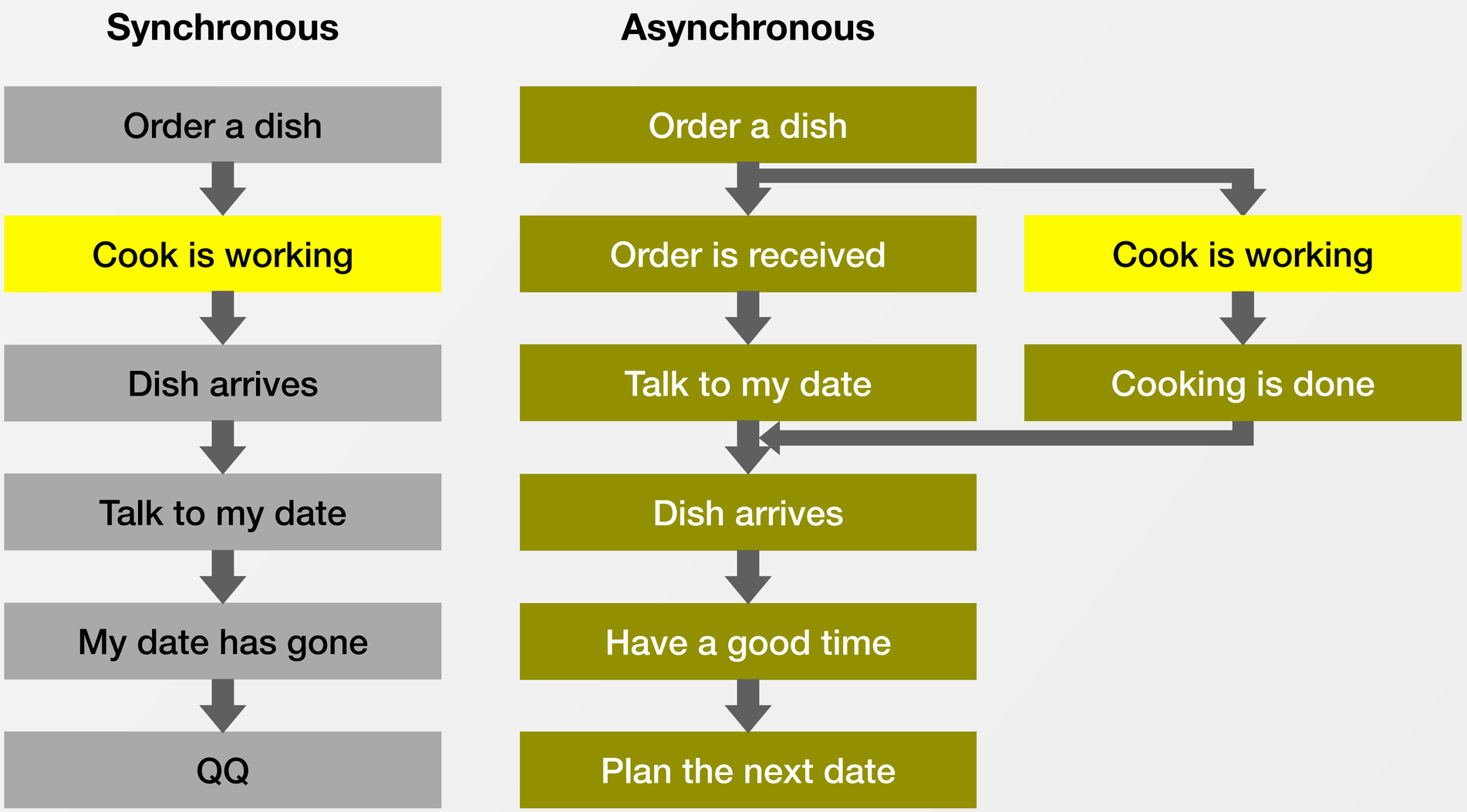
Recall: Blocking vs. Non-Blocking



Blocking vs. Non-Blocking

- 關於 "Blocking or Non-Blocking"，不同的 context 底下其實有不同的意義
- Blocking/Non-Blocking Assignment
  - 大部分的 programming language 的 assignment 都是 blocking 的，但像 Verilog 裡頭描述 combinational block, 就會用 non-blocking assignment (因為是在描述硬體)
- Blocking/Non-Blocking I/O
  - 大部分的電腦系統因為 I/O 相對比較慢，所以採取 non-blocking I/O, 以免影響系統的運算
  - 但一些跟 data dependency 有關的 I/O, 像是 DB access, 就常會使用 blocking I/O

Synchronous or Asynchronous?



Uh? (Non)Blocking = (A)Synchronous?

- 在很多時候，(non)blocking 與 (a) synchronous 常常會被拿來表達同一件事情，而且不會有誤會，也沒有違和感。
- (有此一說)https://stackoverflow.com/questions/2625493/asynchronous-vs-non-blocking
  "synchronous / asynchronous is to describe the relation between two modules; blocking / non-blocking is to describe the situation of one module."
  - 畢竟 "synchronous" (同步) implies 兩個(含)以上的事物對齊某個時鐘/時序，像是「同步電路」一樣必須等到 clock 口令才能做下一個動作，而 "asynchronous" (非同步) 則沒有這樣的限制，完成工作的模組可以繼續下一件事

所以 JavaScript 是 synchronous 還是 asynchronous?

- Try this...

```
document.addEventListener('click', clickHandler);
console.log("started execution");
waitThreeSeconds();
console.log("finished execution");
function waitThreeSeconds(){
  console.log("Wait 3 seconds...");
  var ms = 3000 + new Date().getTime();
  while(new Date() < ms) {}
  console.log("finished function");
}
function clickHandler() {
  console.log("click event!");
}
```

- Click when start executing. See how message is printed

所以 JavaScript 是 synchronous 還是 asynchronous?

```
document.addEventListener
          ('click', clickHandler);
console.log("started execution");
waitThreeSeconds();
console.log("finished execution");

function waitThreeSeconds(){
  console.log("Wait 3 seconds...");
  var ms = 3000 + new Date().getTime();
  while(new Date() < ms) {}
  console.log("finished function");
}

function clickHandler() {
  console.log("click event!");
}
```

Output

- started execution
- Wait 3 seconds...
- finished function
- finished execution

1

undefined 2

- click event! 3

1. JS assignment is blocking
2. Code finishes and returns
3. JS event is asynchronous

Callback Functions

- Asynchronous I/O 的好處是 caller 的 execution 不會因為呼叫了 I/O function 而被停住，但問題是：我怎麼知道 asynchronous function 已經執行完了？
- "Polling" 是一個解決的辦法，但不但沒有效率且可能沒有即時性
- 在 JavaScript 中常見的做法是「傳一個 callback 給 asynchronous function」，像這樣：

```
const result
= someAsyncFunction(url, callbackFunction);
```

Callback Hell

- This kind of callback mechanism is notorious for the potential "callback hell" situation
  - A requests B; while waiting for B, do something
    - B requests C; while waiting for C, do something
      - C requests D...
    - Process C's result. Determine success or fail.
  - Process B's result. Determine success or fail.
- Let along error handling. => Very difficult to debug!

像這樣...

callbackhell.js

```
1
2       var floppy = require('floppy');
3
4       floppy.load('disk1', function (data1) {
5           floppy.prompt('Please insert disk 2', function() {
6               floppy.load('disk2', function (data2) {
7                   floppy.prompt('Please insert disk 3', function() {
8                       floppy.load('disk3', function (data3) {
9                           floppy.prompt('Please insert disk 4', function() {
10                              floppy.load('disk4', function (data4) {
11                                  floppy.prompt('Please insert disk 5', function() {
12                                      floppy.load('disk5', function (data5) {
13                                          floppy.prompt('Please insert disk 6', function() {
14                                              floppy.load('disk6', function (data6) {
15                                                  //if node.js would have existed in 1995
16                                              });
17                                          });
18                                      });
19                                  });
20                              });
21                          });
22                      });
23                  });
24              });
25          });
26      });
27
```

To prevent "callback hell" (ref)

1. Keep your code shallow
2. Modularize your code
3. Handle every single error

Or using better coding mechanism

Introducing Promise

- 傳統用 callback 來作為 asynchronous functions 的回應方法很容易讓 code 變得很混亂、破碎，且不容易將不同非同步事件之間的關係描述得很乾淨

- Promise 就是為了讓在 JavaScript 裡頭的 asynchronous 可以 handle 的比較乾淨

Promise 物件

- 定義好一個 asynchronous function 執行之後成功或是失敗的狀態處理
- 使用情境：將 Promise 物件的 .then(), .catch() 加入此 asynchronous function 應該要被呼叫的地方
- Note: ES6 已經把 promise.js 納入標準

Promise Example (ref)

