Hard Parts: Servers & Node.js

Will Sentance

Academic work:
Oxford, Harvard

Currently:
CEO & Cofounder Codesmith
Frontend Masters

Previously:
Cocreator @Icecomm
Software Engineer @Gem

The power of Node

- Most powerful technology in web development to emerge in 10 years
- Enables applications that can handle millions of users without blocking
- From simple webpages to largest scaled applications, to Windows/Mac desktop apps (with Electron), and hardware (embedded systems)
- Allows us to build entire applications end-to-end in one language - JavaScript

From client side development to full stack development

Our users open `twitter.com` - they need code and data to load `twitter.com` on their computers

- What code/data do they need to load?
- Where's the code/data coming from?

Servers are the behind-the-scenes of all web applications - where our client-side code/data comes from

Computer connected to the internet - a permanent store of code/data, always on, ready to receive messages over the internet from users requesting code/data and send it back

— How’s this computer know what to send back?
— What languages can we use to write these instructions?

But how can we access these inbound messages as developers and send code/data back in response?

Sending the right data back requires using multiple features of the computer

- Network socket - Receive and send back messages over the internet
- Filesystem - that’s where the html/css/javascript code is stored in files
- CPU - for cryptography and optimizing hashing passwords
- Kernel - I/O management

Our dream - be able to use JavaScript to control this computer because (1) we know JavaScript and (2) it has some really nice design decisions

Each programming language (PHP, Ruby, C++, JavaScript) have different levels of ability to interact with these features directly

C++ has many features that let it directly interact with the OS directly

JavaScript does not! So it has to work with C++ to control these computer features. What is this combination known as? … Node.js

JS -> Node -> Computer feature (e.g. network, file system)

Rewind. We had better understand JavaScript to understand Node.js then

It’s a language that does 3 things (and 1 involves a lot of help from C++)

1. Saves data and functionality (code)
2. Uses that data by running functionality (code) on it
3. Has a ton of built-in labels that trigger Node features that are built in C++ to use our computer’s internals

Let’s see the 2 things that JS does by itself - saving and using data

```
let num = 3;
// 1. Save a function (code to run, parameters awaiting inputs)
function multiplyBy2 (inputNumber){
  const result = inputNumber*2;
  return result;
}

// 2a. Call/run/invoke/execute a function (with parens)
// and 2b. insert an input (an argument)
const output = multiplyBy2(num);
const newOutput = multiplyBy2(10);
```

So let’s see JavaScript other talent - built-in labels that trigger Node features

We can set up, with a JavaScript label, a Node.js feature (and so computer internals) to wait for requests for html/css/js/tweets from our users

How? The most powerful built-in Node feature of all: `http` (and its associated built-in label in JS - also `http` conveniently)

Using http feature of Node to set up an open socket

```
const server = http.createServer()
server.listen(80)
```

Inbound web request -> run code to send back message

```
if inbound message -> send back data
```

But at what moment? 🥴

Node auto-runs the code (function) *for us* when a request arrives from a user

```
function doOnIncoming(incomingData, functionsToSetOutgoingData){
    functionsToSetOutgoingData.end("Welcome to Twitter!")
}

const server = http.createServer(doOnIncoming)
server.listen(80)
```

1. We don’t know when the inbound request would come - we have to rely on Node to trigger JS code to run
2. JavaScript is single-threaded & synchronous. All slow work (e.g. speaking to a database) is done by Node in the background (more on this later)

Two parts to calling a function - executing its code and inserting input (arguments)

In `multiplyBy2(3)` the argument is 3 and we, the developer, inserted it

Node not only will auto-run our function at the right moment, it will also automatically insert whatever the relevant data is as the additional argument (input)

Sometimes it will even insert a set of functions in an object (as an argument) which give us direct access to the message (in Node) being sent back to the user and allows us to add data to that message

And that’s exactly what Node does with its http feature

Node inserts the arguments (inputs) automatically in the function it auto-runs:

1. ‘Inbound object’ - all data from the inbound (request) message
2. ‘Outbound object’ - functions to be used to set outbound (response) message data

These objects (the arguments to the auto-run function) aren’t given labels by Node. So how do we access them? We do so with parameters (placeholders).

We must make sure to format the function Node auto-runs the way Node expects (use docs)

Code again

```
function doOnIncoming(incomingData, functionsToSetOutgoingData){
    functionsToSetOutgoingData.end("Welcome to Twitter!")
}

const server = http.createServer(doOnIncoming)
server.listen(80)
```

People often end up using req and res for the parameters...

Let’s get more personalized with what we send back to our user from our server

By writing code to investigate the inbound message to see exactly what she’s asked for

Our user, needs a specific tweet (tweet 3) back. How does their browser tell us that?

