05. 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

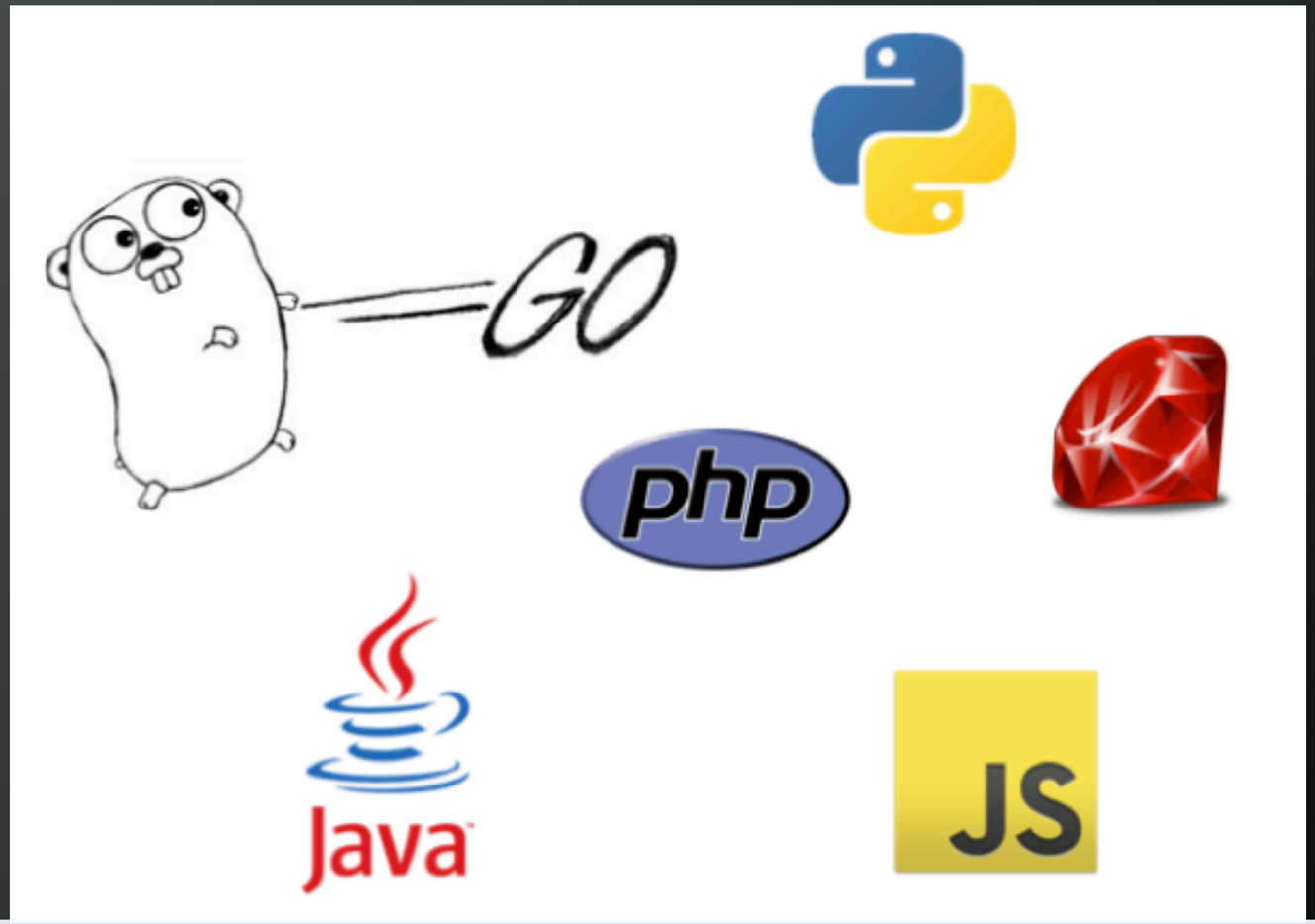
Toward a complete web service

- 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

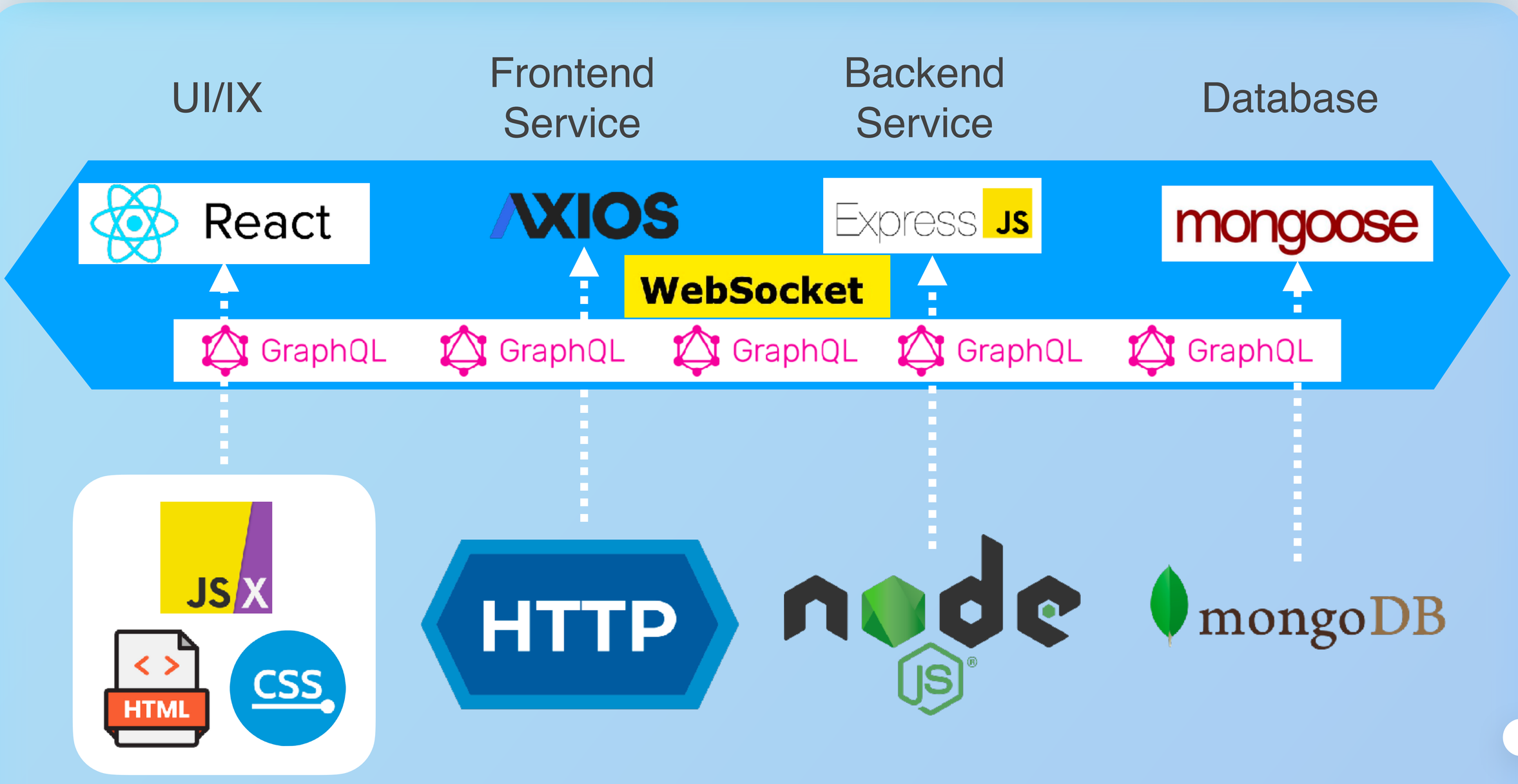
Localhost as a local web server

- 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

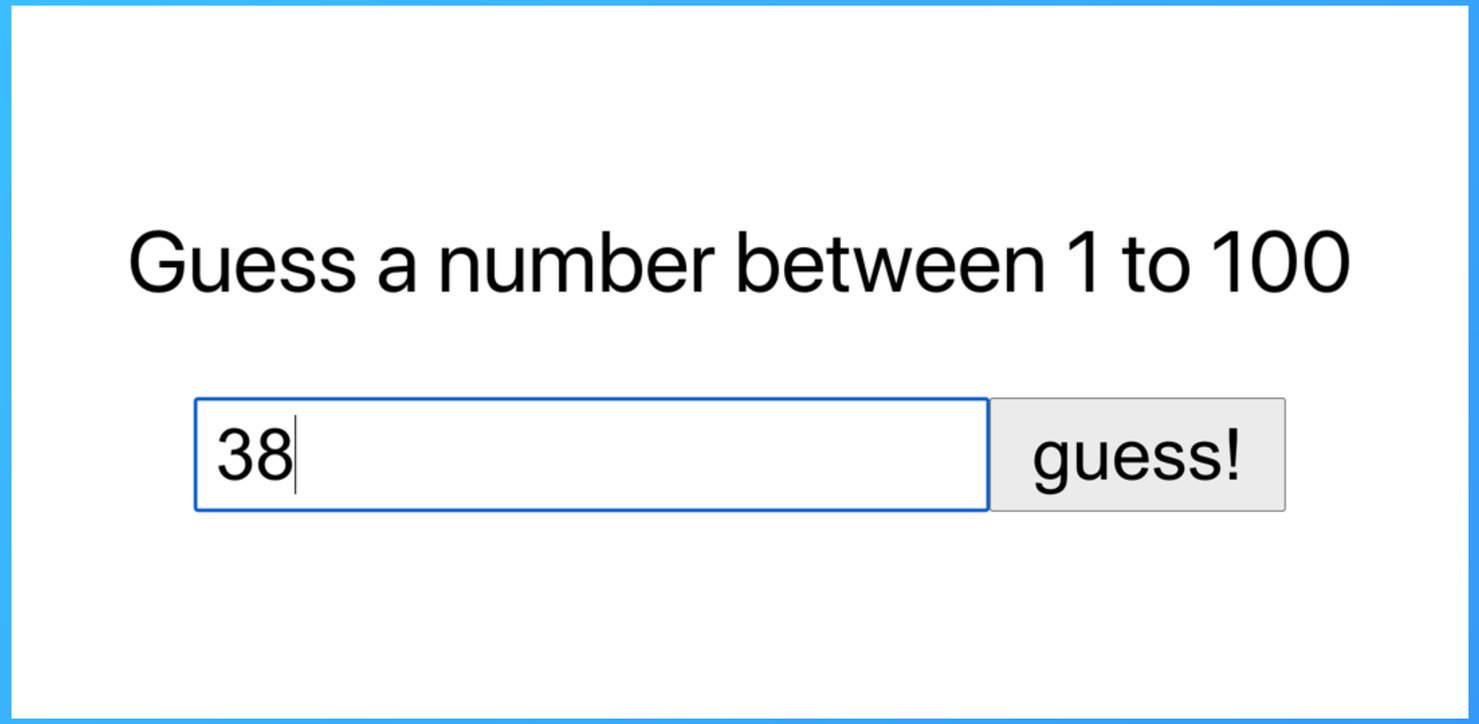


A Typical Web Service Infrastructure

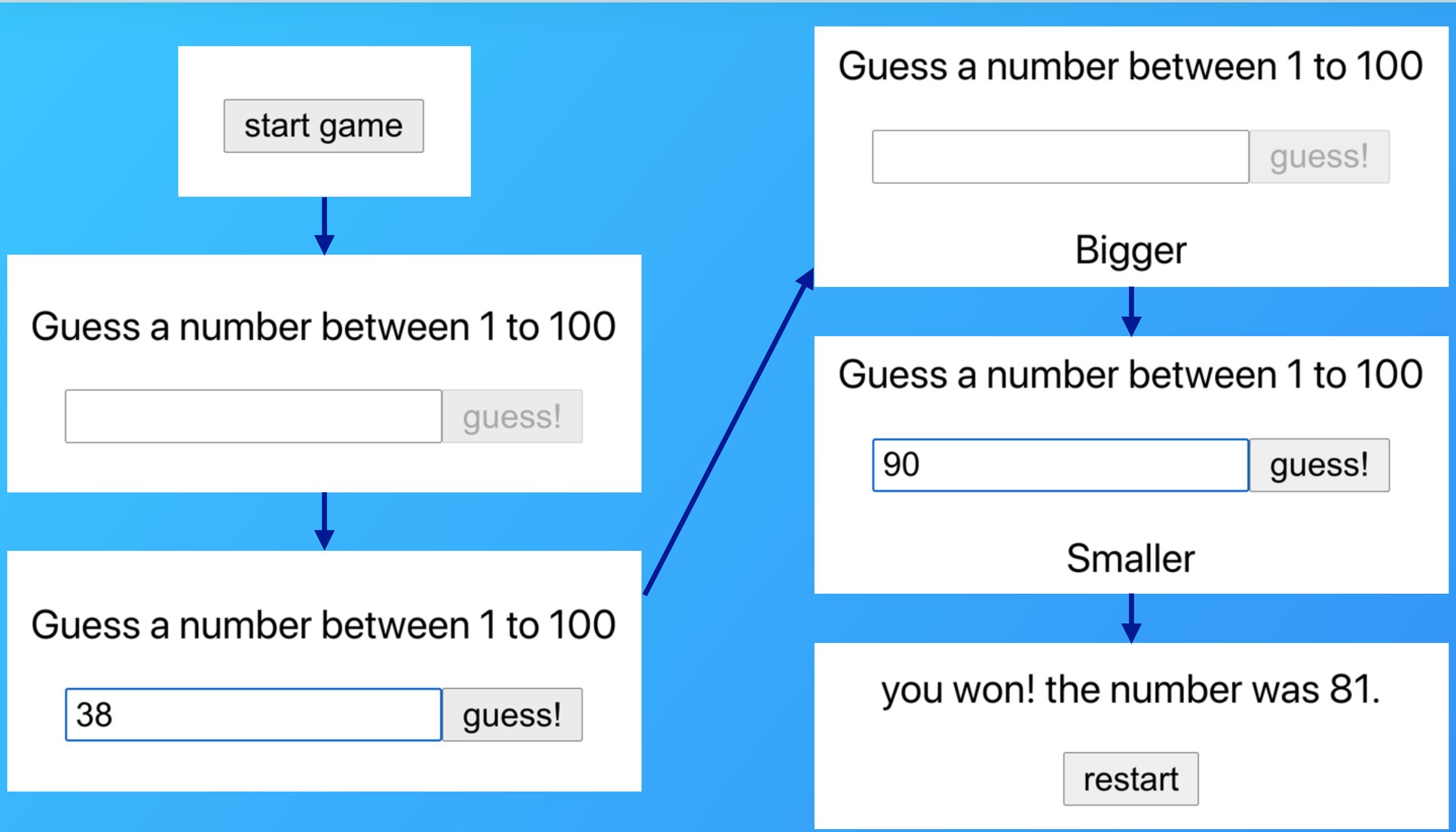


Let's use the incoming
HW#5 as an example...