- 想像一下你是個孩子，你媽承諾(Promise)你下個禮拜會送你一隻新手機(some async event)。
- 現在你並不知道下個禮拜你會不會拿到手機。你媽可能真的買了新手機給你，或者因為你惹她不開心而取消了這個承諾 (resolved, or error)
- 為了把這段人生中小小的事件定義好(所以你可以繼續專心的生活)，你將問了媽媽以後會發生的各種情況寫成一個 JavaScript function:

```
var askMom = function willIGetNewPhone() { … }
askMom();  // execution will be blocked for a week...
```

Promise Example (ref)

- 基本上一個像是 willIGetNewPhone() 的非同步事件會有三種狀態:
  - Pending: 未發生、等待的狀態。到下週前，你還不知道這件事會怎樣。
  - Resolved 完成/履行承諾。你媽真的買了手機給你。
  - Rejected 拒絕承諾。沒收到手機，因為你惹她不開心而取消了這個承諾。
- 我們將把 willIGetNewPhone() 改宣告一個 Promise 物件，來定義上面三種狀態。

Promise Example (ref)

```
var isMomHappy = true;
var willIGetNewPhone
= new Promise((resolve, reject) =>
    (isMomHappy)?{
      var phone = { id: 123 }; // your dream phone
      resolve(phone);
    } : {
      var reason = new Error('Mom is unhappy');
      reject(reason);
    }
  );
```

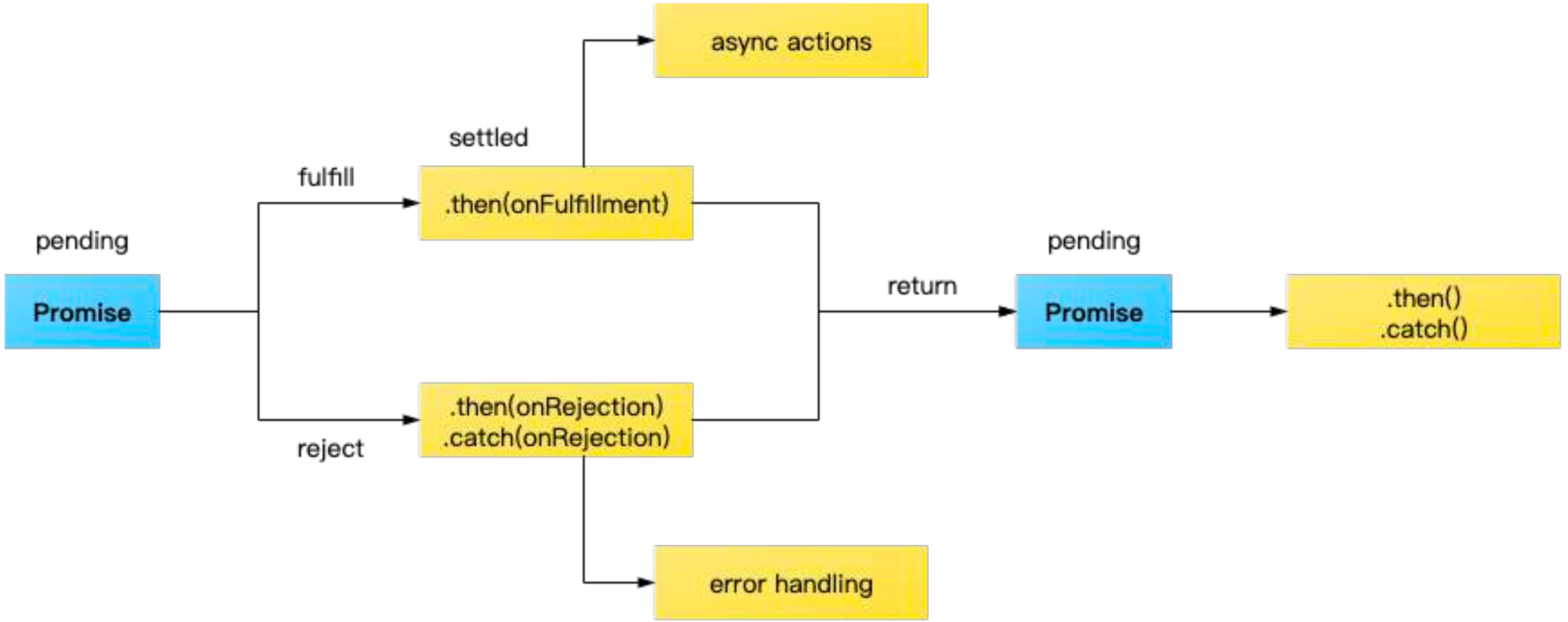
Promise Example (ref)

- 定義 Promise 物件的語法:

```
new Promise(function (resolve, reject) { … });
```

- 宣告一個 Promise 的物件作為一個非同步事件的 agent，而傳入的 executor 接受兩個 callbacks：resolve/reject, 用來作為此非同步事件執行結果出來時，成功/失敗時應該要呼叫的 function

Promise 的處理程序



Putting Things Together

- 把 asynchronous function 包成 Promise 物件，用 .then() 與 .catch() 來處理成功與失敗的結果。

```
const momHappy = (phone) => {...}
const momUnhappy = (reason) => {...}
let willIGetNewPhone = new Promise((resolve, reject) => {
  if (isMomHappy) {
    const phone = getNewPhone();
    resolve(phone);
  } else {
    const reason = "...";
    reject(reason);
  }
});
willIGetNewPhone
.then(momHappy(phone))
.catch(momUnhappy(err));
```

resolve, reject, .then(), .catch()?

- Promise 有許多 APIs, 其中 .then() 可以吃兩個參數：

```
promise.then(successCallback, failureCallback);
```

- resolve/reject 只是參數，不一定要叫這個名字
- "failureCallback" 可省
- .catch(failureCallback) 其實就等於 .then(null, failureCallback)

Chain of Promises

- 假設有一連串的 asynchronous functions 要執行，可以把他們的 Promise 串聯起來:

```
doSomething()
.then(result => doSomethingElse(result))
.then(newResult => doThirdThing(newResult))
.then(finalResult => {...})
.catch(failureCallback);
```

- 其中 doSomething(), doSomethingElse(), doThirdThing() 就是用 Promise 包起來的三個 asynchronous functions

.catch() 後面還是可以接東西

```
new Promise((resolve, reject) => {
    console.log('Initial');
    resolve();
}).then(() => {
    throw new Error('Something failed');
    console.log('Do this');
}).catch(() => {
    console.log('Do that');
}).then(() => {
    console.log('Do this whatever happened before');
});
```

- Output (if no failure) —

```
Initial
Do that
Do this whatever happened before
```

更多 Promise APIs (ref)

- Promise.all(iterable)　　// 都要成功
- Promise.race(iterable)　// 看誰先成功
- Promise.reject(reason)　// 拒絕的 promise
- Promise.resolve(value)　// 成功的 promise

Promise 的出現，雖然解決了
callback hell 的問題，
讓 caller 與 asynchronous
function 之間有個清楚的協定：
成功 / 失敗 後應該做什麼事
但 Promise 的機制實在是不夠直覺...

ES7

ES6

Promise

ES5

async / await

- 語法：

```
async (para1,para2) => {
  // some logic for the async function
  await someStatement (or promise)
  // something after resolve/reject of someStatement
}
```

- async/await 的目的在讓 Promise 概念的使用更直覺
- async 用來修是一個 function, 讓他變 async
- await 把原先非同步結束後才要在 .then()/.catch() 執行的程式碼直接隔開，平行的放到下面
- 基本上 async 匡了一個 top-level async 的區域，然後裡面透過 await 變成是 synchronous

async / await example

```
const startGame = async () => {
  const {
    data: { msg }
  } = await instance.post('/start')

  return msg
}
```

```
const bot = new ConsoleBot();

bot.onEvent(
  async context => {
    if (context.event.text == "qq")
      await context.sendText('Are you OK?');
    else
      await context.sendText('Hello World');
  }
);
```

