

# Schedule

## **NodeJS:**

- Servers, generally
- NodeJS
- npm
- Express

<br/>

- Sending data to the server
- Returning JSON

# Lecture code

All lecture code is on FIT portal:

...

You will need to run the commands we show in lecture to run the server code!

Node installation instructions:

...

Servers

# Server-side programming

The type of web programming we have been doing so far in WPR is called "**client-side**" programming:

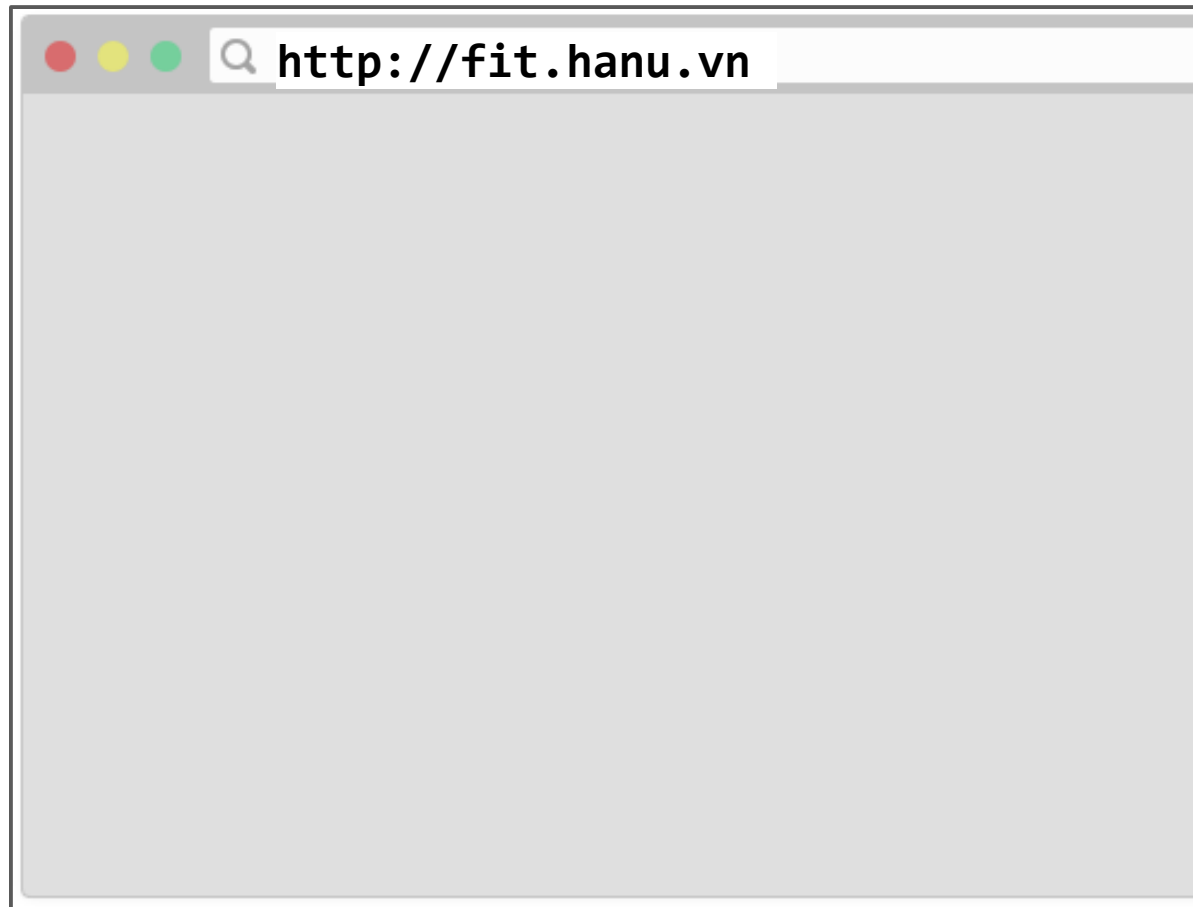
- The code we write gets run in a browser on the user's (client's) machine

Today we will begin to learn about **server-side** programming:

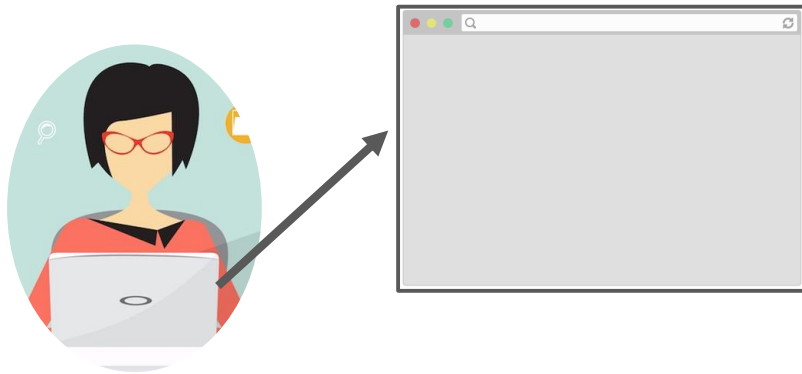
- The code we write gets run on a server.
- Servers are computers run programs to generate web pages and other web resources.

Let's take another look  
at how clients and servers work...

**CLIENT:** You type a URL in  
the address bar and hit  
"enter"



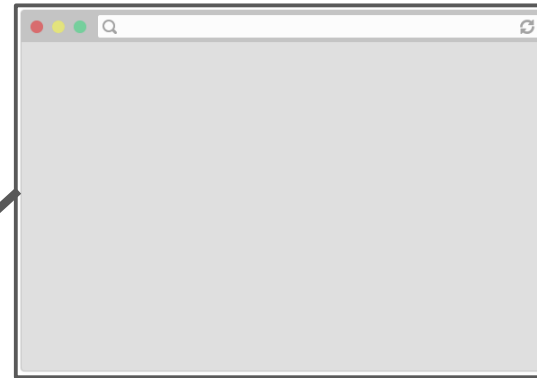
Browser sends an HTTP GET  
request saying "Please GET me  
the index.html file at  
`http://fit.hanu.vn`"



**Let's take a deeper  
look at this process...**

```
Host: fit.hanu.vn
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:70.0) Gecko/20100101 Firefox/70.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: en-US,en;q=0.5
Accept-Encoding: gzip, deflate
Connection: keep-alive
Cookie: _ga=GA1.2.1219603856.1567781253; hibext_instdsigdipv2=1; Mo
Upgrade-Insecure-Requests: 1
```

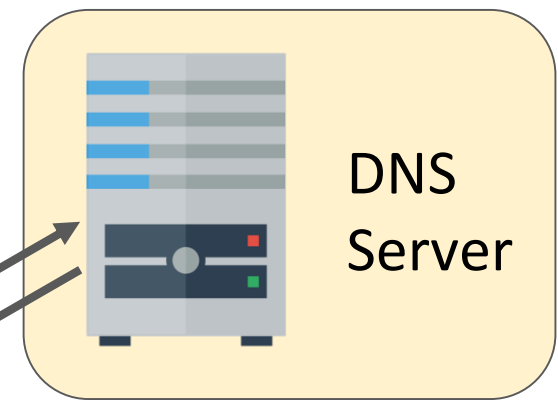
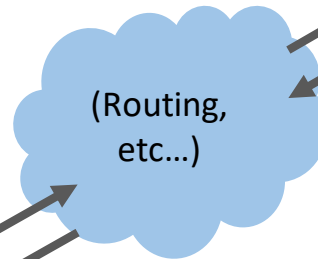
Browser C++ code creates an array of bytes that is formatted in using HTTP request message format



Browser asks operating system, "Hey, can you send this HTTP Get request message to <http://fit.hanu.vn>"?



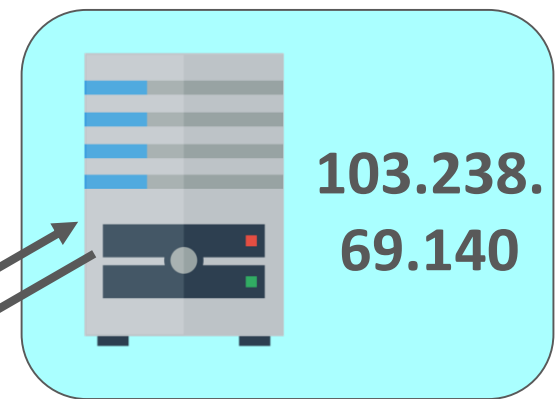
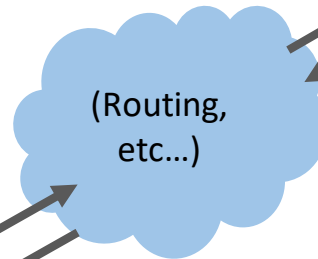
Operating system sends  
a **DNS** query to look up  
the **IP**  
**address** of  
"http://fit.hanu.vn"



DNS server replies with the  
**IP address**, e.g.  
103.238.69.140

- **DNS**: Domain Name System: Translate domain names to **IP address** of the computer associated with that address.
- **IP address**: Numerical unique identifier for every computer connected to the internet.