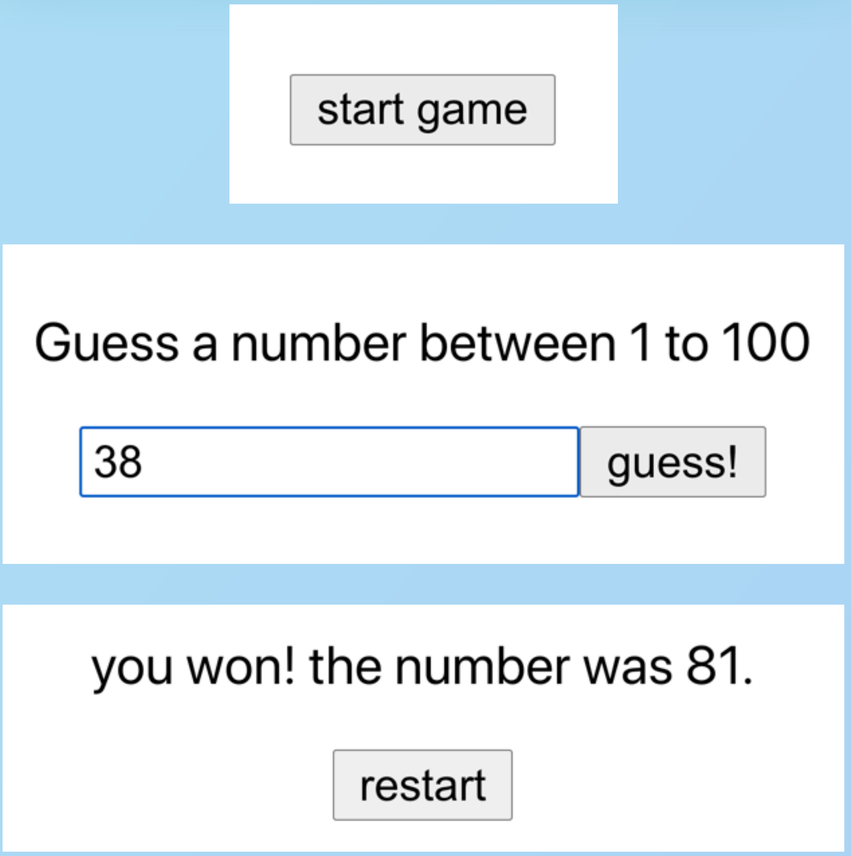
As Simplest As Possible (ASAP) Example



Basic Game Flow...



Frontend



Different Views

Backend

Start
Random Number Generation

Guess
Check and respond

Restart
Return to Start

Different Services

Creating a Full-Stack Project

- Create a new project in hw5

```
> cd hw5; rm -rf *
> yarn init -y
```

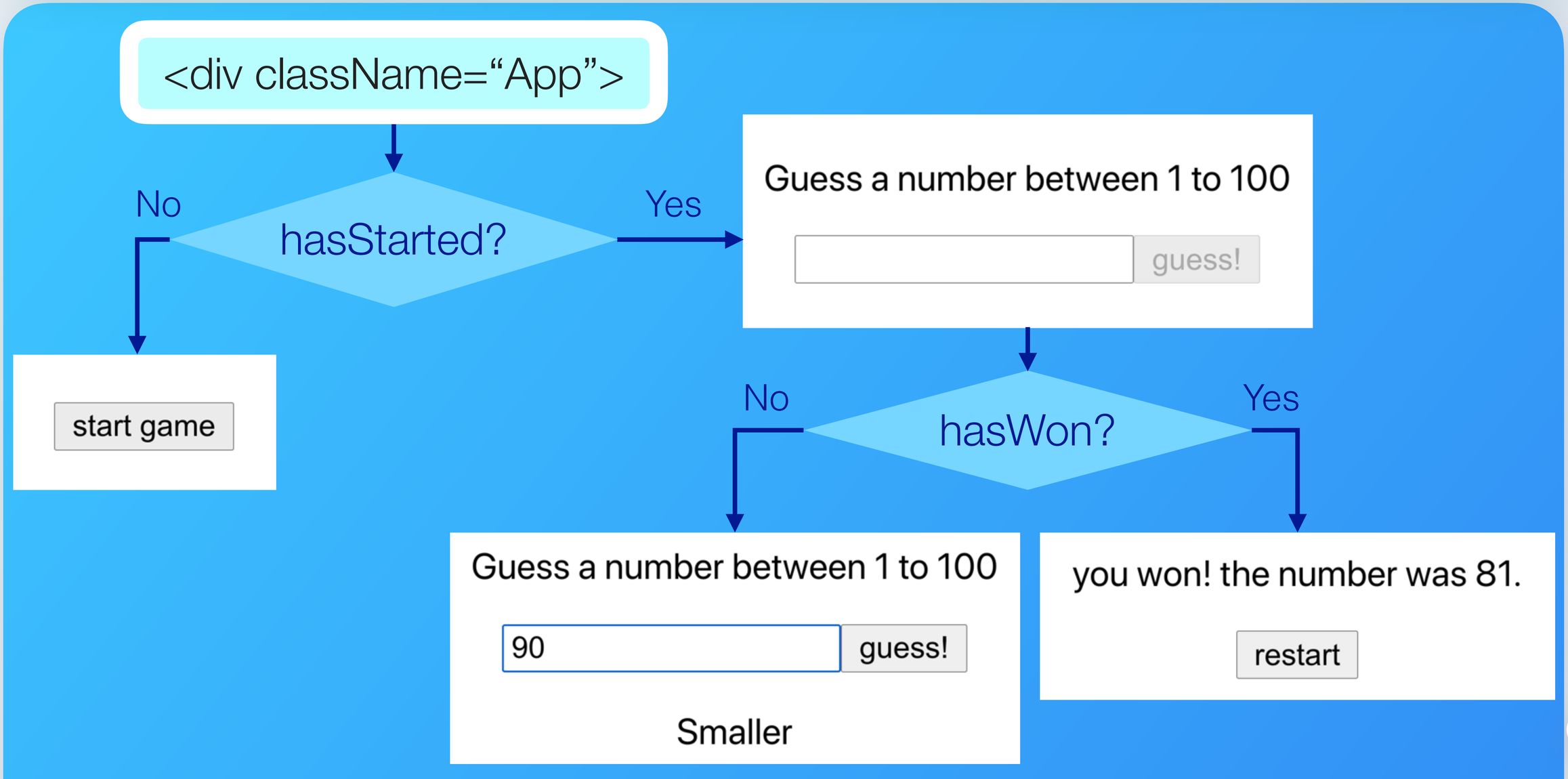
- Create a React project as "frontend"

```
> create-react-app frontend
```