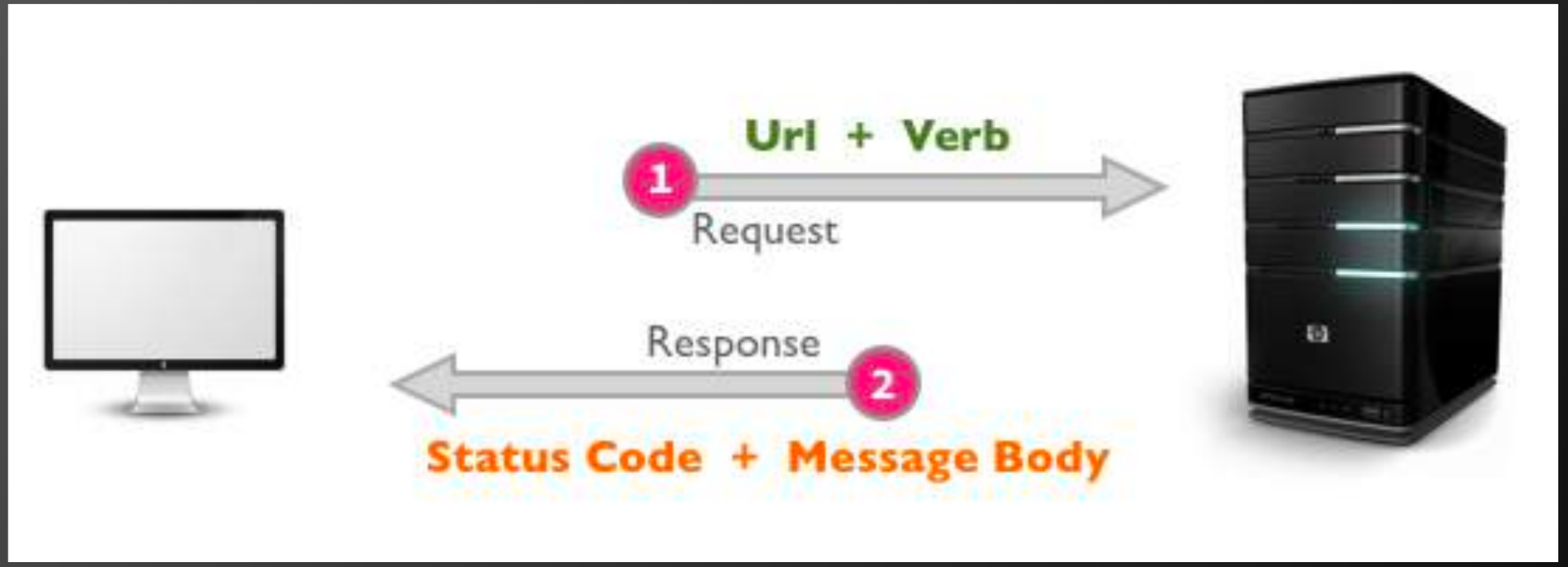
更多的 example 會在今天
講到 backend server 時示範

你有沒有想過...
前端與後端是用什麼管道與方式
來聯絡的呢？

HTTP

- HyperText Transfer Protocol
- HTTP Client 跟 Server 之間進行請求與回應的標準
- Version: 1.0 (1996), 2.0 (2015), 3.0 (2018)

Client Request and Server Response



URL Structure

HTTP Request

- A Request-line
- Zero or more header fields followed by CR/LF
- An empty line (i.e., a line with nothing preceding the CR/LF)
- Optionally a message-body

HTTP Request Example

```
POST /users/123 HTTP/1.1 // Request-line
Host: www.example.com // header fields
Accept-Language: en-us // ..
Connection: Keep-Alive // ..
// empty line
licenseID=string&content=string&/paramsXML=string
// message-body
```

常見的 HTTP Request Methods (ref)

- GET — The GET method requests a representation of the specified resource. Requests using GET should only retrieve data.
- POST — The POST method is used to submit an entity to the specified resource, often causing a change in state or side effects on the server.
- PUT — The PUT method replaces all current representations of the target resource with the request payload.
- DELETE — The DELETE method deletes the specified resource.

其他 HTTP Request Methods (ref)

- HEAD — The HEAD method asks for a response identical to that of a GET request, but without the response body.
- PATCH — The PATCH method is used to apply partial modifications to a resource.

An Analogy of HTTP Request Methods (ref)

- 假設現在我們要點餐，
- 我們必須先知道菜單是甚麼（get），
- 我們會向服務生點餐（post），
- 我們想要取消剛才點的餐點（delete），
- 我們想要重新點一次（put），
- 我們想要加點甜點和飲料（patch）。

HTTP Response

- A Status-line
- Zero or more header fields followed by CR/LF
- An empty line (i.e., a line with nothing preceding the CR/LF)
- Optionally a message-body

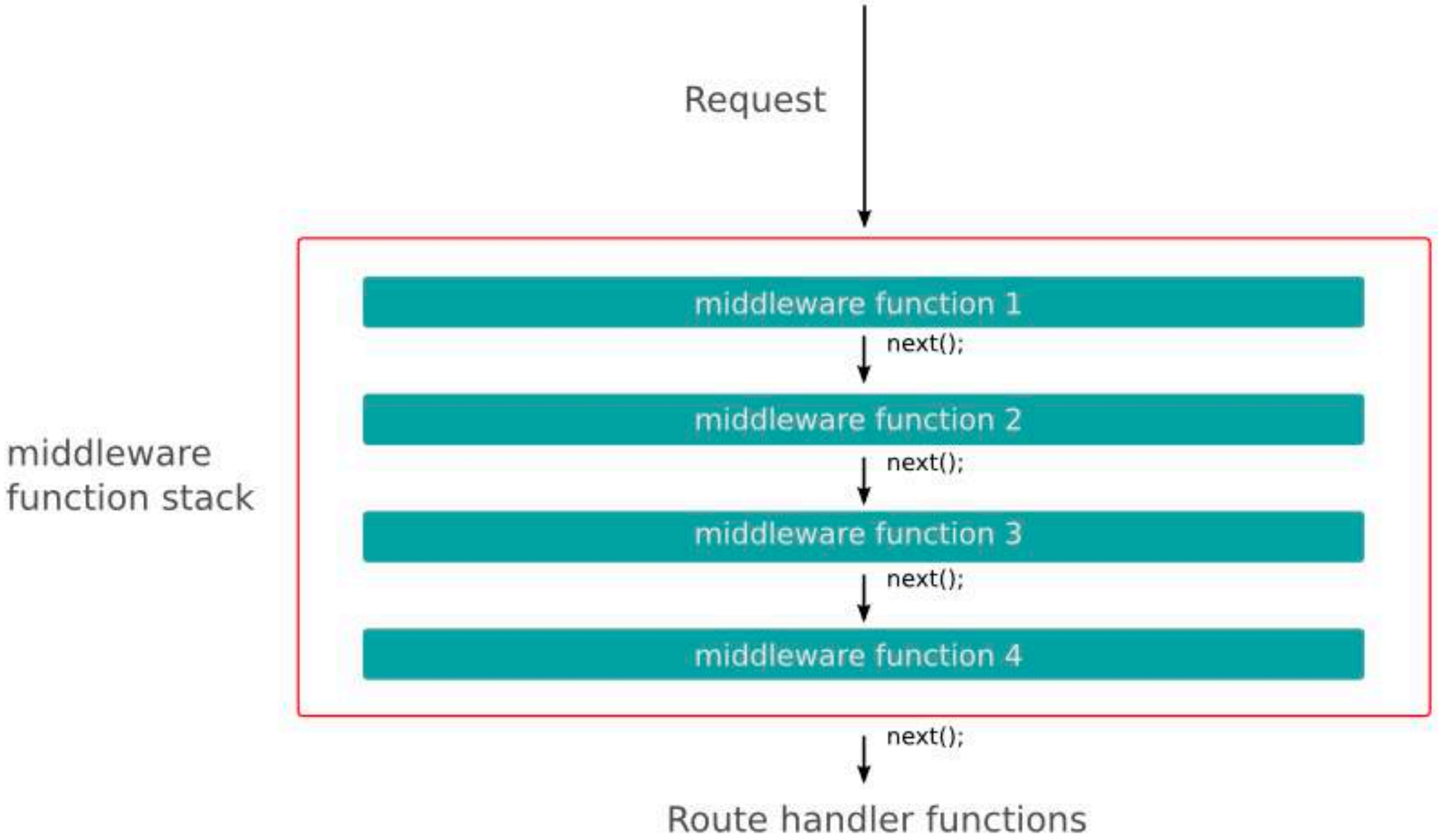
HTTP Response Example

```
HTTP/1.1 200 OK // Status-line
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Connection: keep-alive
Content-Length: 53
Content-Type: text/html; charset=UTF-8
Date: Tue, 11 Oct 2016 16:32:33 GMT
ETag: "53-1476203499000"
Last-Modified: Tue, 11 Oct 2016 16:31:39 GMT
// empty line
<html> // message-body
<body>
<h1>Hello, World!</h1>
</body>
</html>
```

【Middleware】

Middleware functions are functions that have access to the request object (req), the response object (res), and the next middleware function in the application's request-response cycle. The next middleware function is commonly denoted by a variable named next.