Operating system  
opens a **TCP**  
connection with the  
computer at  
103.238.69.140



After the **TCP** connection is  
established, the OS can send the  
HTTP message to 103.238.69.140  
*through* the **TCP** connection.

- **TCP**: Transmission Control Protocol, defines the data format for sending information over the wire. (Can be used for HTTP, FTP, etc)

103.238.69.140



**SERVER:** There is a computer that is connected to the internet at IP address 103.238.69.140

.

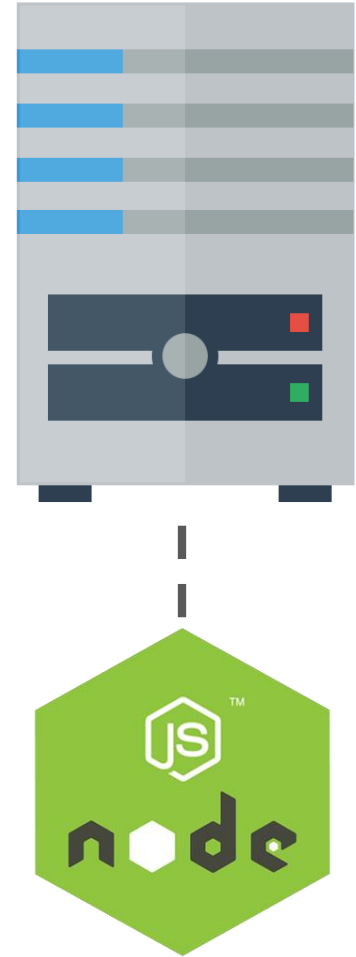
103.238.69.140

On this computer is a **web server program**:

- The web server program is **listening** for incoming messages that are sent to it.
- The web server program can **respond** to messages that are sent to it.

**Node:** The platform we will use to create a web server program that will receive and respond to HTTP requests.

- Also known as "**NodeJS**"; these terms are synonyms

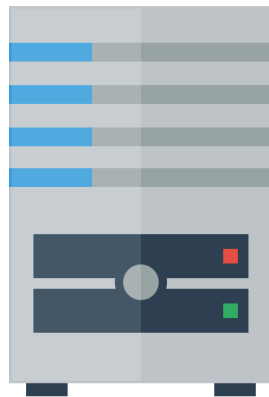


# Aside: "Server"

The definition of **server** is overloaded:

- Sometimes "server" means the machine/computer that runs the server software.
- Sometimes "server" means the software running on the machine/computer.

You have to use context to know which is being meant.



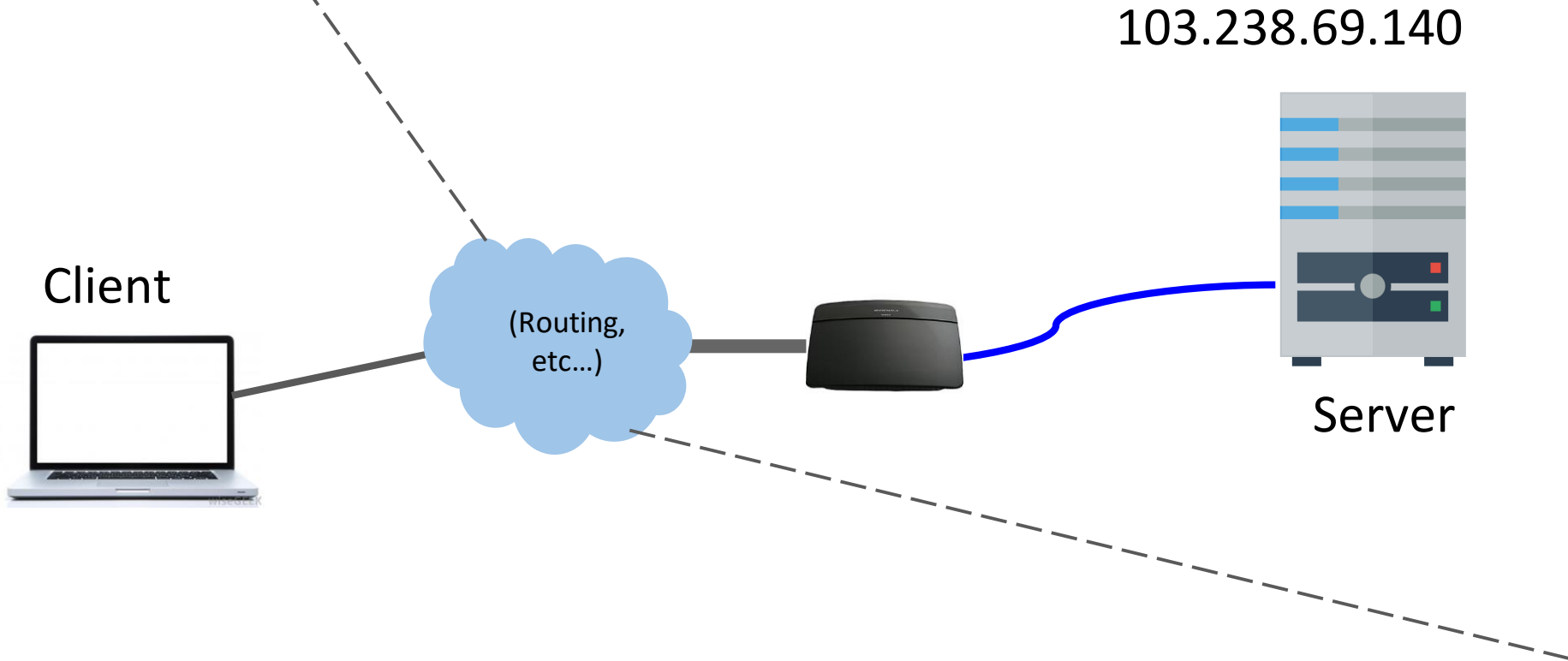
# Aside: Sockets

**Q: What does it mean for a program to be "listening" for messages?**

When the server first runs, it executes code to create a **socket** that allows it to receive incoming messages from the OS.

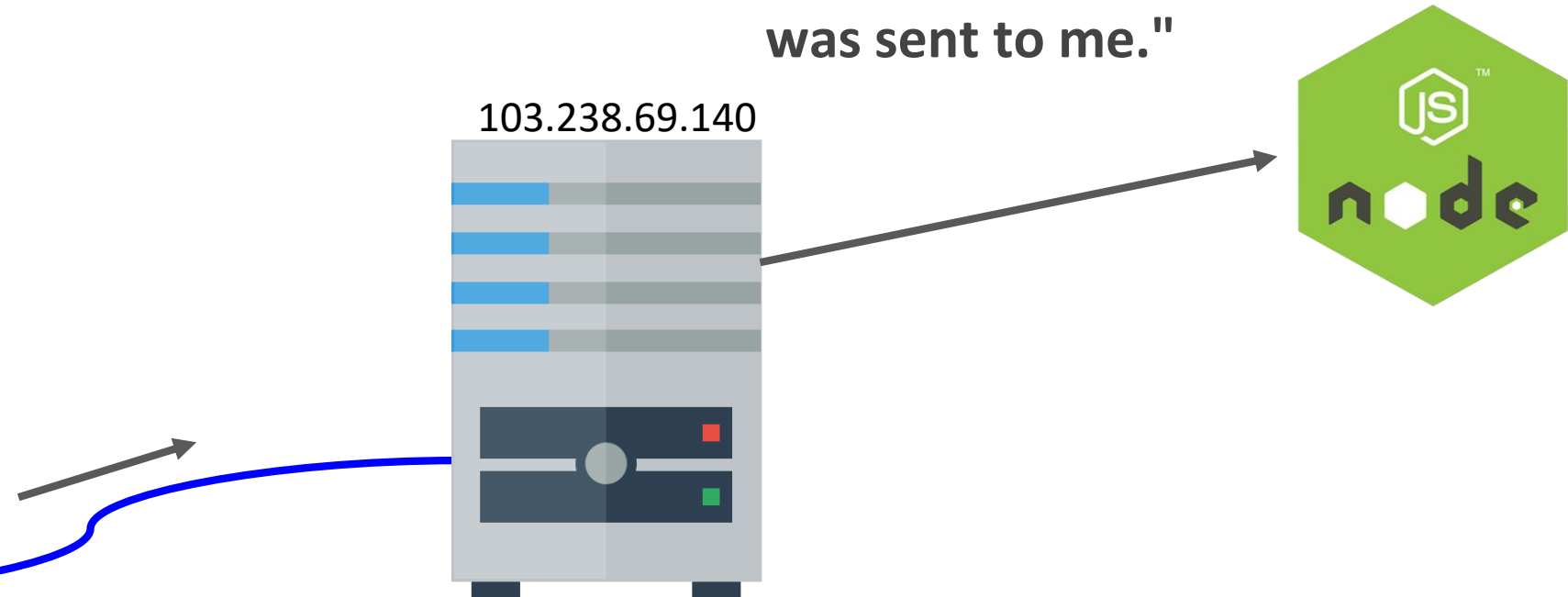
A [socket](#) is one end of a communication channel. You can send and receive data on sockets.