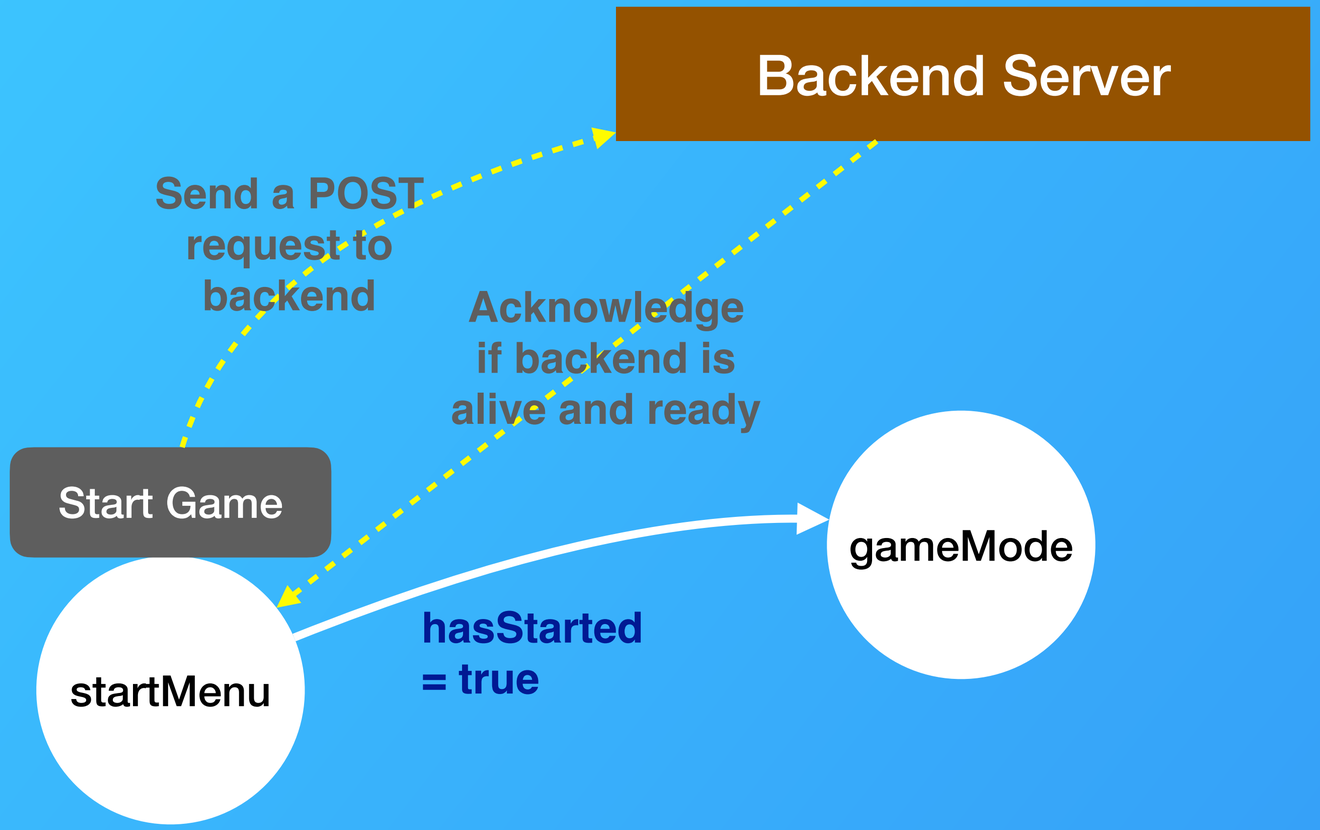
- Insert these lines to hw5/package.json

```
"scripts": {
    "start": "cd frontend && yarn start",
    "server": "cd backend && yarn server"
  }
}
```

Frontend ◦ Flow of the Guessing Game

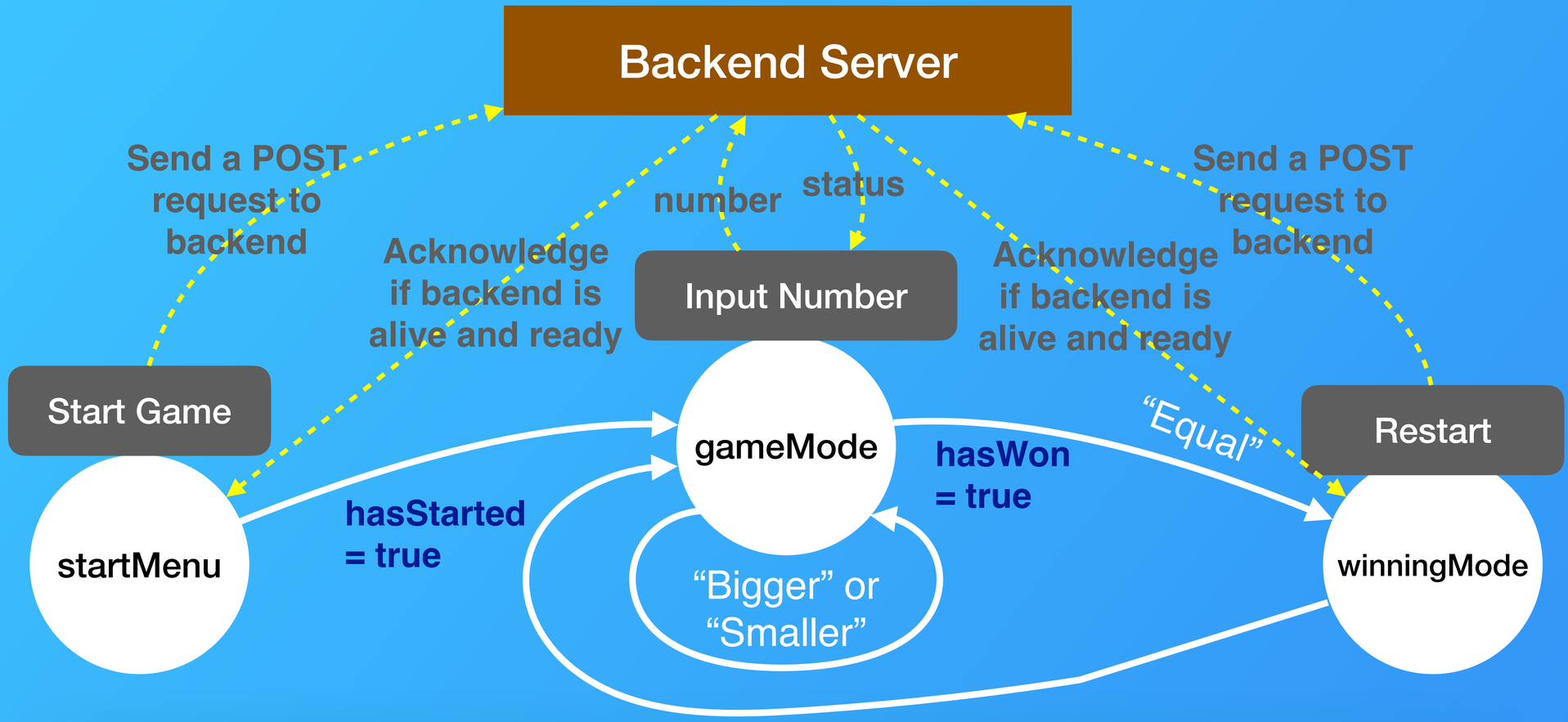


Frontend ◦ Define some states **to control the view**



States: (hasStarted)

Frontend ◦ Define some states **to control the view**



States: (hasStarted, hasWon, number, status)

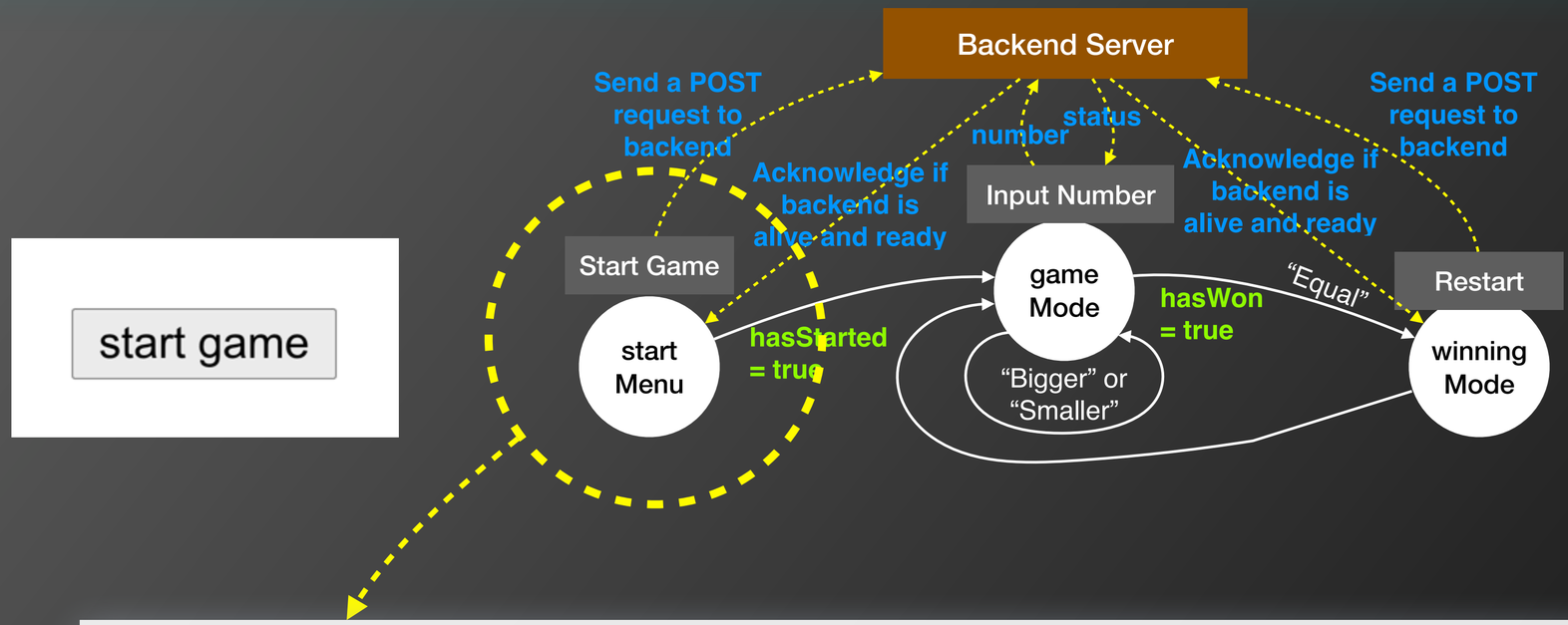
Frontend • Define hooks for states

- Use Hooks to define states:

```
function App() {
  const [hasStarted, setHasStarted] = useState(false)
  const [hasWon, setHasWon] = useState(false)
  const [number, setNumber] = useState('')
  const [status, setStatus] = useState('')

  return ...
}
```