Middleware。一個生產線的概念



今天到目前為止，學了什麼？

- Blocking vs. Non-Blocking
- Synchronous vs. Asynchronous
- Promise, Async/Await
- HTTP
- Middleware

我們接下來要介紹目前用
JavaScript 做後端最流行
的框架：Node.js

Introduction to Node.js

- 由 Ryan Dahl 在 2009 開發，目標在實現 "JavaScript Everywhere" 的典範，讓 web development 可以統一在一個語言底下

- 基本上 Node.js 就是一個 JavaScript 的 runtime environment, 讓 JavaScript 可以在 Browser 以外的環境執行，所以 Node.js 可以用來開發以 JS 為基礎的後端程式

Node.js 的核心就是 Google 開源的 Chrome V8 JS 引擎 (in C++)

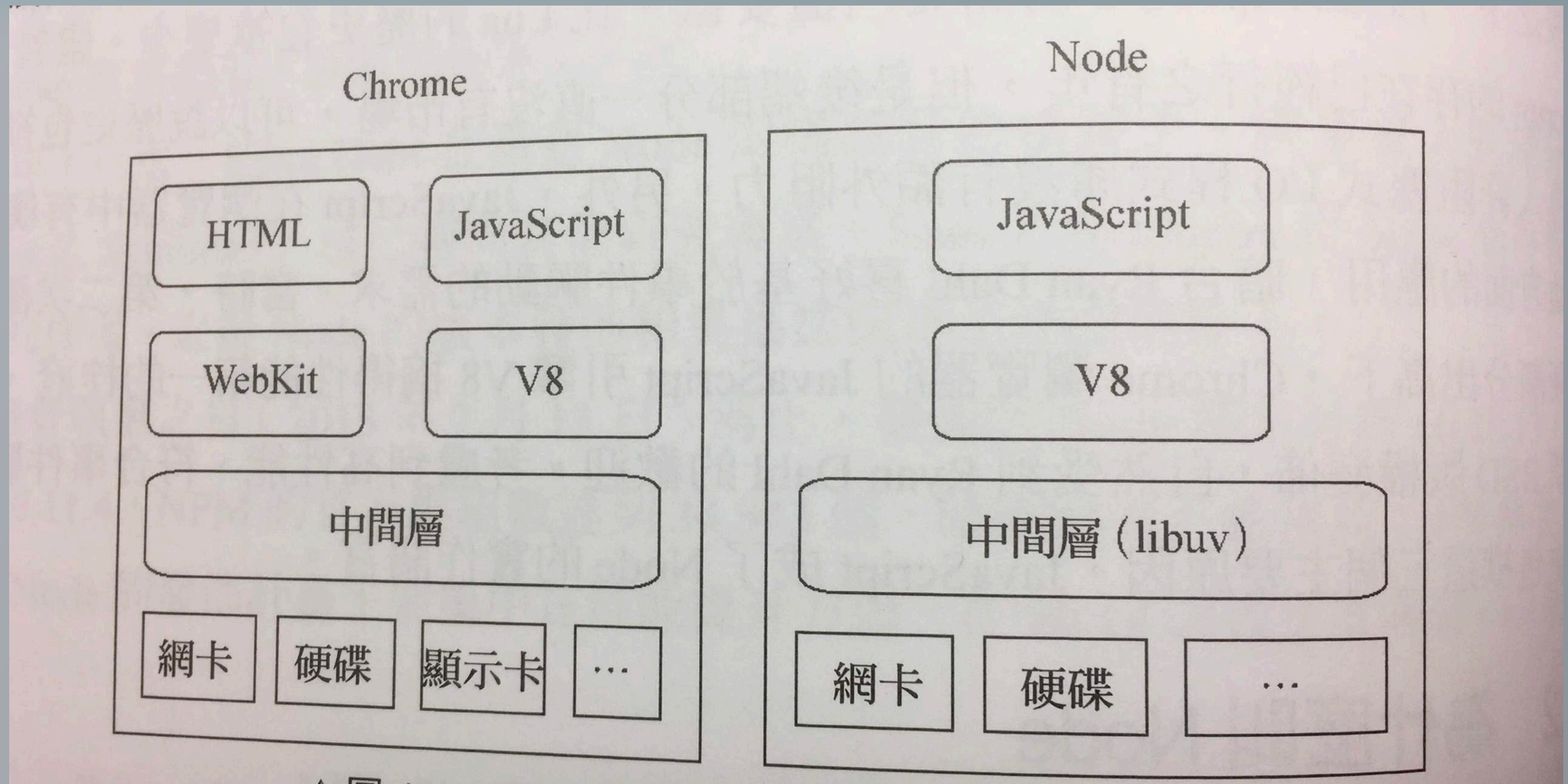


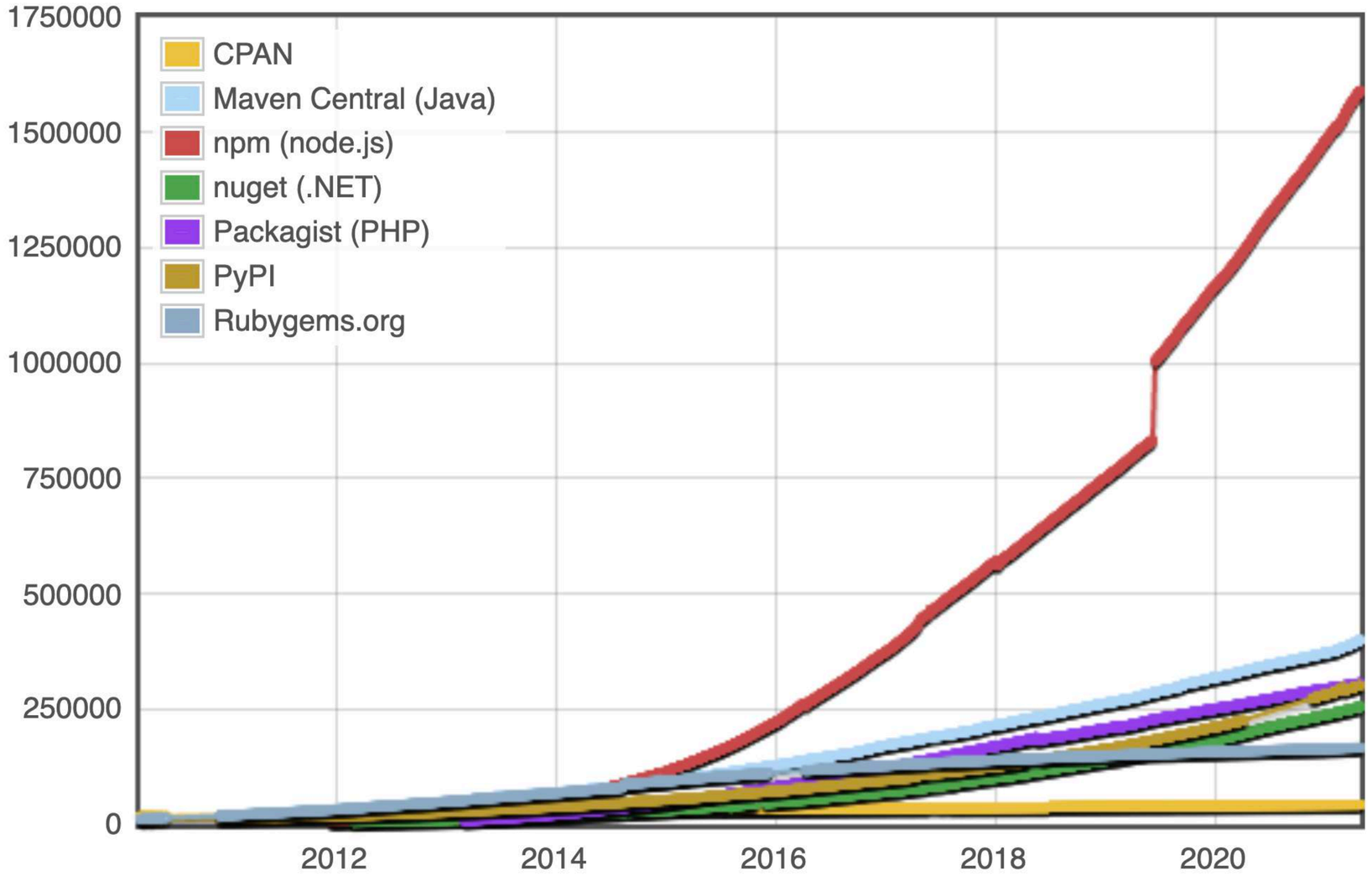
圖 1-1 Chrome 瀏覽器和 Node 的構成組件

Node.js 雖然有 .js, 但它並不是指某個單一的 JS 檔案

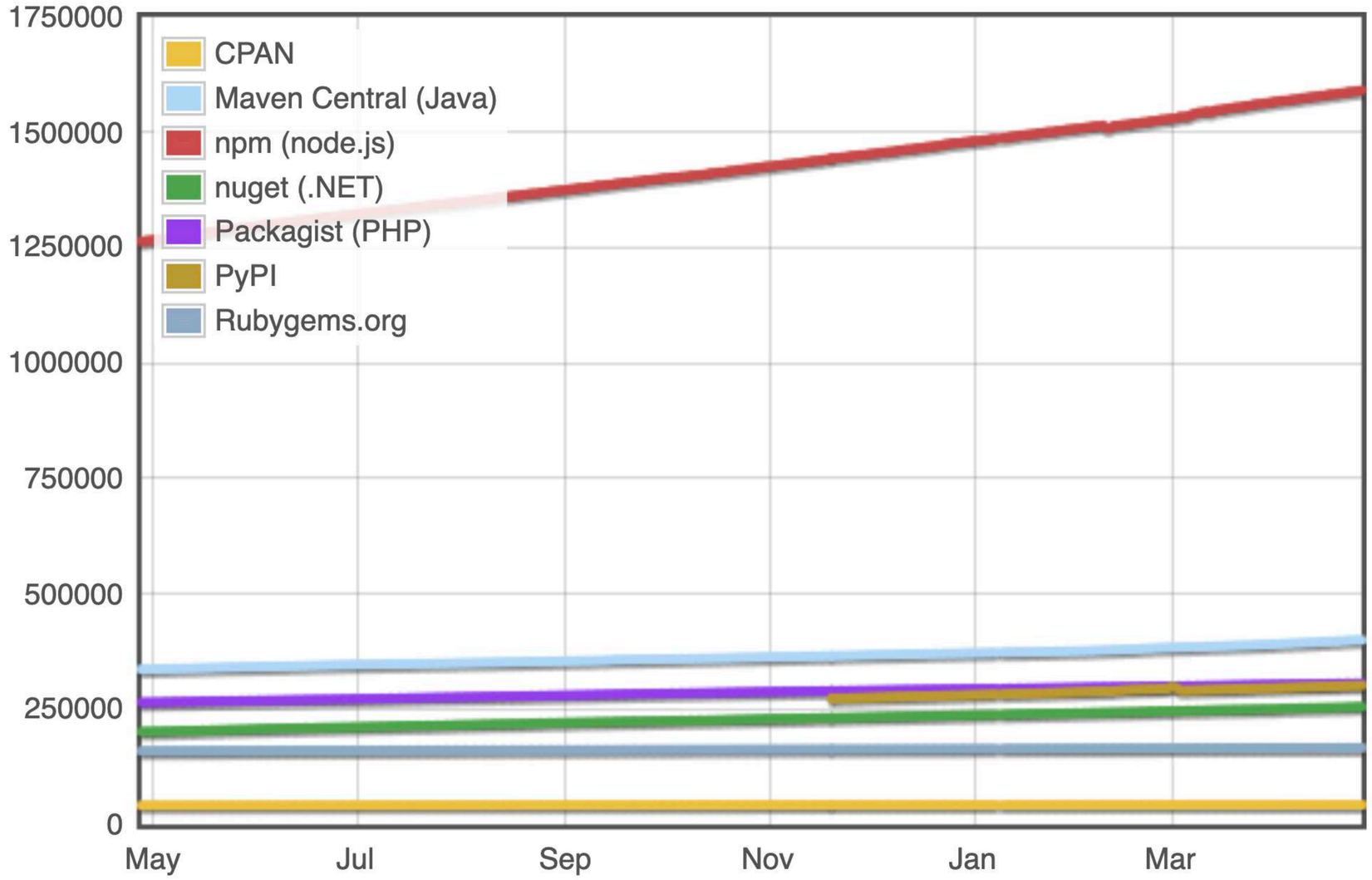
- 事實上，它是個 JS runtime environment, 允許新增各種用 JS 寫的功能模組 (modules)，包含了 file system I/O, networking, binary data (buffers), cryptography functions, data streams, etc.

- 它的各種模組可以透過像是 "npm" (Node Package Management, introduced in 2010) 等工具來管理，而且由於它開源的關係，Node 相關的套件正在以非常驚人的速度在增加

Module Counts



Module Counts



所以，面對現實，如果你要學
Web Programming, 然後只想
focus 在一個語言，那學
JavsScript 準沒錯，而且
Node.js 是必學！

非常活躍的 Node.js Project

Release	Status	Code name	Release date	Active LTS start	Maintenance start	Maintenance end
0.10.x	End-of-Life		2013-03-11	N/A	2015-10-01	2016-10-31
0.12.x	End-of-Life		2015-02-06	N/A	2016-04-01	2016-12-31
4.x	End-of-Life	Argon[68]	2015-09-08	2015-10-01	2017-04-01	2018-04-30
5.x	End-of-Life		2015-10-29	N/A		2016-06-30
6.x	End-of-Life	Boron[68]	2016-04-26	2016-10-18	2018-04-30	2019-04-30
7.x	End-of-Life		2016-10-25	N/A		2017-06-30
8.x	End-of-Life	Carbon[68]	2017-05-30	2017-10-31	2019-01-01[69]	2019-12-31
9.x	End-of-Life		2017-10-01	N/A		2018-06-30
10.x	Maintenance LTS	Dubnium[68]	2018-04-24	2018-10-30	2020-05-19	2021-04-30
11.x	End-of-Life		2018-10-23	N/A		2019-06-01
12.x	Maintenance LTS	Erbium[68]	2019-04-23	2019-10-21	2020-11-30	2022-04-30
13.x	End-of-Life		2019-10-22	N/A	2020-04-01	2020-06-01
14.x	Active LTS	Fermium[68]	2020-04-21	2020-10-27	2021-10-19	2023-04-30
15.x	Maintenance		2020-10-20	N/A	2021-04-01	2021-06-01
16.x	**Current**	Gallium[68]	2021-04-20	2021-10-26	2022-10-18	2024-04-30
17.x	Planned		2021-10-19	N/A	2022-04-01	2022-06-01
18.x	Planned		2022-04-19	2022-10-25	2023-10-18	2025-04-30

Legend: Old version | Older version, still maintained | **Latest version** | Future release

Node.js Versions

- 每六個月有個 Major Release (四月、十月)
- 奇數版本在十月份發行的時候，先前在四月發行的偶數版本就會自動變成 Long Term Support (LTS) 版本，然後會被積極、持續地維護 18 個月，結束後會被延長維護 12 個月，然後就壽終正寢
- 奇數版本沒有 LTS

Node.js is singly-threaded,
event-driven, and
non-blocking I/O

- 因為 Node.js 是 singly-threaded, 然後又讓要花較多時間的 I/O 利用 non-blocking 的方式來溝通，因此，可以同時 handle 成千上萬個 events, 而不會有一般平行程式會遇到的 context switching 的 overhead.