**However, NodeJS will abstract this away so we don't have to think about sockets.**



A TCP connection is established between the client and the server, so now the client and server can send messages directly to each other.

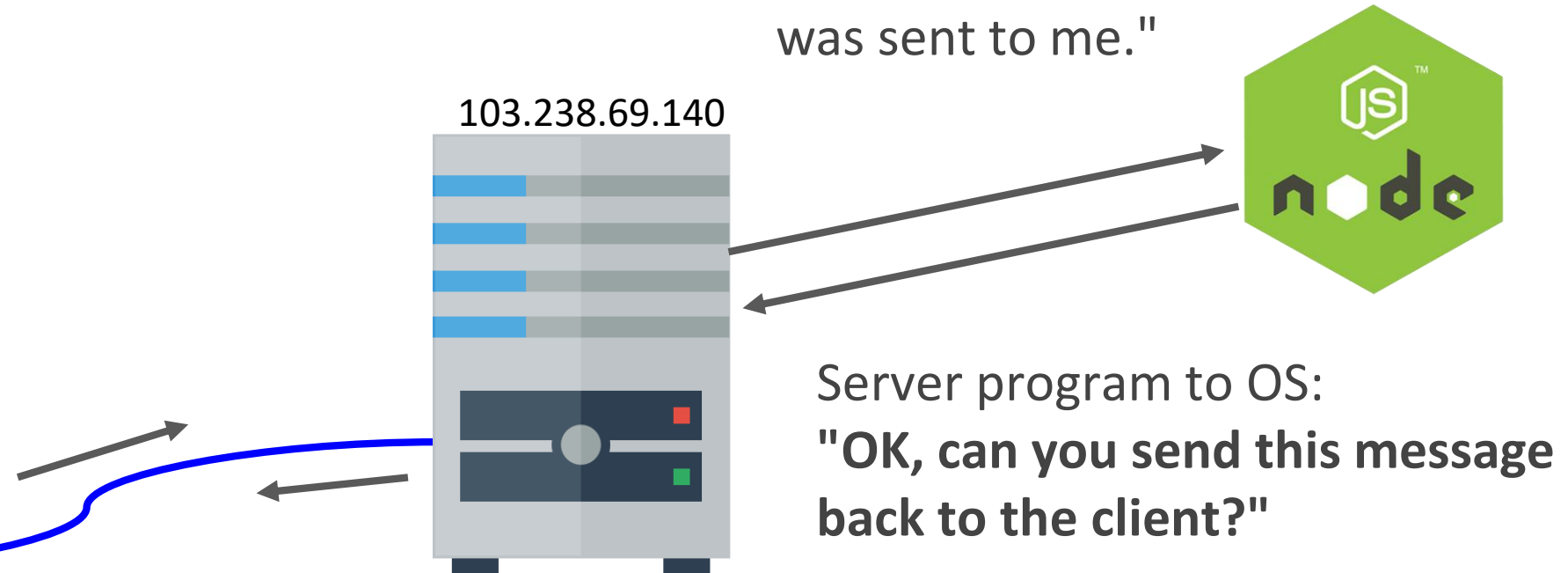
OS to server program:  
**"Hey, here's a message that  
was sent to me."**



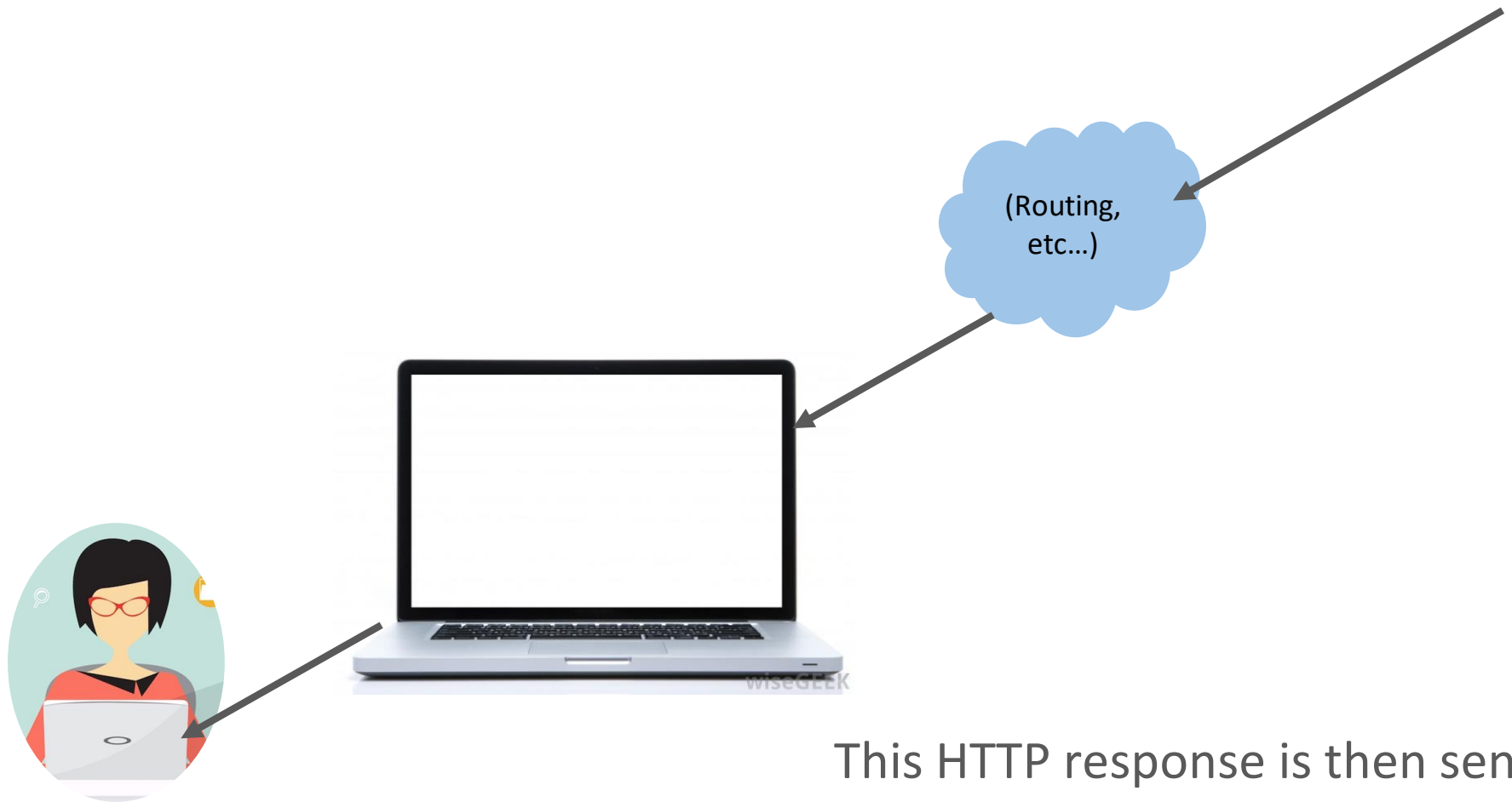
Now the operating system is receiving TCP packets from the wire, and the operating system begins sending the contents of the request to the server program.



OS to server program:  
"Hey, here's a message that  
was sent to me."



The server software parses the HTTP request and then decides what message it wants to send in response. It formats this message in HTTP, then asks the OS to send this response message over TCP back to the sender.



This HTTP response is then sent back to the client's OS, which notifies the browser of the HTTP response, and then the browser displays the web page.