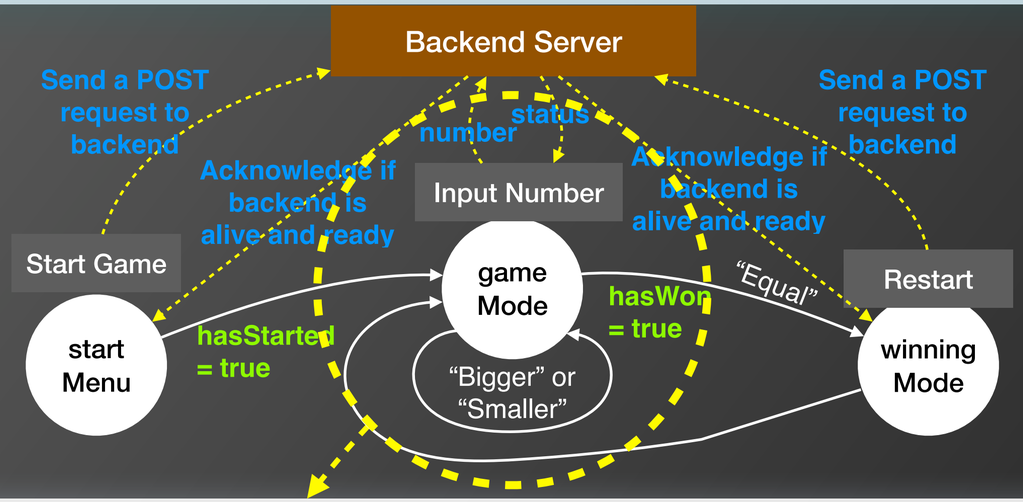
- Note:
  - 不在 frontend 檢查輸入數字格式，所以用字串存 number
  - 如果不是合法數字，backend 回傳 “406” Error code (see “HTTP” later)

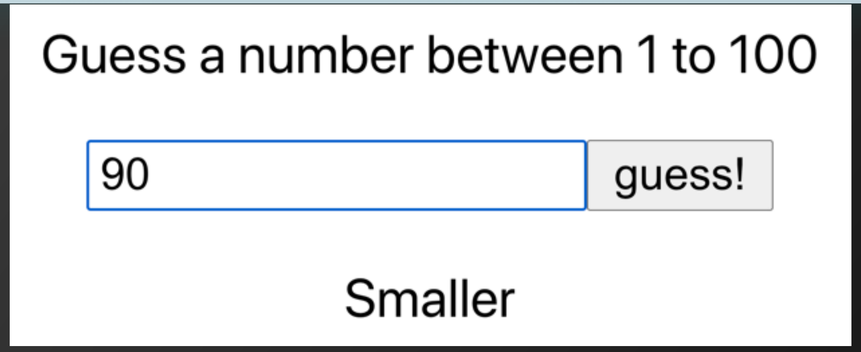
Frontend • Define three different views (1/3)




```
const startMenu =
  <div>
    <button onClick = {
      // someFunctionToBackend; and setHasStarted
    } > start game </button>
  </div>
```

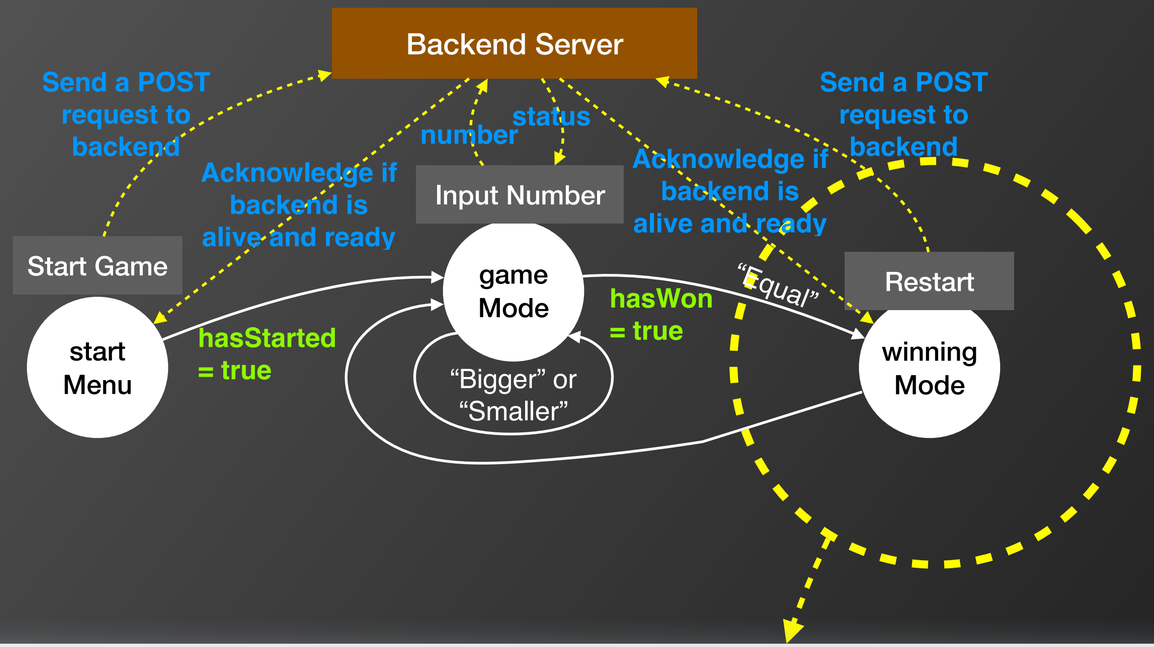
Frontend • Define three different views (2/3)






```
const gameMode =
  <>
    <p>Guess a number between 1 to 100</p>
    <input   // Get the value from input
    ></input>
    <button  // Send number to backend
      onClick={handleGuess}
      disabled={!number}
    >guess!</button>
    <p>{status}</p>
  </>
```

Frontend ◦ Define three different views (3/3)



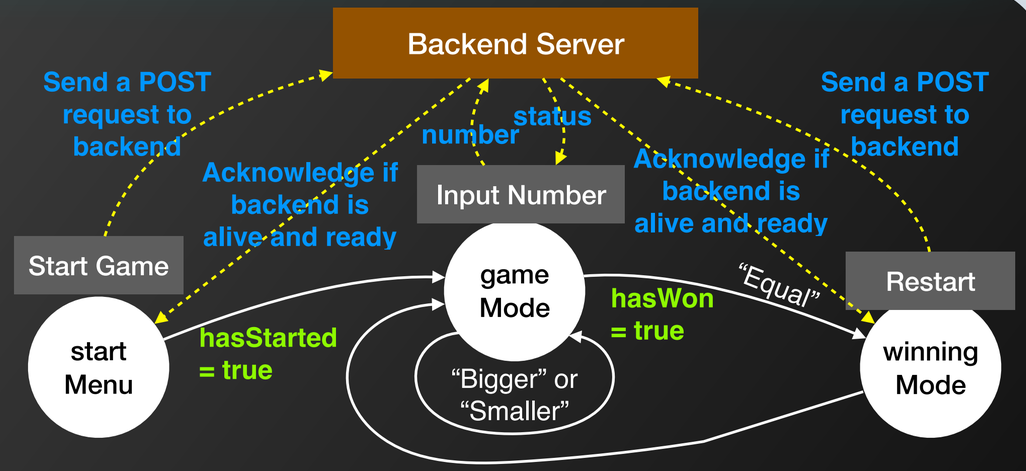
you won! the number was 81.

restart

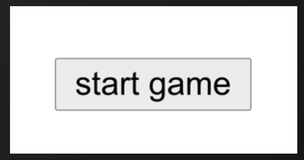
```
const winningMode = (
    <>
      <p>you won! the number was {number}.</p>
      <button  // Handle restart for backend and frontend
      >restart</button>
    </>
  )
```

Frontend ◦ Handle final rendering

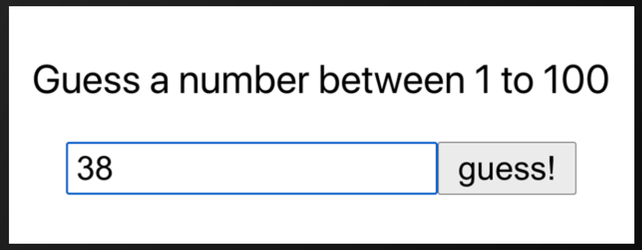
```
function App() {
  // Define states
  // Define three different views


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



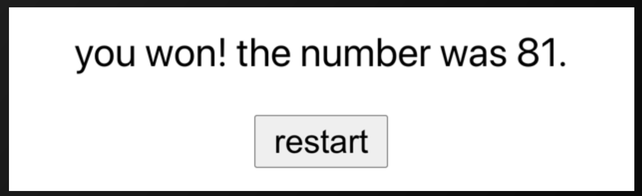
startMenu



gameMode



winningMode



Looks good.

但這個 handleGuess() 要怎麼寫 (才能把數字順利地送到後端)？

```
const gameMode =
  <>
    <p>Guess a number between 1 to 100</p>
    <input   // Get the value from input
    ></input>
    <button  // Send number to backend
      onClick={handleGuess}
      disabled={!number}
    >guess!</button>
    <p>{status}</p>
  </>
```

Asynchronous Communication Issues

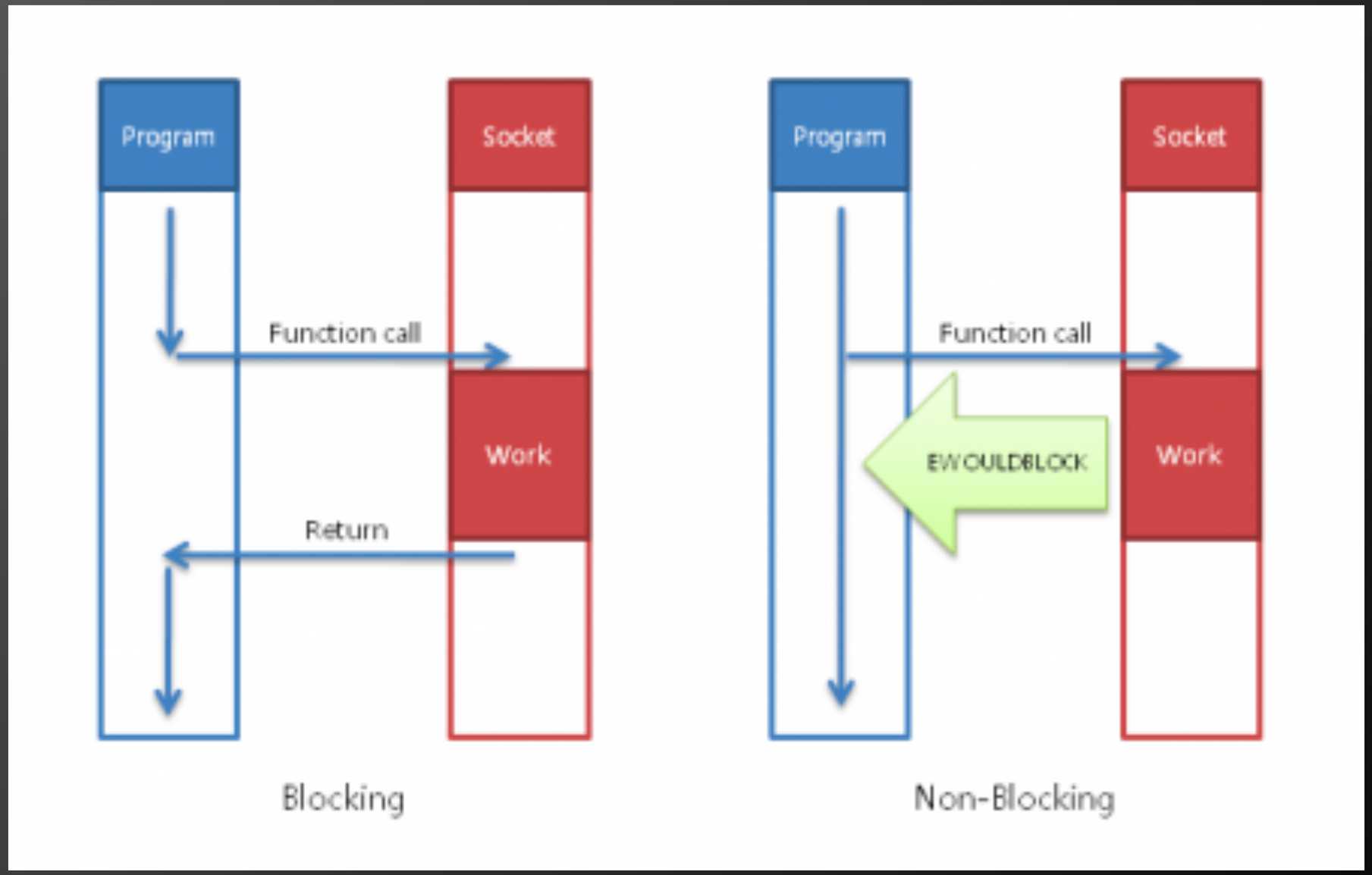
- A naive trial:

```
const handleGuess = () => {
  const response = processGuessByBackend(number)

  if (response === 'Equal') setHasWon(true)
  else {
    setStatus(response)
    setNumber('')
  }
}
```

What's the problem?

- 假設 "processGuessByBackend()" 要花一些時間執行，
  那麼 handleGuess() 會停在這行嗎？
  還是會直接繼續執行下去？
  如果是停在那邊，要等到什麼時候？

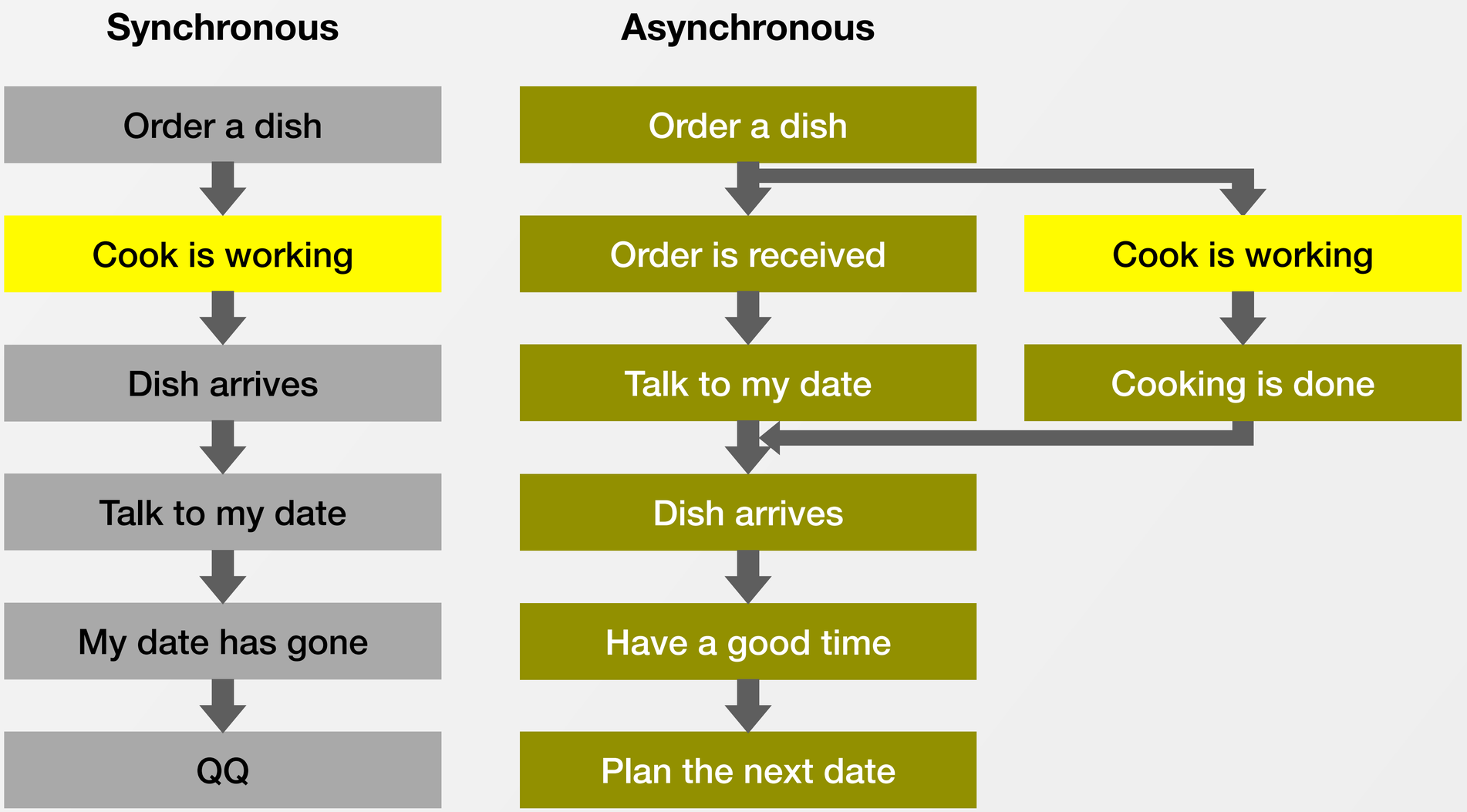
Recall: Blocking vs. Non-Blocking



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 on console…

```
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，等到 async function 執行完後呼叫執行」，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

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

floppy.load('disk1', function (data1) {
    floppy.prompt('Please insert disk 2', function() {
        floppy.load('disk2', function (data2) {
            floppy.prompt('Please insert disk 3', function() {
                floppy.load('disk3', function (data3) {
                    floppy.prompt('Please insert disk 4', function() {
                        floppy.load('disk4', function (data4) {
                            floppy.prompt('Please insert disk 5', function() {
                                floppy.load('disk5', function (data5) {
                                    floppy.prompt('Please insert disk 6', function() {
                                        floppy.load('disk6', function (data6) {
                                            //if node.js would have existed in 1995
                                        });
                                    });
                                });
                            });
                        });
                    });
                });
            });
        });
    });
});

```

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)

- 先把是否會拿到手機這件事定義成一個 Promise 物件

```
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);
    }
  );
```

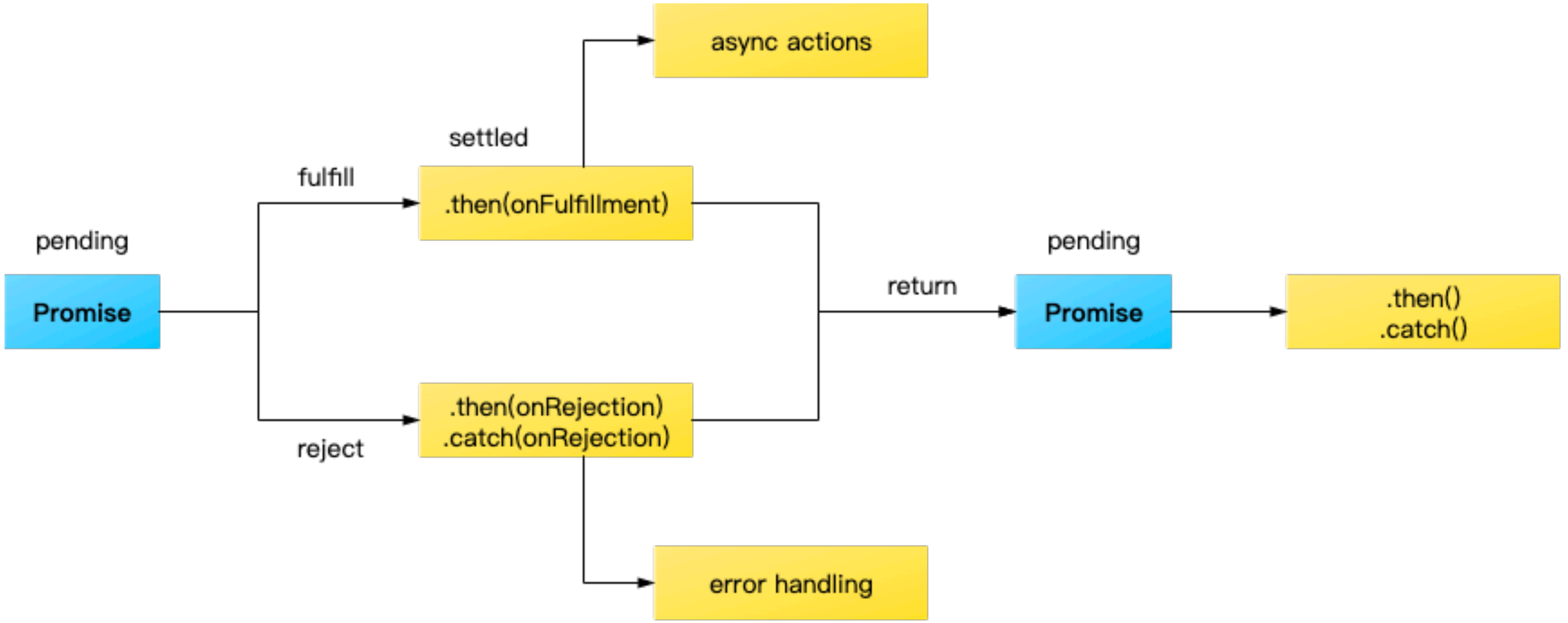
Promise Example (ref)

- 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)
  .catch(momUnhappy);
  // continue my life…
})()
```

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 用來修飾一個 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 vs. No-Async

- Try to compare and explain

```
console.log("started execution");
const f = async() => {
  console.log("Start async...");
  await waitThreeSeconds();
  console.log("Done async!");
}
f()
console.log("finished execution");
```

需要等待 async 完成才做的 action

不想等待 async 完成才做的 action

- started execution
- Start async...
- Wait 3 seconds...
- finished function
- finished execution
- Done async!

```
console.log("started execution");
const f = () => {
  console.log("Start async...");
  waitThreeSeconds();
  console.log("Done async!");
}
f()
console.log("finished execution");
```