- 而這些 non-blocking tasks 通常是透過 callback functions 來告知主程式任務完成，並且利用 event loop 來做到非同步 (asynchronous) 的排程，也就是說，事件之間不會有事先固定的順序，而是按照實際完成的先後順序來處理，可以盡量省下事件之間互相等待的情形

Node.js Modules 與 CommonJS 規範

- 最早 Node.js 是 follow CommonJS 的標準來實踐 modules, 但近來已逐漸使用 ECMAScript Specification 作為 default
- CommonJS 的誕生是為了要讓眾多的 JS modules 有一個共同的標準，得以彼此共生在 browser 以外的不同環境底下，建立應用生態系
  - 主要包含了 模組規範、套件規範、I/O、File System、Promise 等
  - Node.js 就是 CommonJS 的一個主要實踐者

Recall: CommonJS 規範

- CommonJS 是在 runtime 加載(require) modules

```
let { stat, exists, readFile } = require('fs');
const math = require('math');
```

- 然後就可以用了：

```
console.log(math.add(1, 2));
```

- 至於輸出模組，則用 "exports.functionName"

```
exports.incrementOne = function (num) {
  return math.add(num, 1);
};
```

CommonJS 規範。ES6

- ES6 則是強調「靜態時」就要決定模組的相依性
- By "export" &"import"

```
export var firstName = 'Michael';
export function multiply(x, y) { return x * y; }
       as MM;
export class MyClass extends React.Component...;
```

- from 後面的 path 可以是絕對或是相對位址; '.js' 可省

```
import { foo } from './myApp.js';
import { add, sub } from './myMath.js';
import { aVeryLongName as someName }
       from '../someFile.js'
```

"export default"

- 在前面的例子當中，使用者需要知道 import 進來的檔案裡頭原先的那些變數、function、class 的名字為何，需要跟原來檔案裡頭定義的名字一樣，才可以使用
  - 而且 import 時要記得加 { }
- "export default" 則讓我們可以不用管原來檔案裡頭這些function/class 叫什麼名字，甚至是可以 anonymous

```
export default (a, b) => (a+b);
```

"export default"

- 不過既然 function/class 都可以 anonymous 了，所以：
  - export 的檔案就只能有一個 "export default" 的 function or class
  - 在 import 時的名字是屬於 import 那個檔案的 scope，且不可以加 { }
  - from 後面的檔案名稱可以把 .js 省略

```
export default (a, b) => (a+b); // myMath.js
```

```
import myAdd from myMath;
```

比較這兩種寫法

- Specifically state that the class is extended from React.Component

```
import React from 'react'
class MyClass extends React.Component { ... }
```

- "React" is an "export default", while "Component" is a regular export

```
import React, {Component} from 'react'
class MyClass extends Component { ... }
```

My first Node.js example

- Save the following as a file "test.js"
- 然後執行 "node test.js"
- 打開 "localhost:3000" 看看！

```
const http = require('http');
const PORT = process.env.PORT || 3000;
const server = http.createServer(function(req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World!');
});
server.listen(PORT, function() {
  console.log
  (`Server listening on: http://localhost:${PORT}`);
});
```

- 當然，你也可以直接用 node 的 command line.
- 只要在 terminal 鍵入 "node", 你就可以試用各種 Node.js 的指令與模組了！ (Use .help to get help)

- 我們往後會再教大家如何建立 Node.js server, 現在先來學一個 Node.js 的 "web framework" --- Express.js

Learning "Express.js"