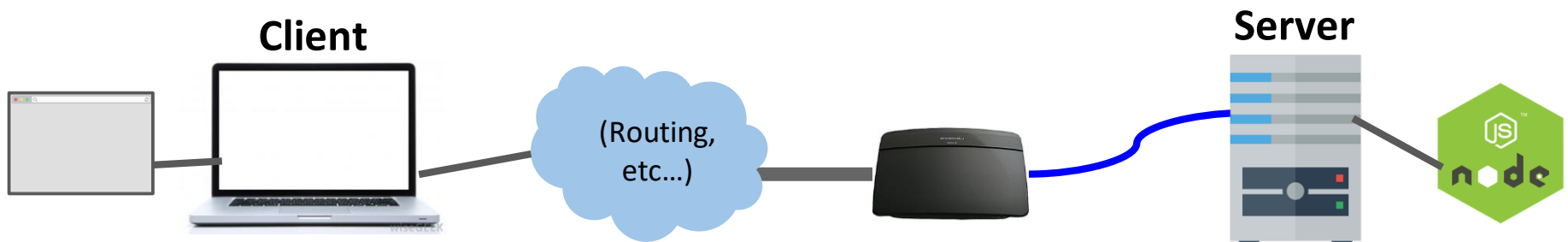
# Summary

When you navigate to a URL:

- Browser creates an HTTP GET request
- Operating system sends the GET request to the server over TCP

When a server computer receives a message:

- The server's operating system sends the message to the server software (via a socket)
- The server software then parses the message
- The server software creates an HTTP response
- The server OS sends the HTTP response to the client over TCP



NodeJS

# NodeJS

## **NodeJS:**

- A JavaScript runtime written in C++.
- Can interpret and execute JavaScript.
- Includes support for the NodeJS API.

## **NodeJS API:**

- A set of JavaScript libraries that are useful for creating server programs.

## **V8 (from Chrome):**

- The JavaScript interpreter ("engine") that NodeJS uses to interpret, compile, and execute JavaScript code

# NodeJS

## NodeJS:

- A JavaScript runtime written in C++.
- Can interpret and execute JavaScript.
- Includes support for the NodeJS API.

**Q: What does this mean?**

## NodeJS API:

- A set of JavaScript libraries that are useful for creating server programs.

## V8 (from Chrome):

- The JavaScript interpreter ("engine") that NodeJS uses to interpret, compile, and execute JavaScript code

# First: Chrome



## Chrome:

- A browser written in C++.
- Can interpret and execute JavaScript code.
- Includes support for the DOM APIs.

## DOM APIs:

- JavaScript libraries to interact with a web page

## V8:

- The JavaScript interpreter ("engine") that Chrome uses to interpret, compile, and execute JavaScript code

# Chrome, V8, DOM



Parser

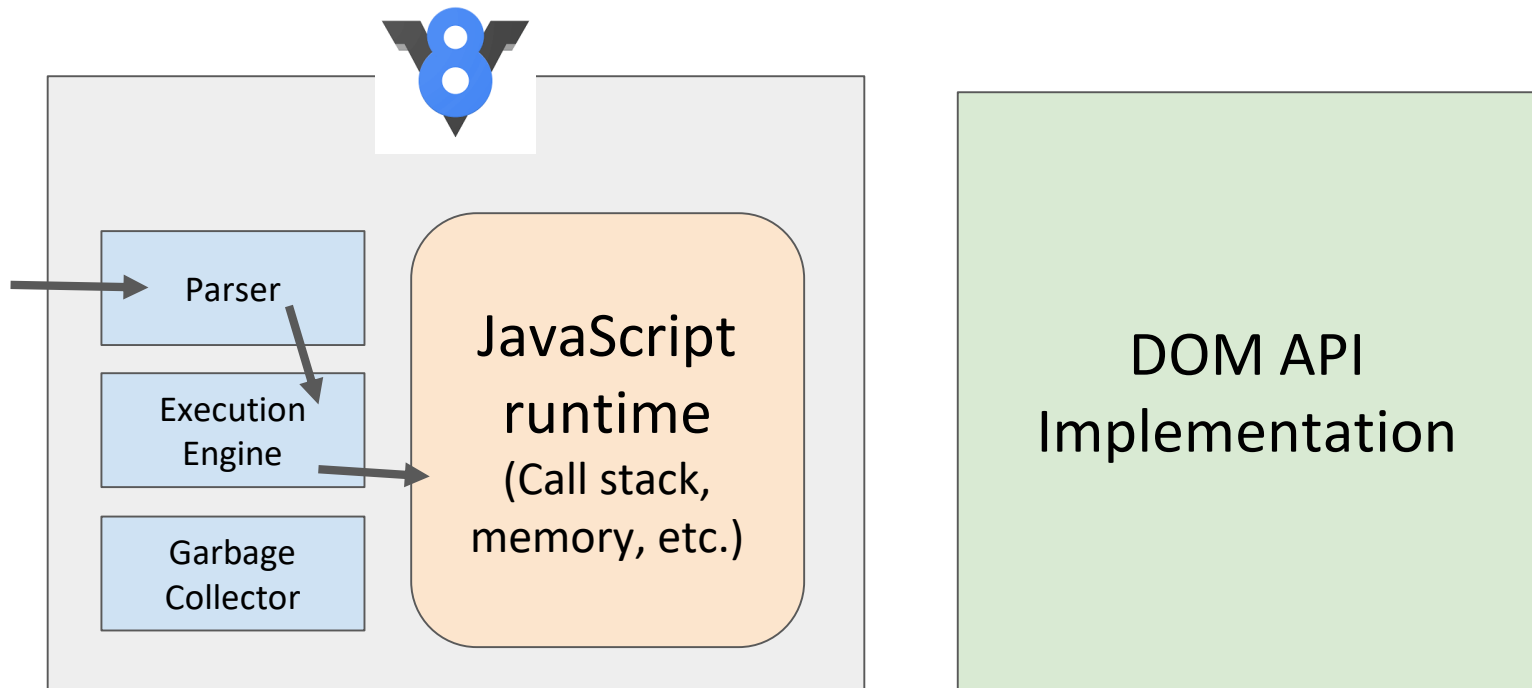
Execution  
Engine

Garbage  
Collector

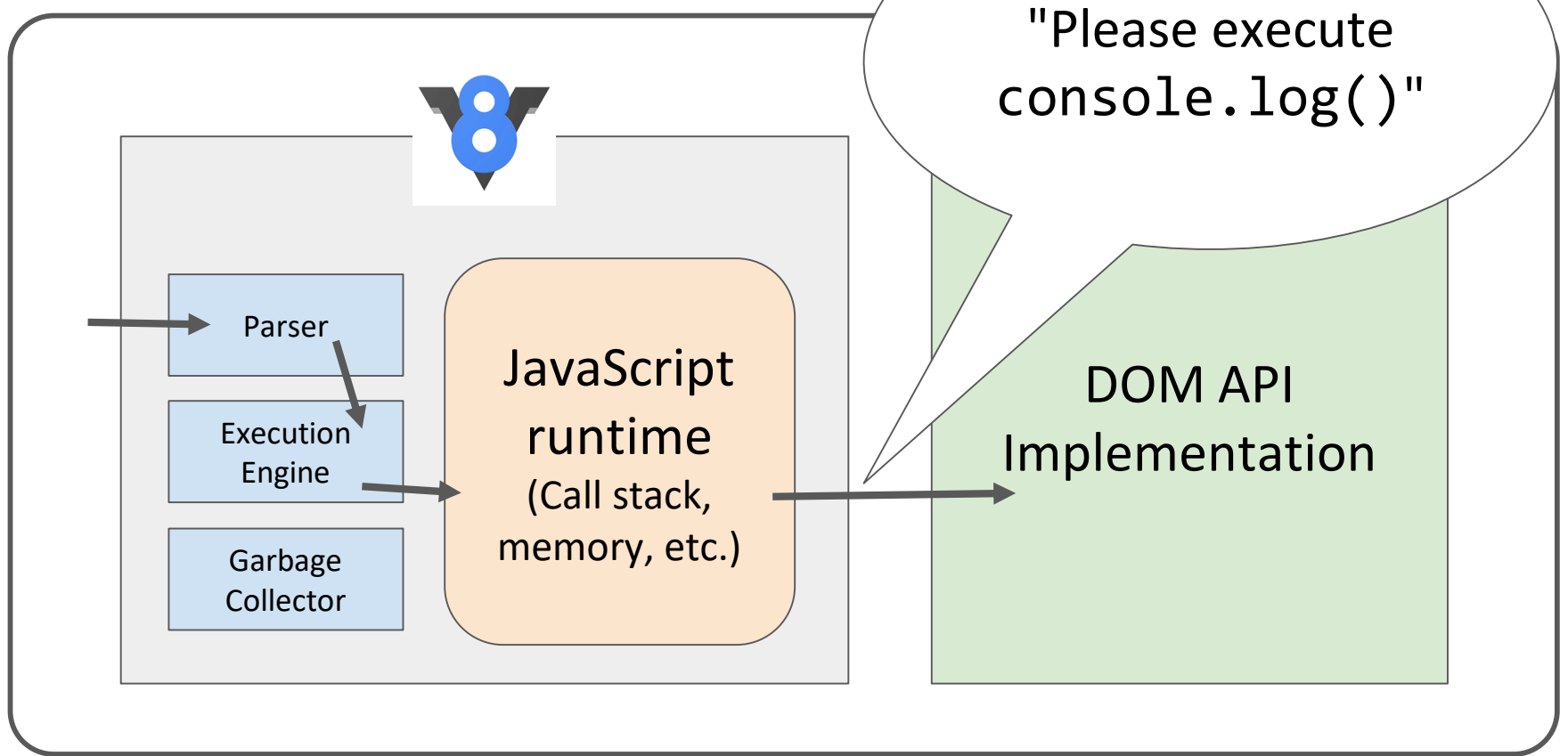
JavaScript  
runtime  
(Call stack,  
memory, etc.)