- started execution
- Start async...
- Wait 3 seconds...
- finished function
- Done async!
- finished execution

More 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');
  }
);
```

但請注意，這樣寫是錯的！

```
const f = async(() => {
  console.log("Start async...");
  await waitThreeSeconds();
  console.log("Done async!");
})

__
Uncaught SyntaxError: await is only valid in async
functions and the top level bodies of modules
```

所以，handleGuess 應該改寫成 —

```
const handleGuess = async () => {
  const response = await processGuessByBackend(number)

  if (response === 'Equal') setHasWon(true)
  else {
    setStatus(response)
    setNumber('')
  }
}
```

我們現在大致準備好前端的
畫面了，接下來要來看看它
要如何跟後端對接服務

Backend

Frontend 如何把 input number 傳給 backend?

Backend 檢查完後如何把 status 回傳給 frontend

Start
Random Number Generation

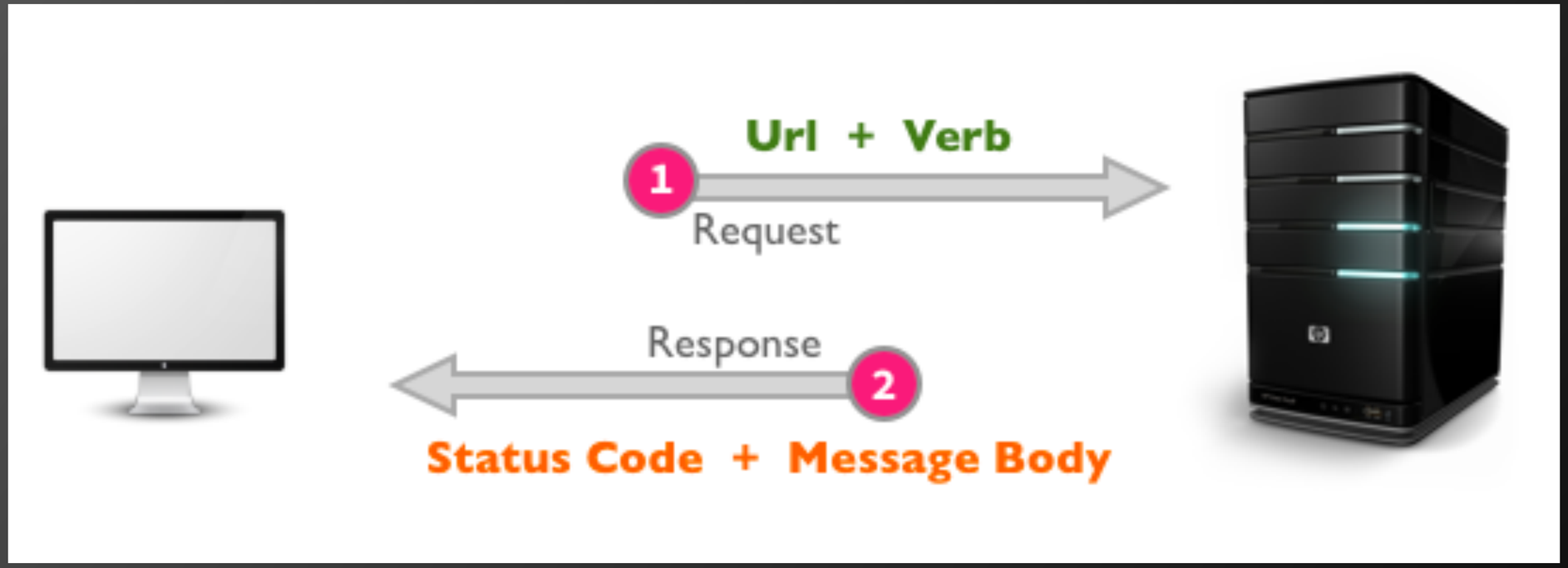
Guess
Check and respond

Restart
Return to Start

HyperText Transfer Protocol (HTTP)

- 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

常見的 HTTP Request Methods (Verb) (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 (Verb) (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 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
```

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

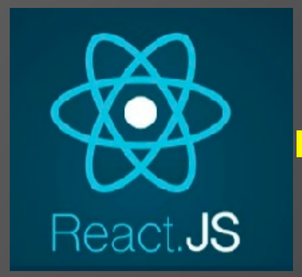
```
HTTP/1.1 200 OK // Status-line
Accept-Ranges: bytes
Cache-Control: public, max-age=0
Connection: keep-alive
Content-Length: 53
Content-Type: text/html; charset=UTF-8
Date: Tue, 11 Oct 2016 16:32:33 GMT
ETag: "53-1476203499000"
Last-Modified: Tue, 11 Oct 2016 16:31:39 GMT
// empty line
<html> // message-body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

HTTP 狀態碼

- 當使用 HTTP 溝通時不管是成功或是失敗，都應該回傳適當的狀態碼，以利雙方了解並且判斷錯誤原因。
- HTTP狀態碼的官方登錄檔由網際網路號碼分配局（Internet Assigned Numbers Authority）維護，分配如下：
  - 1xx: 代表請求已被接受，需要繼續處理，如: 100 Continue
  - 2xx: 代表請求已成功被伺服器接收、理解、並接受，如: 200 OK
  - 3xx: 代表重新導向，需要客戶端採取進一步的操作才能完成請求，如: 300 Multiple Choices
  - 4xx: 代表客戶端錯誤，妨礙了伺服器的處理，如: 403 Forbidden, 404 Not Found, 406 Not Acceptable
  - 5xx: 代表伺服器端錯誤，500 Internal Server Error, 502 Bad Gateway, 504 Gateway Timeout
- 一些有趣的圖像化 HTTP Status: HTTP Cats, HTTP Dogs

那，React 到底是怎麼透過
HTTP 把資料傳到後端的呢？

Well, you may have heard about OSI 7–Layer Model



7 Layers of the OSI Model

Application	• End User layer • HTTP, FTP, IRC, SSH, DNS
Presentation	• Syntax layer • SSL, SSH, IMAP, FTP, MPEG, JPEG
Session	• Synch & send to port • API's, Sockets, WinSock
Transport	• End-to-end connections • TCP, UDP
Network	• Packets • IP, ICMP, IPSec, IGMP
Data Link	• Frames • Ethernet, PPP, Switch, Bridge
Physical	• Physical structure • Coax, Fiber, Wireless, Hubs, Repeaters

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."

  // REST: RESTful API (covered later)

  // 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 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: {}
}
```

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
}
```

Server's API Endpoint

Routing for the post/get APIs

API。Endpoint。Routing

- Application Programming Interface (API): 用來定義兩個系統 (e.g. 前端與後端) 之間溝通的介面
- Endpoint: 溝通介面其中一端的端點，通常是一個 URL
- Routing: Routing 之於一個網路服務系統，就像是資料夾/檔案之於一個電腦系統一樣，是讓網路服務在定義其不同用途的 API 時，可以有自己的 "address (URL)", 但這個 URL 通常不是對應到一個實體的資源或者是某個後端程式擺放的位子，它只是方便讓不同的 API 可以有自己獨立的 service URL

Closer look at the previous example...

可以想像 “startGame” 是對應到後端的:
post('http://localhost:4000/api/guess/start') 這個 API

```
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
}
```

可以想像後端的 “guess” 這個 API 會回應 :
{ msg : "Some message" }

Wrapping up: HW#5 Frontend

src/App.js

```
import { useState } from 'react'
import './App.css'
import { guess, startGame, restart }
  from './axios'
function App() {
  // Define states
  // Define three different views
  const game =
    <div>
      {hasWon ?
       winningMode : gameMode}
    </div>
  return <div className="App">
    {hasStarted ? game : startMenu}
    </div>
}
export default App
```

適當地應用在這些 views 的 event handlers 裏頭

src/axios.js

```
import axios from 'axios'
const instance =
axios.create({ baseURL: '...' })
const startGame = async () => {
  const { data: { msg } }
  = await instance.post('/start')
  return msg
}
const guess = async (number) => {
  try {
    ...
    return msg
  }
  catch (error) { ... }
}
const restart = ...
export { startGame, guess, restart }
```

記得 "yarn add axios"

所以，handleGuess 應該改寫成 —

```
import { guess, startGame, restart } from './axios'
...     // 誰的 event 綁定了 handleGuess?
const handleGuess = async () => {
  const response = await guess(number)

  if (response === 'Equal') setHasWon(true)
  else {
    setStatus(response)
    setNumber('')
  }
}
```

你的前端跑起來了嗎？

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

Backend。Preparing...

- 先依建立 backend 的目錄與 basic packages

```
> cd hw5; mkdir backend
> cd backend; yarn init -y
> yarn add cors express nodemon
> yarn add -D @babel/cli @babel/core @babel/node @babel/preset-env
```

- 在 backend 增加一個 .babelrc 的檔案

```
{
  "presets": [
    "@babel/preset-env"
  ]
}
```

- 在 backend 的 package.json 加上：