- My first Express App —

```
> npx express express-test
```

- Note: npx 為 node.js v5.2 之後的一個內建 CLI 工具，可以臨時性的安裝非全局性必要的套件，省下許多安裝及使用的流程與步驟、省下了磁碟空間，也避免了長期汙染

Learning "Express.js"

- It will create an "express app" called "express-test"

```
> cd express-test
> yarn
> yarn start
```

- Note: yarn 會把 node 套件裝在一個 global cache, 讓你可以不用每次都要從網路下載一大堆套件，對於套件的相依性，也有比較好的管理
- go to "localhost:3000"

Auto update with "nodemon"

- 不過，跟 create-react-app 不同的是，當你修改檔案的時候，上頁的 node server 並不會自動重啟
- 解決方式：

```
> yarn add nodemon // 相當於 npm install –save
```

// Note: "--save" to save the dependencies to package.json

- 開啟 package.json, 在 "scripts" 裡頭，加上

```
"devstart": "nodemon ./bin/www"
```

- 然後重啟 server by "yarn run devstart"

The "express-test" directory

- ./bin/www: node 開始執行的 script
- ./app.js: 定義 server
- ./routes/: 定義 application 的 routing
- ./views: 定義 application 的 viewing template

./bin/www

```
var app = require('../app');
var debug = require('debug')('express-test1:server');
var http = require('http');

var port = normalizePort(process.env.PORT || '3000');
app.set('port', port);

var server = http.createServer(app);

server.listen(port);
server.on('error', onError);
server.on('listening', onListening);
```

./app.js

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

var indexRouter = require('./routes/index');
var usersRouter = require('./routes/users');

var app = express();

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'jade');

app.use(...);  // middlewares

module.exports = app;  // for ./bin/www
```

./routes/index.js

```
var express = require('express');
var router = express.Router();

/* GET home page. */
router.get('/', function(req, res, next) {
  res.render('index', { title: 'Express' });
});

module.exports = router;
```

For viewing template, see "./views/index.jade"

```
extends layout

block content
  h1= title
  p Welcome to #{title}
```

蛤
是在
express
什麼？

Express 定義了各種 middlewares 來協助 client 與 server 進行溝通

```
Express is a routing and middleware web framework
that has minimal functionality of its own:
An Express application is essentially a series of
middleware function calls.
```

Types of Middlewares in Express

- Application-level middleware
- Router-level middleware
- Error-handling middleware
- Built-in middleware
- Third-party middleware

Application-level middleware

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

app.use([path,] callback [, callback...]);
app.METHOD(path, callback [, callback ...]);
// METHOD can be:
// checkout    copy    delete    get    head    lock
// merge    mkactivity    mkcol    move    m-search
// notify    options    patch    post    purge    put
// report    search    subscribe    trace    unlock
// unsubscribe
```

如果 path 沒有指定，則每次都會被執行

```
var app = express()

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

如果指定 path，則只有 routing path 符合的時候才會被執行

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

可以有多個 callback functions

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

如果沒有 next()

- 則執行完就會結束這個 request-response cycle

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

// This will never be called!
app.get('/user/:id', function (req, res, next) {
  res.end(req.params.id)
})
```

也可以傳 array of callbacks

```
function logOriginalUrl (req, res, next) {
  console.log('Request URL:', req.originalUrl)
  next()
}

function logMethod(req, res, next) {
  console.log('Request Type:', req.method)
  next()
}

var logStuff = [logOriginalUrl, logMethod]
app.get('/user/:id', logStuff, function (req, res, next) {
  res.send('User Info')
})
```

Router-level middleware

- 先用 application-level middleware 指定 routing

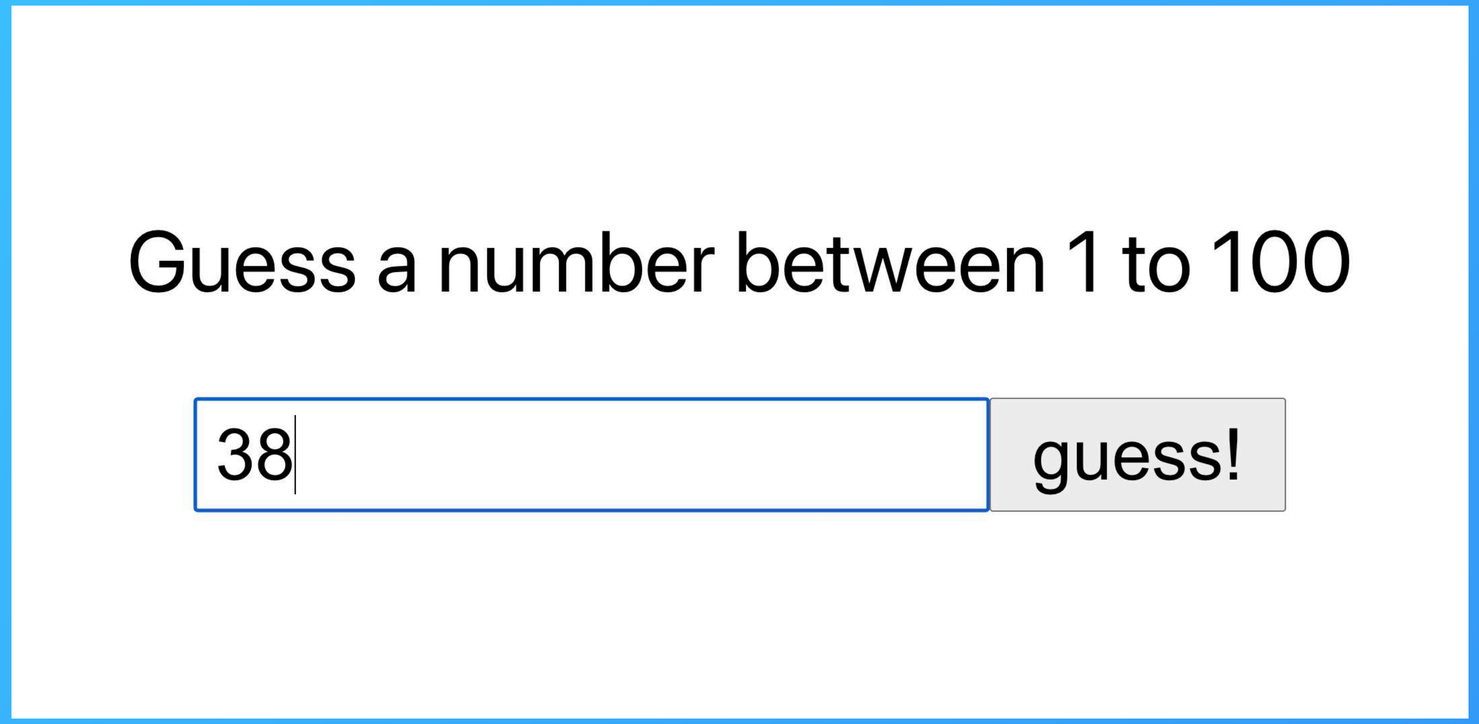
```
var express = require('express');
var app = express();
var usersRouter = require('./routes/users');
app.use('/', usersRouter);
```

- 定義 router-level middleware // "routes/users.js"