DOM API  
Implementation



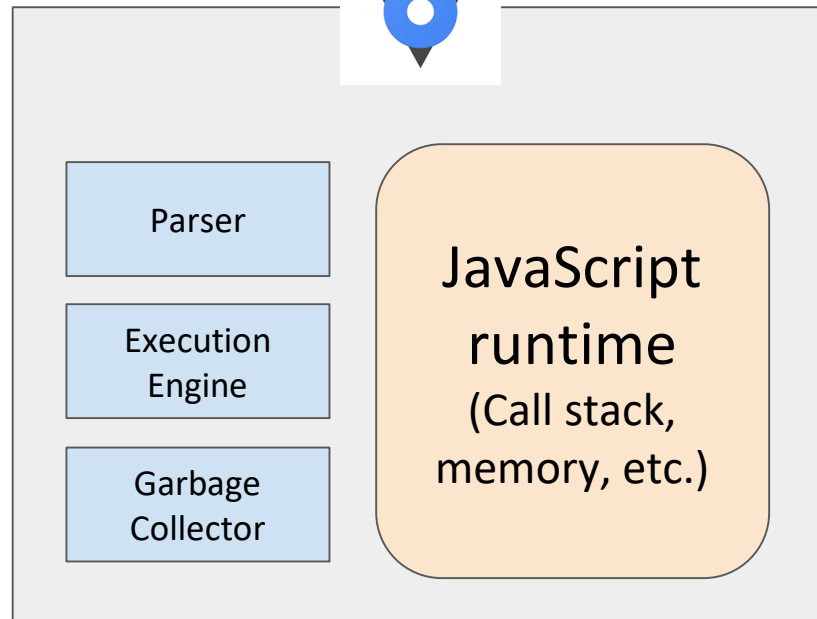


```
const name = 'V8';
```

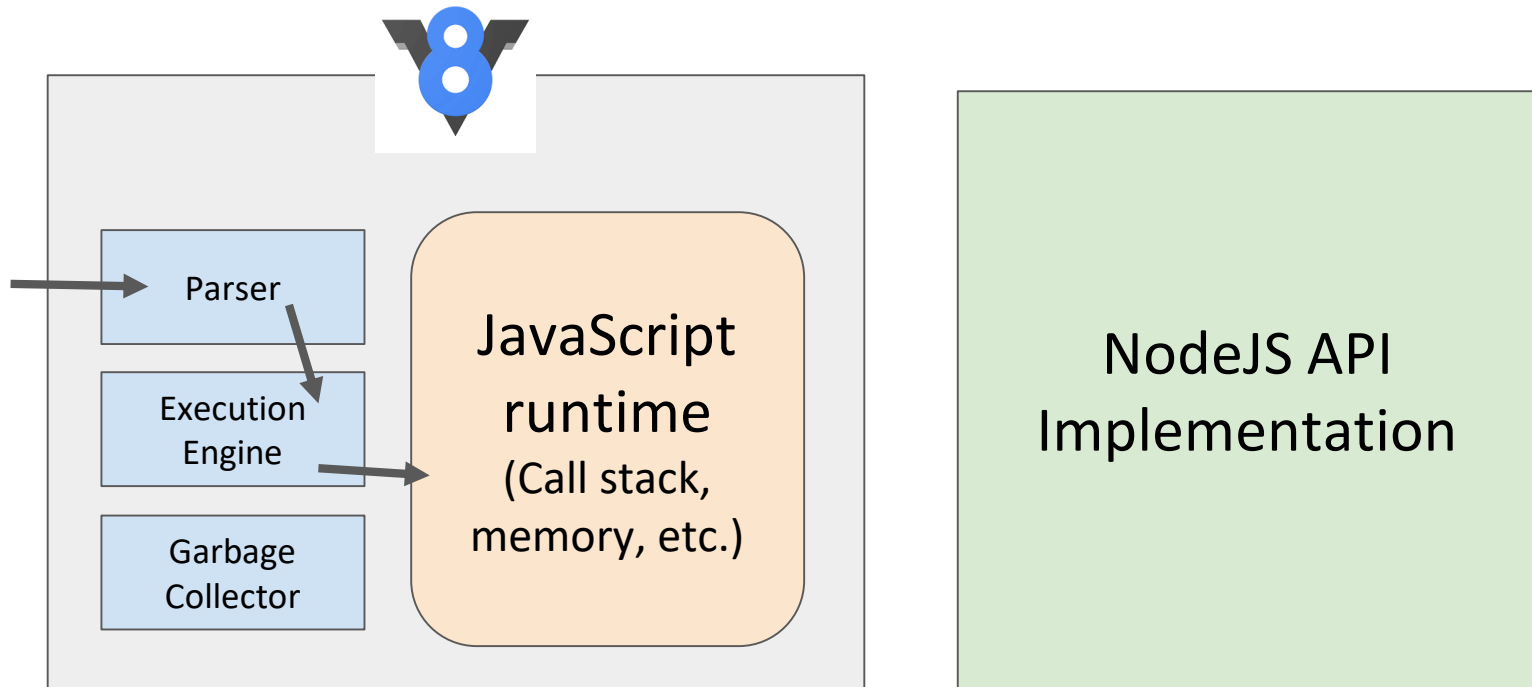


```
console.log('V8');
```

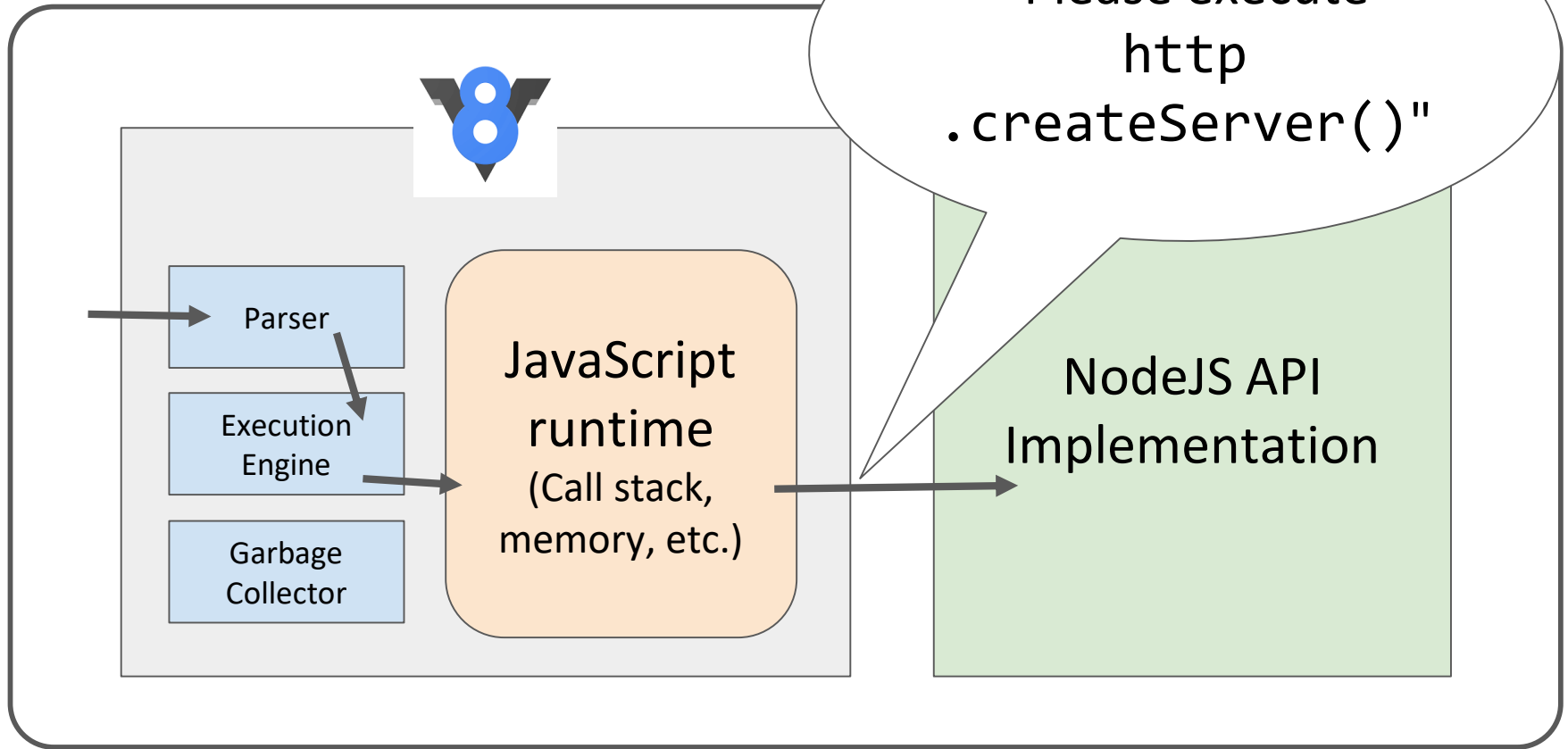