```
  "scripts": {
    "server": "nodemon server.js --ext js --exec babel-node"
  },
```

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++)

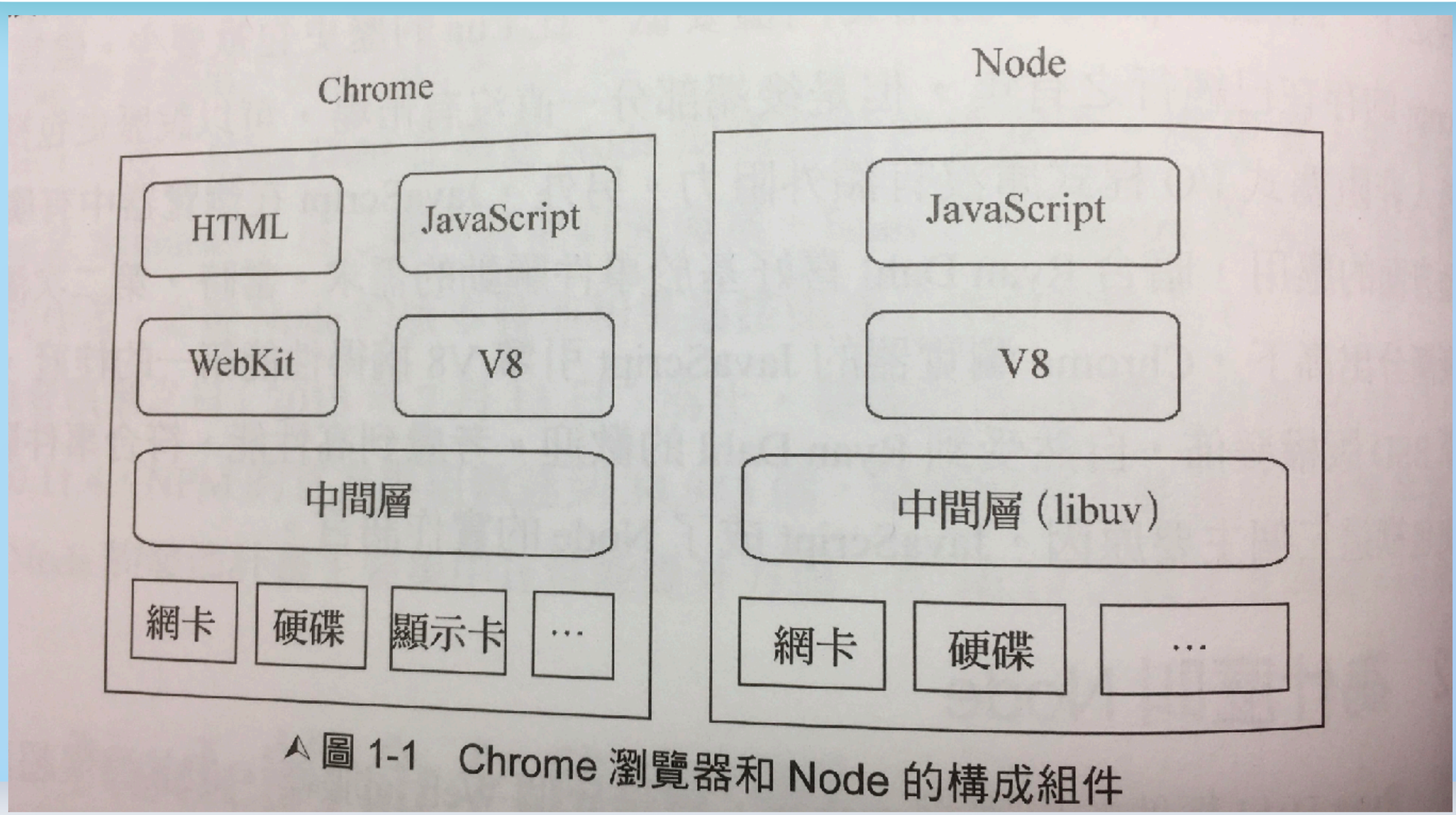


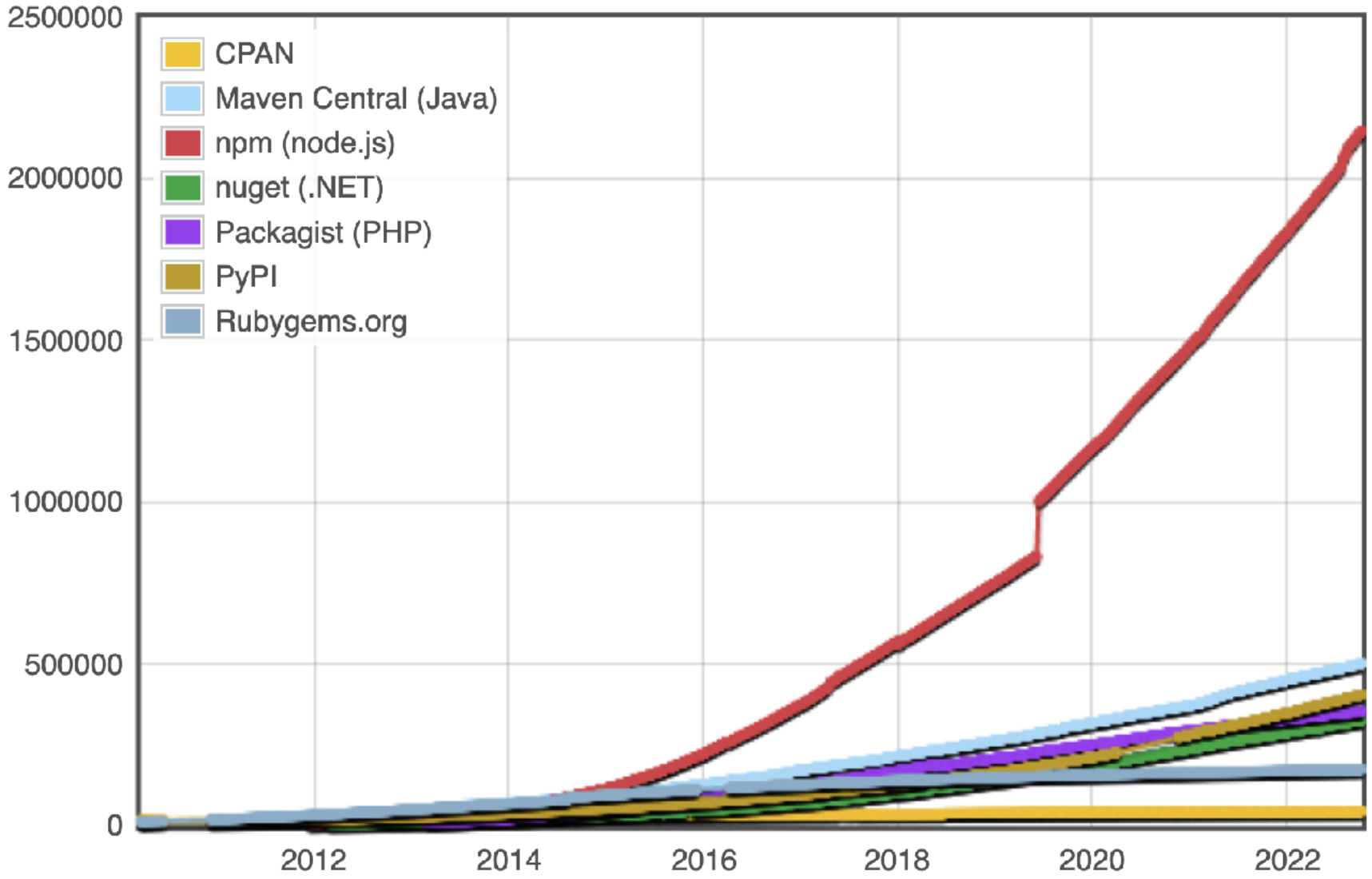
圖 1-1 Chrome 瀏覽器和 Node 的構成組件

Node.js 雖然有 .js, 但它並不是指某個單一的 JS 檔案

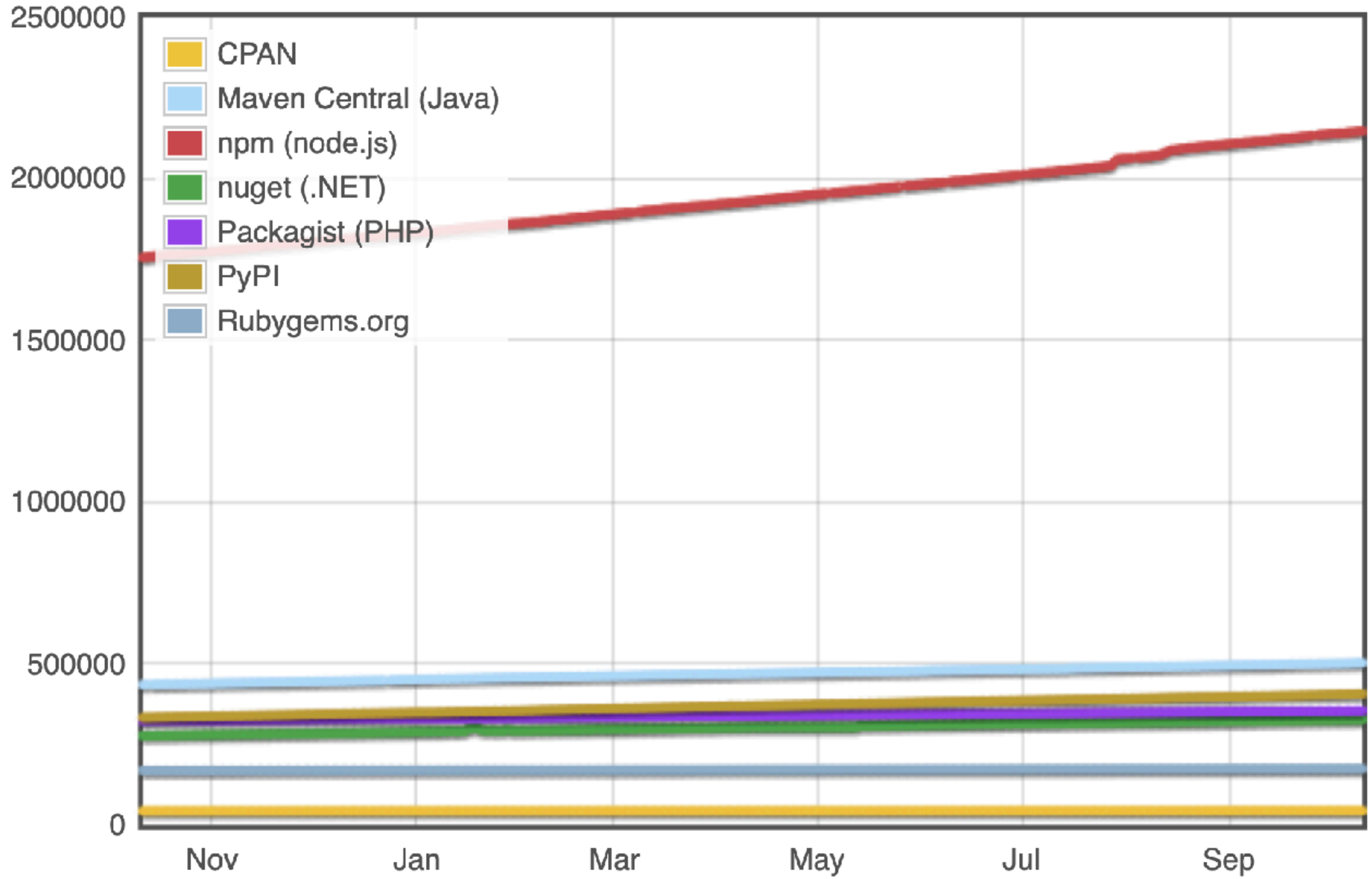
- 事實上，它是個 JS runtime environment, 允許新增各種用 JS 寫的功能模組 (modules)，包含了 file system I/O, networking, binary data (buffers), cryptography functions, data streams, etc.

- 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[68]	Status	Code name	Release date	Maintenance end
0.10.x	End-of-Life		2013-03-11	2016-10-31
0.12.x	End-of-Life		2015-02-06	2016-12-31
4.x	End-of-Life	Argon[69]	2015-09-08	2018-04-30
5.x	End-of-Life		2015-10-29	2016-06-30
6.x	End-of-Life	Boron[69]	2016-04-26	2019-04-30
7.x	End-of-Life		2016-10-25	2017-06-30
8.x	End-of-Life	Carbon[69]	2017-05-30	2019-12-31
9.x	End-of-Life		2017-10-01	2018-06-30
10.x	End-of-Life	Dubnium[69]	2018-04-24	2021-04-30
11.x	End-of-Life		2018-10-23	2019-06-01
12.x	End-of-Life	Erbium[69]	2019-04-23	2022-04-30
13.x	End-of-Life		2019-10-22	2020-06-01
14.x	Maintenance LTS	Fermium[69]	2020-04-21	2023-04-30
15.x	End-of-Life		2020-10-20	2021-06-01
16.x	Active LTS	Gallium[69]	2021-04-20	2023-09-11[70]
17.x	End-of-Life		2021-10-19	2022-06-01
18.x	**Current**		2022-04-19	2025-04-30
19.x	Planned		2022-10-18	2023-06-01
20.x	Planned		2023-04-18	2026-04-30

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 asynchronous 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 asynchronous 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

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}`);
});
```