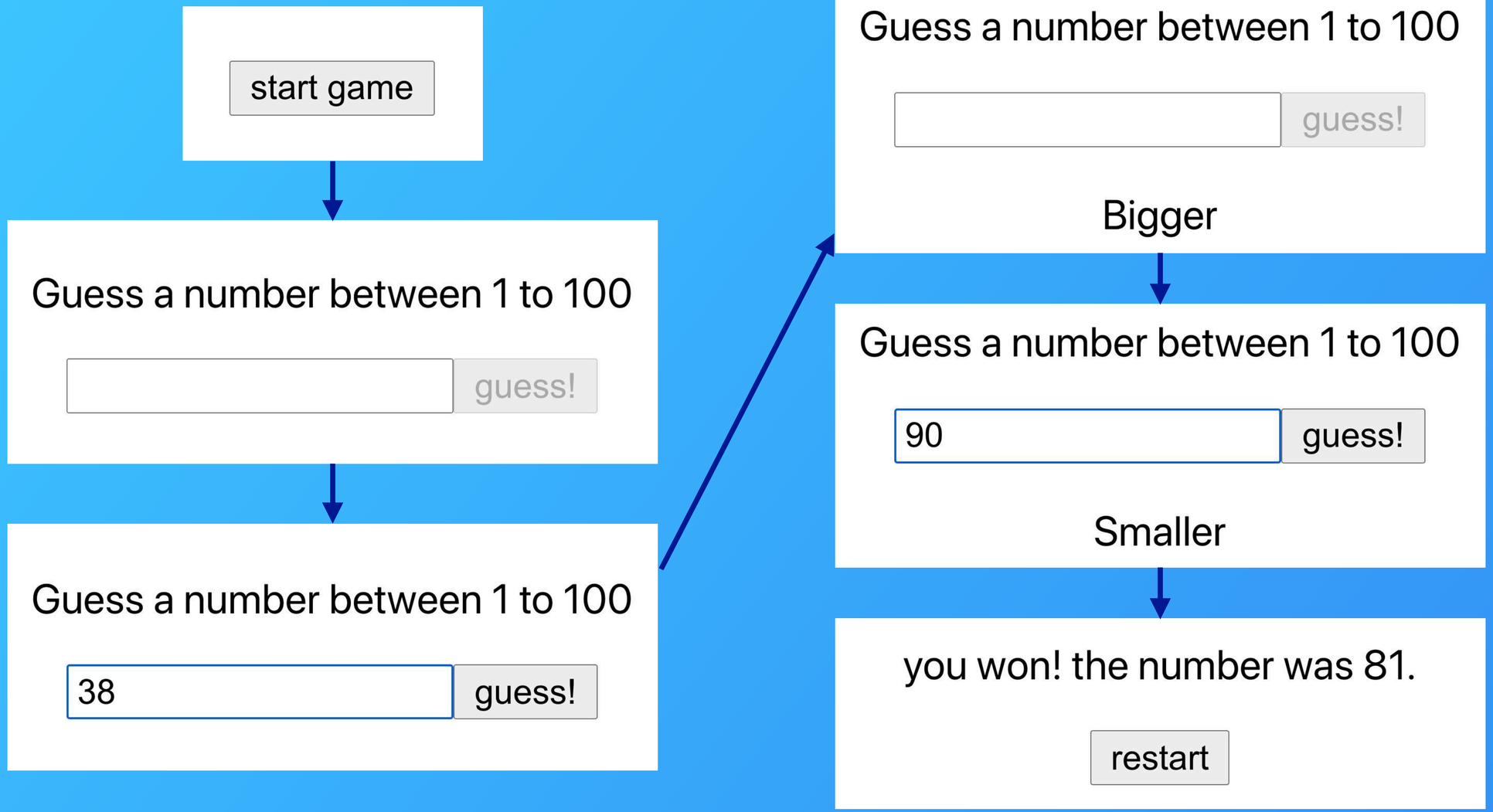
```
var express = require('express');
var router = express.Router();
router.get('/:id', function(req, res, next) {
  console.log('Request URL:', req.originalUrl)
  res.send(`respond with a resource ${req.params.id}`);
});
module.exports = router;
```

Let's use the incoming
HW#5 as an example...

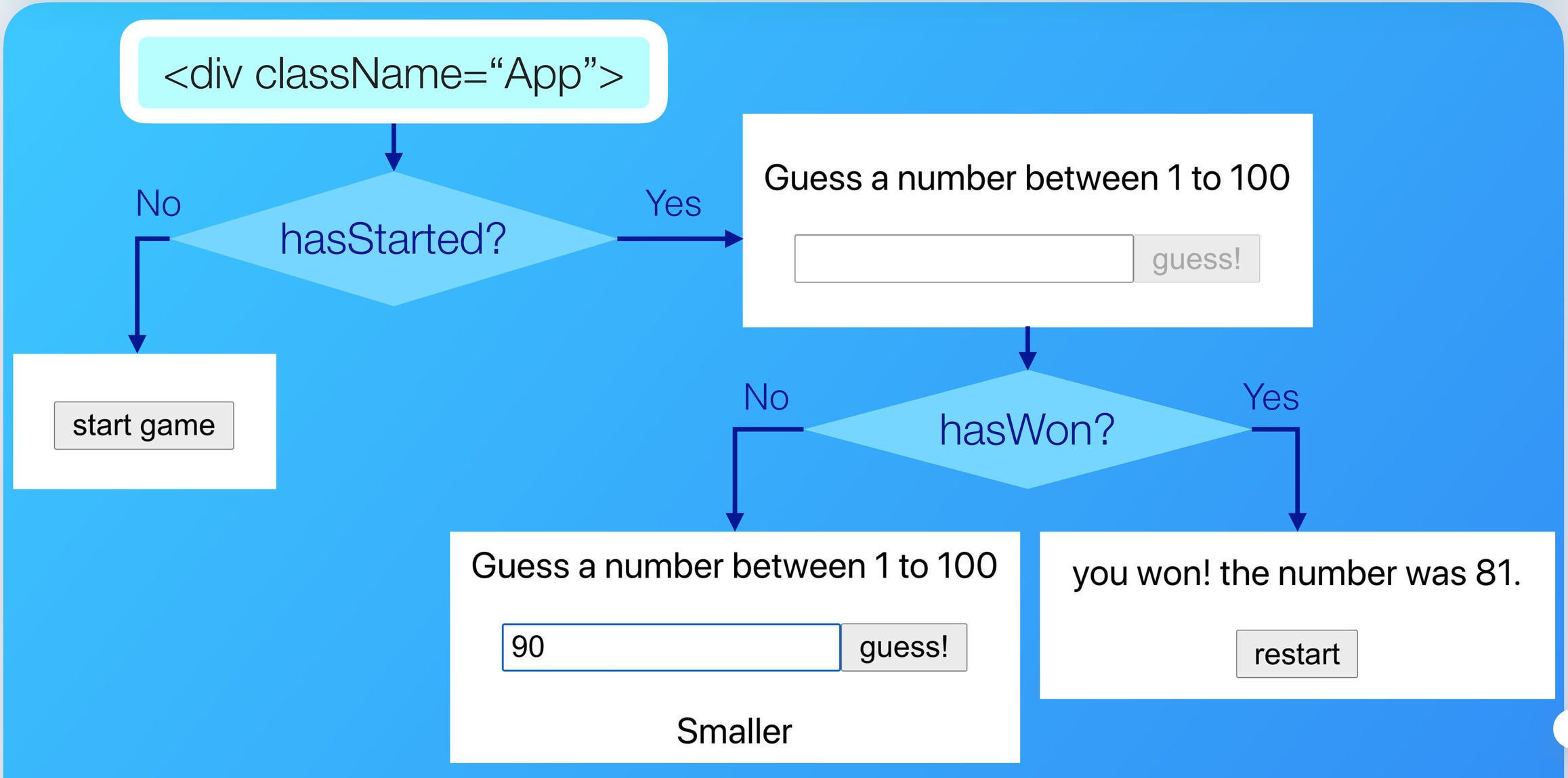
As Simplest As Possible (ASAP) Example



Basic Game Flow...



Define some states to control the view



【Client Side】

```
> cd src
// React App entry
> index.js
// Client side component
> App.js
// talk to server
> axios.js
```

yarn start

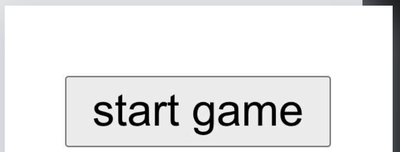
【Server Side】

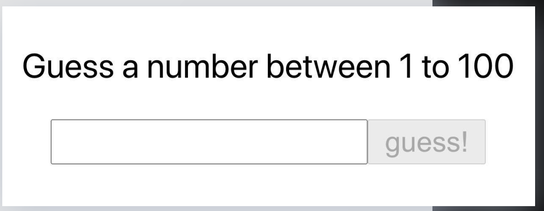
```
> cd server
// Server entry function
> server.js
// Game key logic
> routes/guess.js
// Game kernel function
> core/getNumber.js
```

yarn server

Client ◦ App.js — hasStarted, hasWon

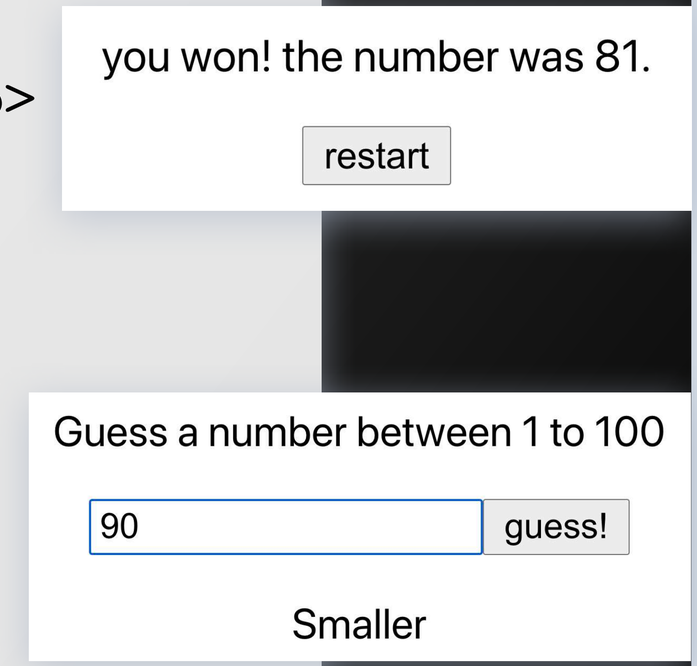
```
function App() {
  const [hasStarted, setHasStarted] = useState(false)
  const [hasWon, setHasWon] = useState(false)

  const startMenu = (
    <div> <button> start game </button> </div>
  )

  const game = (
    <div> {hasWon ? winningMode : gameMode}
    </div>
  )

  return <div className="App">
          {hasStarted ? game : startMenu}</div>
}
```





Client ◦ App.js — number, status

```
function App() {
  ...
  const [number, setNumber] = useState('')
  const [status, setStatus] = useState('')
  const winningMode = (
    <>
      <p> you won! the number was {number}. </p>
      <button> restart </button>
    </>
  )
  const gameMode = (
    <>
      <p> Guess a number between 1 to 100 </p>
      <input ...></input>
      <button ...> guess! </button>
      <p> {status} </p>
  )
  ...
}
```



Client ◦ App.js ◦ startMenu

```
const startMenu = (
  <div>
    <button
      onClick={async () => {
        await startGame()
        setHasStarted(true)
      }}
    >
      start game
    </button>
  </div>
)
```

An async function
(explained later)

Why async/await?
What will happen if we
don’t use async/await?