# NodeJS, V8, NodeJS APIs



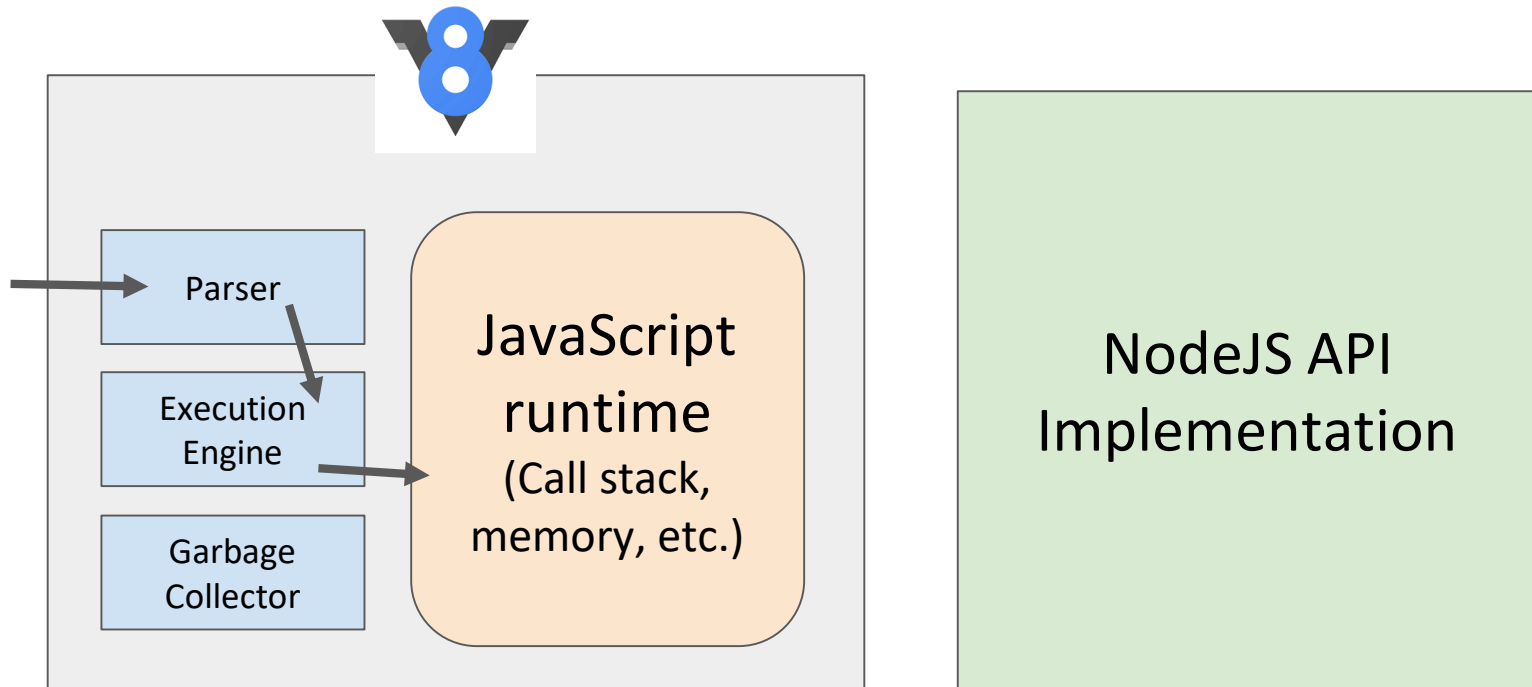
**NodeJS API  
Implementation**



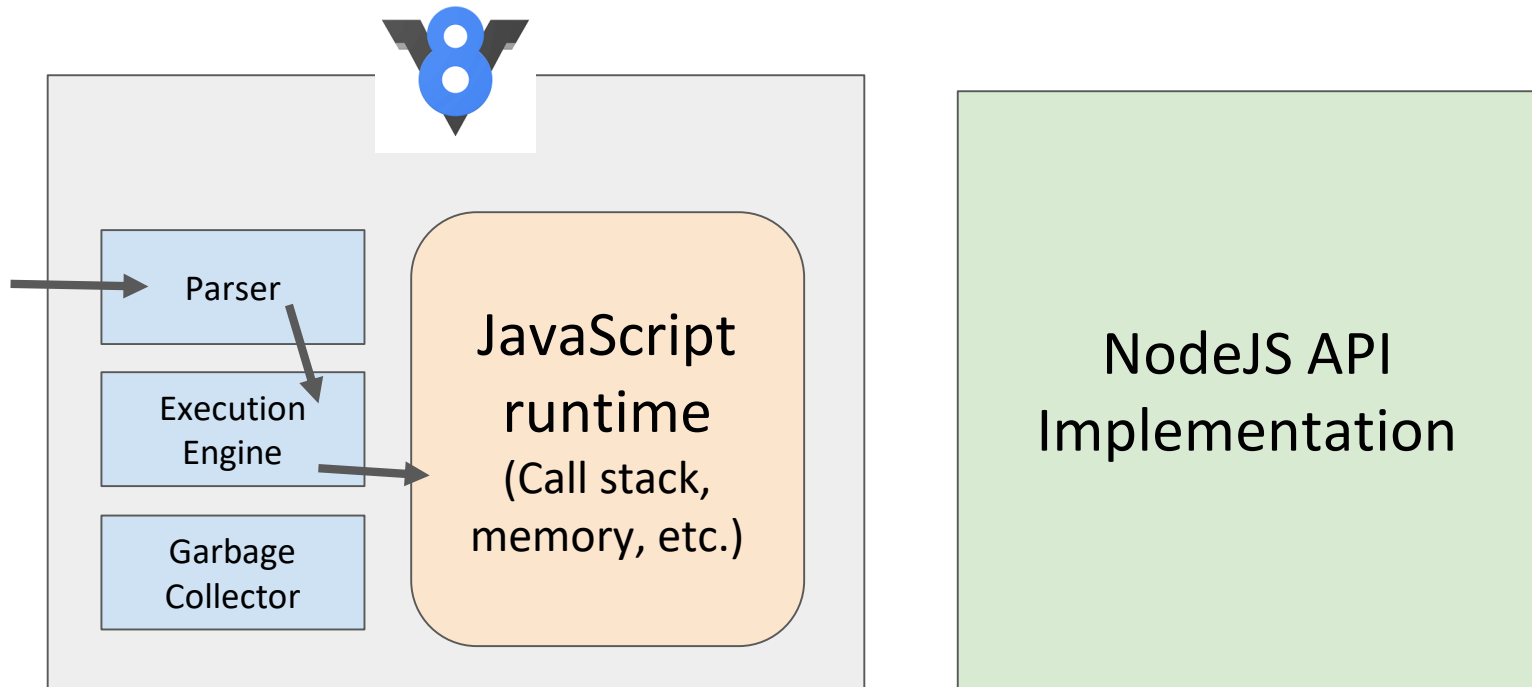
```
const x = 15;  
x++;
```



```
http.createServer();
```