Messages are sent in HTTP format - The ‘protocol’ for browser-server interaction

HTTP message: Request line (url, method), Headers, Body (optional)

```
const tweets = ["Hi", "😂", "Hello", "👋", "👻"]
function doOnIncoming(incomingData, functionsToSetOutgoingData){
    const tweetNeeded = incomingData.url.slice(8)-1
    functionsToSetOutgoingData.end(tweets[tweetNeeded])
}

const server = http.createServer(doOnIncoming)
server.listen(80)
```

Our return message is also in HTTP format

We can use the body to send the data and `headers` to send important metadata

In the headers we can include info on the format of the data being sent back - e.g. it’s `html` so to load it as a webpage

Getting access to Node’s built in features with `require`

We have to tell Node we want to have access to each of its C++ features independently - we get a built-in function to do this
`require`

```
const http = require(‘http’);
```

How do we start Javascript off to do all this?

1. Write the code (VSCode et al)
2. Load it into Node and run it (have to load in using the terminal interface)
3. Need to reload our code with Node every time we make a change so nodemon

Do we need an always-on computer in our house to run a server?

1. Write code on your computer
2. SSH into someone else’s computer (one of AWS’s)
3. Set up DNS to match domain name to right IP

But what about testing our server?

Do you need to load the code to be run on an AWS computer?

OS developers included the loopback feature with `localhost` as the pseudo-domain

This is what you will be doing in the pair-programming

Pair-programming

In server side development we get errors

Understandable - we’re interacting with others’ computers over the internet - there’s lots of issues that could arise

How can we handle this? We need to understand our background Node http server feature better

What triggers the `doOnIncoming` function to run? Events

- `http.createServer(doOnIncoming)` is actually a one-line version of setting up the server that will auto-release (‘emit’) events (‘messages in Node’) that trigger a function to auto-run if we’ve set one
- These events are preset on http: ‘request’, ‘error’

Node will automatically send out the appropriate event depending on what it gets from the computer internals (http message or error 😭)

```
function doOnIncoming(incomingData, functionsToSetOutgoingData){
    functionsToSetOutgoingData.end("Welcome to Twitter")
}

function doOnError(infoOnError){
    console.error(infoOnError)
}

const server = http.createServer();
server.listen(80)

server.on('request', doOnIncoming)
server.on('clientError', doOnError)
```

We have much of our twitter app set up now - handling, inspecting and responding to these messages (‘requests’) is the core of our app, of Node, and of servers

- But, Node can do even more. We have an archive of tweets stored in a huge file (1.5GB)
- Unfortunately they’re saved on our computer, not in our little JavaScript-specific data store (JavaScript memory)
- Could we load them into JavaScript to run a function that removes bad tweets?
- We can use fs to do so but there might be some issues with a file that large

Importing our tweets with fs

```
function cleanTweets (tweetsToClean){
  // code that removes bad tweets
}

function useImportedtweets(errorData, data){
    const cleanedTweetsJson = cleanTweets(data);
    const tweetsObj = JSON.parse(cleanedTweetsJson)
    console.log(tweetsObj.tweet2)
}

fs.readFile('./tweets.json', useImportedtweets)
```

— Every file has a ‘path’ (a link - like a domestic url)

— JSON is a javascript-ready data format

What if Node used the 'event' (message-broadcasting) pattern to send out a message ('event') each time a sufficient batch of the json datahad been loaded in

And at each point, take that data and start cleaning it - in batches

```
let cleanedTweets = "";

function cleanTweets (tweetsToClean){
  // algorithm to remove bad tweets from `tweetsToClean`
}

function doOnNewBatch(data){
    cleanedTweets += cleanTweets(data);
}

const accessTweetsArchive = fs.createReadStream('./tweetsArchive.json')

accessTweetsArchive.on('data', doOnNewBatch);
```

Introducing Event loop and Callback Queue

The call stack, event loop and callback queue in Node

- **Call stack**: JavaScript keeps track of what function is being run and where it was run from. Whenever a function is to be run, it's added to the call stack
- **Callback queue** - any functions delayed from running (and run automatically by Node) are added to the callback queue when the background Node task has completed (or there's been some activity like a request)
- **Event loop** - Determines what function/code to run next from the queue(s)

Bringing it all together

But Node is most powerful because of the automated JS function execution triggered by Node at just the right moment.

This means we don’t have to wait in JS for the right moment to run code and block any other code running

But it also means we better know intimately how Node decides what to automatically execute at what moment…

The event loop is very strict. What rules does it set for what code to run next and when it may run?

```
function useImportedtweets(errorData, data){
    const tweets = JSON.parse(data)
    console.log(tweets.tweet1)
}

function immediately(){console.log("Run me last 😭")}

function printHello(){console.log("Hello")}

function blockFor500ms(){
  // Block JS thread DIRECTLY for 500 ms
  // With e.g. a for loop with 5m elements
}

setTimeout(printHello,0)

fs.readFile('./tweets.json', useImportedtweets)

blockFor500ms()

console.log("Me first")
setImmediate(immediately)
```

Rules for the automatic execution of the JS code by Node

1. Hold each deferred function in one of the task queues when the Node background API ‘completes’
2. Add the function to the Call stack (i.e. execute the function) ONLY when the call stack is totally empty (Have the Event Loop check this condition)
3. Prioritize functions in Timer ‘queue’ over I/O queue, over setImmediate (‘check’) queue

Fin