Client ◦ App.js ◦ game (winning case)

```
const winningMode = (
  <>
    <p>you won! the number was {number}.</p>
    <button
      onClick={async () => {
        await restart()
        setHasWon(false)
        setStatus('')
        setNumber('')
      }}
    >
      restart
    </button>
  </>
)
```

Client ◦ App.js ◦ game (play mode)

```
const gameMode = (
  <>
    <p>Guess a number between 1 to 100</p>
    <input
      value={number}
      onChange={(e) => setNumber(e.target.value)}
    ></input>
    <button
      // TODO: use async/await to call guess(number),
      // process the response to set the proper state values
      onClick={}
      disabled={!number}
    >
      guess!
    </button>
    <p>{status}</p>
  </>
)
```

axios.js

- "Axios is a promise based HTTP client for the browser and Node.js. Axios makes it easy to send asynchronous HTTP requests to REST endpoints and perform CRUD operations. It can be used in plain JavaScript or with a library such as Vue or React."

  // CRUD: Create, Read, Update, Delete

ref: http://zetcode.com/javascript/axios/

Axios API

- 基本的語法是：axios(config)

```
// Send a POST request
axios({
  method: 'post',
  url: '/user/12345',
  data: {
    firstName: 'Fred',
    lastName: 'Flintstone'
  }
});
```

```
// GET request for remote
// image in node.js
axios({
  method: 'get',
  url:
    'http://bit.ly/2mTM3nY',
  responseType: 'stream'
})
.then(function (response) {
  response.data.pipe
  (fs.createWriteStream
   ('ada_lovelace.jpg'))
});
```

Axios Request Config (ref)

- 前頁在呼叫 axios API 時，傳入的 config 有右列的 options, 其中只有 url is required, 而 method 沒有被指定時，default 是 GET

```
{
  url: '/user', method: 'get',  baseURL: 'https://some-domain.com/api/',
  transformRequest: [function (data, headers) { return data; }],
  transformResponse: [function (data) { return data; }],
  headers: {'X-Requested-With': 'XMLHttpRequest'},
  params: { ID: 12345 },
  paramsSerializer: function (params) {
    return Qs.stringify(params, {arrayFormat: 'brackets'}) },
  data: { firstName: 'Fred' },  data: 'Country=Brasil&City=Belo Horizonte',
  timeout: 1000,         withCredentials: false,
  adapter: function (config) { },
  auth: { username: 'janedoe', password: 's00pers3cret' },
  responseType: 'json',          responseEncoding: 'utf8',
  xsrfCookieName: 'XSRF-TOKEN', xsrfHeaderName: 'X-XSRF-TOKEN',
  onUploadProgress: function (progressEvent) { },
  onDownloadProgress: function (progressEvent) { },
  maxContentLength: 2000,        maxBodyLength: 2000,
  validateStatus: function (status) { return status >= 200 && status < 300; },
  maxRedirects: 5,       socketPath: null,
  httpAgent: new http.Agent({ keepAlive: true }),
  httpsAgent: new https.Agent({ keepAlive: true }),
  proxy: { host: '127.0.0.1', port: 9000,
    auth: { username: 'mikeymike', password: 'rapunz3l' } },
  cancelToken: new CancelToken(function (cancel) { }),
  decompress: true
}
```

Axios Request Method Aliases

- For convenience, 我們常常把 method 搬出來，變成 alias

```
axios.request(config)
axios.get(url[, config])
axios.delete(url[, config])
axios.head(url[, config])
axios.options(url[, config])
axios.post(url[, data[, config]])
axios.put(url[, data[, config]])
axios.patch(url[, data[, config]])
```

Axios API in Promise

```
// Make a request for a user with a given ID
axios.get('/user?ID=12345')
.then(function (response) {
  // handle success
  console.log(response);
})
.catch(function (error) {
  // handle error
  console.log(error);
})
.then(function () {
  // always executed
});
```

Axios API in Async/Await

- 使用 async/await, code 變得比較直觀

```
// Add the `async` keyword to your outer
// function/method.
async function getUser() {
  try {
    const response = await
      axios.get
      ('/user?ID=12345');
    console.log(response);
  } catch (error) {
    console.error(error);
  }
}
```

Axios Instance

- 常常不同的 applications 會有相同的 baseURL 但有不同的 routing ==> create 一個 axios instance

```
const instance = axios.create
  ({ baseURL: 'http://localhost:4000/api/guess' })

const startGame = async () => {
  const { data: { msg } } = await instance.post('/start')
  return msg
}
const guess = async (number) => {
  const { data: { msg } } = await instance.get
                            ('/guess', { params: { number } })
  return msg
}
const restart = async () => {
  const { data: { msg } } = await instance.post('/restart')
  return msg
}
```

Axios Response Schema (ref)

- 利用 axios API 從 server 端獲得資訊時，回傳的 response 的 schema 如右：

```
{
  data: {},
  status: 200,
  statusText: 'OK',
  headers: {},
  config: {},
  request: {}
}
```

Server。guess.js。express.Router()

- 產生一個 router 物件

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

- 語法

```
const router.METHOD(_path_, callBack)
```

- 符合 _path_ 才會用 callBack 回應 http 的 METHOD
- callBack = (req, res) => { ... }

startGame <-> post('/start', ...)

- Client side ("src/axios.js")—

```
const startGame = async () => {
  const {
    data: { msg }
  } = await instance.post('/start')
  return msg
}
```

Evoked by clicking on the “start game” button (see p95)

- Server side —

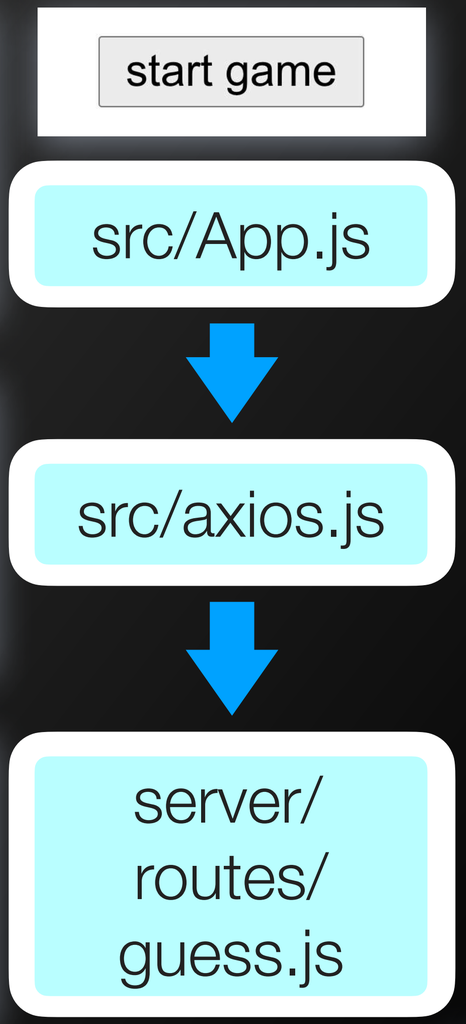
```
router.post('/start', (_, res) => {
  getNumber(true)

  res.json({ msg: 'The game has started.' })
})
```

From Frontend to Backend

```
function App() {
    <div> <button onClick={async () => {
            await startGame()}> start game </button>
    </div>
)}
```

```
const instance = axios.create({ baseURL: '...' })
const startGame = async () => {
  const { data: { msg } } = await instance.post('/start')
  return msg
}
```

```
const router = express.Router()
router.post('/start', (_, res) => {
  getNumber(true)
  res.json({ msg: 'The game has started.' })
})
```



HW#5 TODOs

- 所有 TODOs 都在 "TODO"
- Client side
  1. (App.js) 完成 "play mode" 時按鈕的邏輯
- Server side
  1. (core/getNumber.js) 產生 0-99 間的隨機亂數
  2. (routers/guess.js)
     - 完成 '/guess' GET method 的遊戲與判斷邏輯
     - 完成 '/restart' POST method service

感謝聆聽！

Ric Huang / NTUEE

(EE 3035) Web Programming