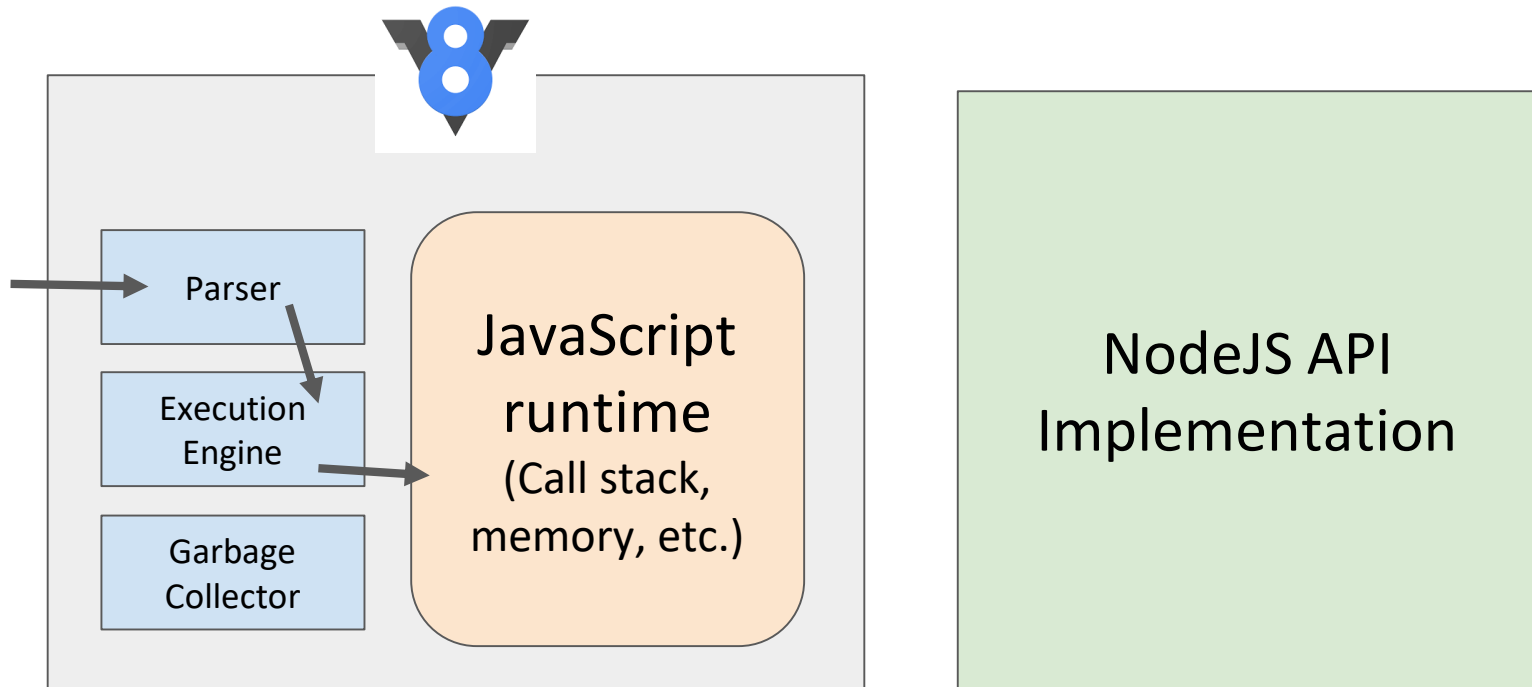


What if you tried to call  
`document.querySelector('div');`  
in the NodeJS runtime?



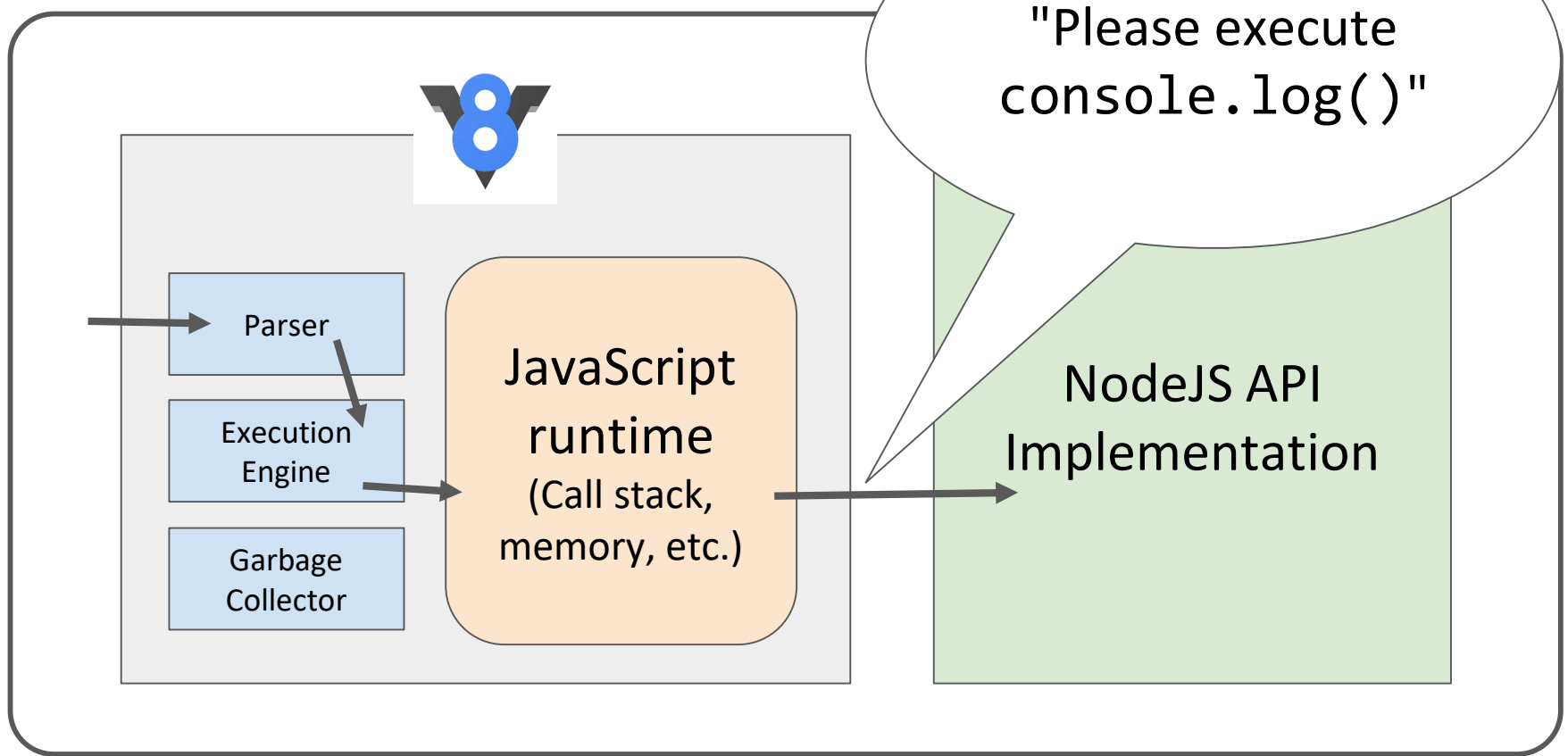
```
document.querySelector('div');
```

**ReferenceError: document is not defined**



What if you tried to call `console.log('nodejs');` in the NodeJS runtime?





```
console.log('nodejs');
```

(NodeJS API [implemented their own console.log](#))

# NodeJS

## **NodeJS:**

- A JavaScript runtime written in C++.
- Can interpret and execute JavaScript.
- Includes support for the NodeJS API.

## **NodeJS API:**

- A set of JavaScript libraries that are useful for creating server programs.

## **V8 (from Chrome):**

- The JavaScript interpreter ("engine") that NodeJS uses to interpret, compile, and execute JavaScript code

# Installation

**NOTE:** The following slides assume you have already installed NodeJS.

NodeJS installation instructions:

...

# node command

Running node without a filename runs a REPL loop

- Similar to the JavaScript console in Chrome, or when you run "python"

```
$ node
```

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

```
undefined
```

```
> x++
```

```
5
```

```
> x
```

```
6
```

# NodeJS

NodeJS can be used for writing scripts in JavaScript, completely unrelated to servers.

`simple-script.js`

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

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

# node command

The node command can be used to execute a JS file:

```
$ node fileName
```

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

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

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

# Node for servers

Here is a very basic server written for NodeJS:

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

const server = http.createServer();