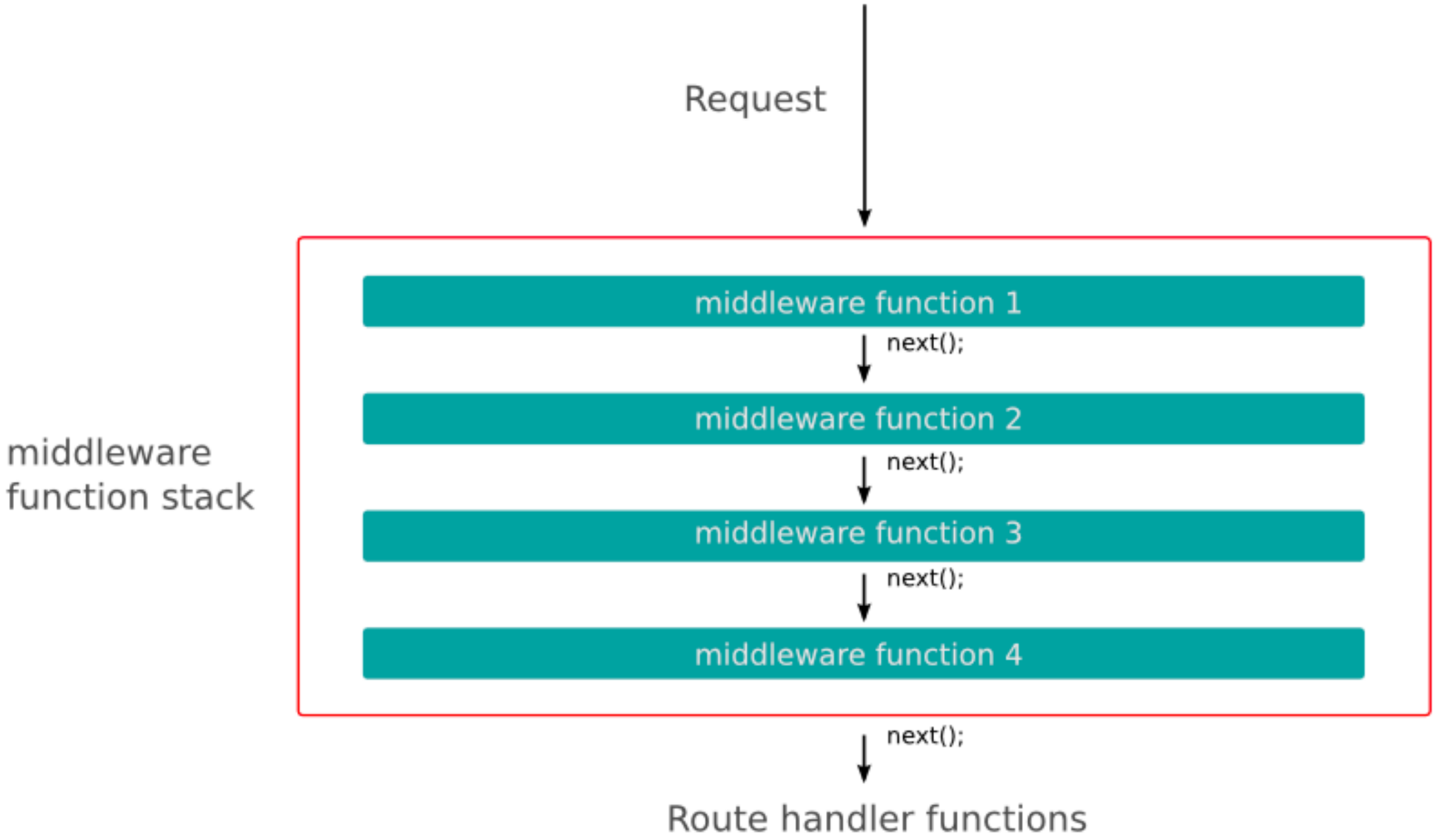
- 當然，你也可以直接用 node 的 command line.
- 只要在 terminal 鍵入 "node", 你就可以試用各種 Node.js 的指令與模組了！ (Use .help to get help)

- Node.js 是一個 JS 的 runtime environment, 我們在寫後端時，常常會採用一些包裝好，包含許多 middleware 的 "web framework" 來進行開發，例如: Express.js

【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



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.
```

【Client Side】

In "frontend/src"

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

yarn start

【Server Side】

In "backend/"

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

yarn server

Backend。用 Express 來建立 HW#6 的 Server

- backend/server.js

```
import express from 'express'
import cors from 'cors'
import guessRoute from './routes/guess'

const app = express()

// init middleware
app.use(cors())

// define routes
app.use('/api/guess', guessRoute)

// define server
const port = process.env.PORT || 4000
app.listen(port, () => {
  console.log(`Server is up on port ${port}.`)
})
```

Types of Middlewares in Express

- Application-level middleware
- Router-level middleware

最常用

- Error-handling middleware
- Built-in middleware
- Third-party middleware

Application-level middleware

```
import express from '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

```
const app = express()

// init middleware
app.use(cors())
```

- 如果沒有“app.use(cors())”會怎麼樣？

```
Access to XMLHttpRequest at 'http://localhost:4000/
api/guess/start' from origin 'http://
localhost:3000' has been blocked by CORS policy: No
'Access-Control-Allow-Origin' header is present on
the requested resource.
```

如果指定 path, 則只有 routing path 符合的時候才會被執行

```
// define routes
app.use('/api/guess', guessRoute)
```

- 定義了 baseURL: 'http://localhost:4000/api/guess' 這個 API 是由 guessRoute() 這個 callback function 來服務的

可以有多個 callback functions

- 用 "next()" 來串起多個同樣 URL 的 callback functions

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

app.get('/user/:id', function (req, res, next) {
  res.end(req.params.id)
})
```

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')
})
```

Use Router-level middleware in HW#5

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

```
import express from 'express'
const router = express.Router()
router.post('/start', (_, res) => {
  genNumber()  // 用亂數產生一個猜數字的 number，存在 memory DB
  res.json({ msg: 'The game has started.' })
})
router.get('/guess', (req, res) => {
  // 去 (memory) DB 拿答案的數字
  // 用 req.query.number 拿到前端輸入的數字
  // check if NOT a num or not in range [1,100]
  // 如果有問題 =>
  // res.status(406).send({ msg: 'Not a legal number.' })
  // 如果沒有問題，回傳 status
router.post('/restart', (_, res) => { ... })
export default router
```

再用 application-level middleware 指定 routing

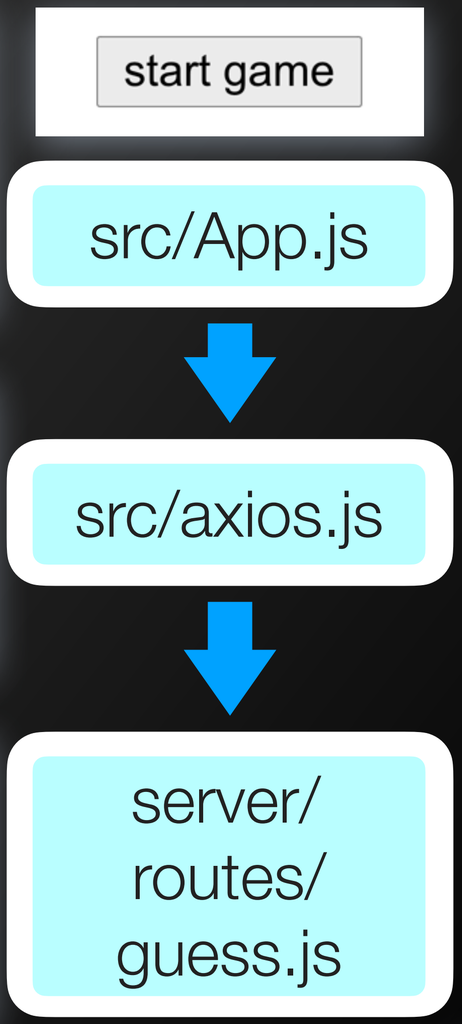
```
import express from 'express'
import guessRoute from './routes/guess'
const app = express()
app.use('/api/guess', guessRoute);
```

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



補充：Auto update with "nodemon"

- 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

```
"server": "nodemon server.js”
```

- 然後重啟 server by "yarn server”

補充：Compile 時遇到不同版本 JS/Browser 的問題

- 我們常常在 import / require 了 3rd party 的套件之後，常常會遇到各種因為不同版本 JS/Browser 的問題，像這樣：

```
Error [ERR_MODULE_NOT_FOUND]: Cannot find module ‘xxx’
imported from /qqq/yyy.js
```

```
(node:75390) Warning: To load an ES module, set “type”:
“module” in the package.json or use the .mjs extension.
```

```
require('dotenv-defaults').config();
^
ReferenceError: require is not defined
    at file:...
```

類似這樣的情況，十之八九都是因為 import 的 modules 版本雜亂，彼此並不相容，因此才需要 google 各種 workarounds，但這樣除了會讓 code 變醜之外，常常也會有挖東牆補西牆，最後還是一直遇到問題的情況。

一個根本的解決之道就是好好的使用 Babel

Babel — The Modern JavaScript Transcompiler

- (From Wiki) Babel is a free and open-source JavaScript transcompiler that is mainly used to convert ECMAScript 2015+ (ES6+) code into a backwards compatible version of JavaScript that can be run by older JavaScript engines.
- It was originally created by Sebastian McKenzie at age of 17 (2015). He then joint Facebook at 18. He was also the author of yarn.
- Read this for more of his story.

Adopting Babel in your project

- 妥善安裝 Babel 相關套件

```
# -D to install in dev mode
yarn add -D @babel/cli @babel/core @babel/node @babel/preset-env
```

- 增加一個 .babelrc 的設定檔

```
{
  "presets": [
    "@babel/preset-env"
  ]
}
```

- 修改 package.json:

```
  "scripts": {
    "server": "nodemon server.js --ext js --exec babel-node"
  },
```

For more information about Babel, please refer to:

“Babel 用得好，Compile 沒煩惱”

HW#5 TODOs

- Client side
  1. App.js 產生畫面以及管理 states
  2. axios.js 定義三個 { startGame, guess, restart } 與後端溝通的 functions
- Server side
  1. 沒有用 DB/file, 而是直接在 core/getNumber.js 產生 0-99 間的隨機亂數 (as a global variable in-memory DB)
  2. routers/guess.js：定義 { "/start", "/guess", "/restart" } 三個 APIs
  3. server.js 定義 express server

感謝聆聽！

Ric Huang / NTUEE

(EE 3035) Web Programming