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

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

server.listen(3000);
```

**(WARNING: We will **not** actually be writing servers like this!!!)**

We will be using ExpressJS to help, but we haven't gotten there yet.

# require()

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

The NodeJS `require()` statement loads a module, similar to `import` in Java or `include` in C++.

- We can `require()` modules included with NodeJS, or modules we've written ourselves.
- In this example, 'http' is referring to the [HTTP NodeJS module](#)



# require()

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

The `http` variable returned by `require('http')` can be used to make calls to the HTTP API:

- `http.createServer()` creates a Server object

# Emitter.on

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

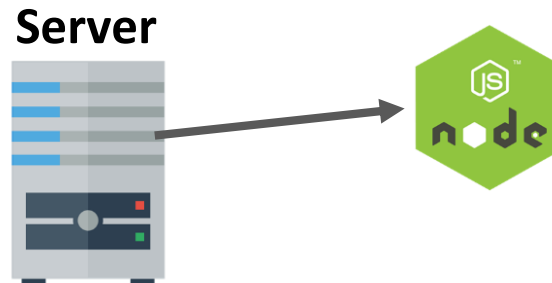
```
server.on('listening', function() {  
  console.log('Server running!');  
});
```

The [on\(\)](#) function is the NodeJS equivalent of `addEventListener`.

# Emitter.on

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

The [request](#) event is emitted each time there is a new HTTP request for the NodeJS program to process.



# Emitter.on

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

The [req](#) parameter gives information about the incoming request, and the [res](#) parameter is the response parameter that we write to via method calls.

- [statusCode](#): Sets the HTTP status code.
- [setHeader\(\)](#): Sets the HTTP headers.
- [end\(\)](#): Writes the message to the response body then signals to the server that the message is complete.

# listen() and listening

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

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

The listen() function will make the program start accepting messages sent to the given **port number**.

- The listening event will be emitted when the server has been bound to a port.

**Q: What's a port? What is binding?**

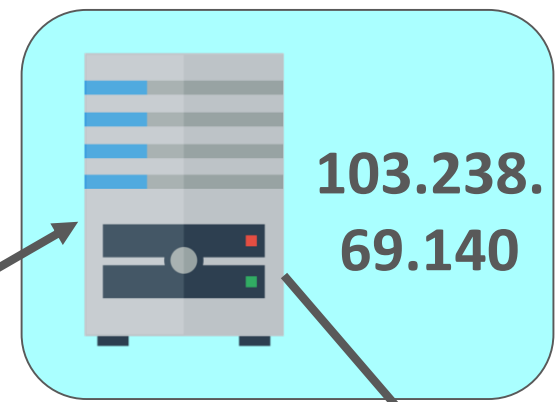
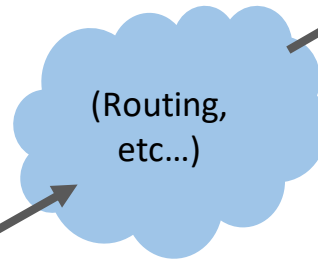
# Ports and binding

**port:** In the context of networking, a "logical" (as opposed to a physical) connection place

- A number from 0 to 65535 (16-bit unsigned integer)

When you start running a server process, you tell the operating system what port number to associate with it. This is called **binding**.

Operating system  
opens a TCP  
connection on port **80**  
of the computer at  
103.238.69.140.



The server process  
running on port **80**  
Is responding to  
requests.



A TCP connection requires an IP address  
**and** a port number.

- If no port number is specified, 80 is the default for HTTP requests.

# Ports defaults

There are many **well-known ports**, i.e. the ports that will be used by default for particular protocols:

21: File Transfer Protocol (FTP)

22: Secure Shell (SSH)

23: Telnet remote login service

25: Simple Mail Transfer Protocol (SMTP)

53: Domain Name System (DNS) service

80: Hypertext Transfer Protocol (HTTP) used in the World Wide Web

110: Post Office Protocol (POP3)

119: Network News Transfer Protocol (NNTP)

123: Network Time Protocol (NTP)

143: Internet Message Access Protocol (IMAP)

161: Simple Network Management Protocol (SNMP)

194: Internet Relay Chat (IRC)

443: HTTP Secure (HTTPS)



# Development server

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

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

For our development server, we can choose whatever port number we want. In this example, we've chosen 3000.

# Running the server

When we run `node server.js` in the terminal, we see the following:

```
vrk:node-server $ node server.js  
Server running!
```

The process does not end after we run the command, as it is now waiting for HTTP requests on port 3000.

**Q: How do we send an HTTP request on port 3000?**

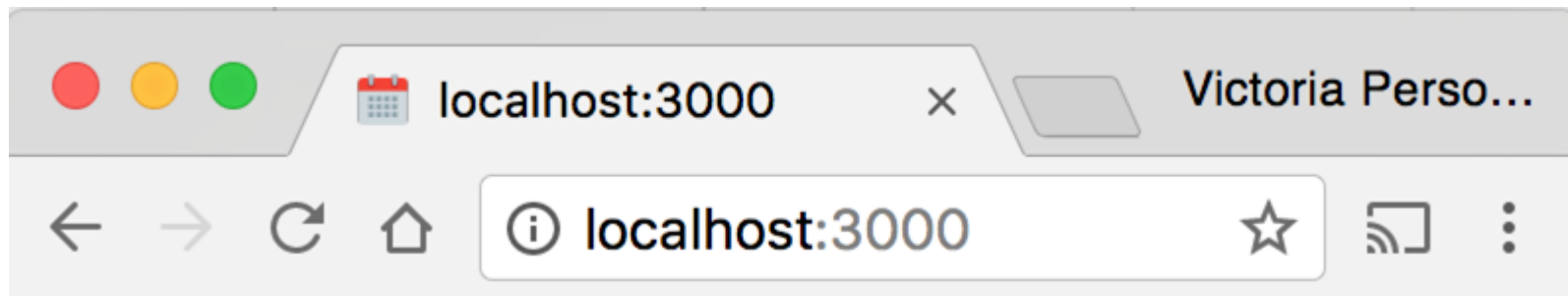
# Localhost

We can send an HTTP GET request running on one of the ports on the local computer using the URL:

**`http://localhost:portNumber`**, e.g.

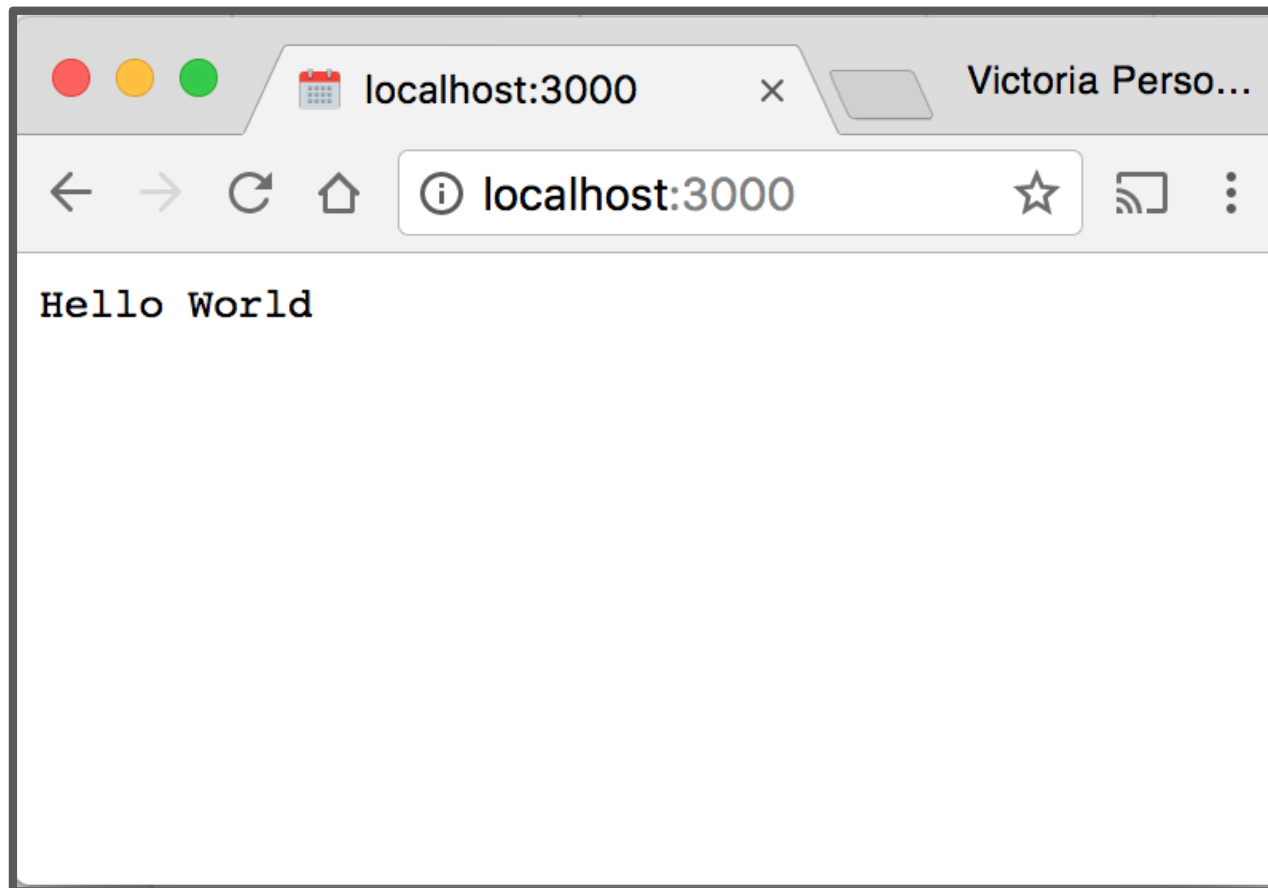
**`http://localhost:3000`**

Localhost is a hostname that means "this computer."



# Server response

Here is the result of the request to our HTTP server:



# Node for servers

This server  
returns the same  
response no  
matter what the  
request is.

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

const server = http.createServer();

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

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

server.listen(3000);
```

# Node for servers

The NodeJS server APIs are actually pretty low-level:

- You build the request manually
- You write the response manually
- There's a lot of tedious processing code

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

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

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

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

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

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

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

<br />

# ExpressJS

We're going to use a library called ExpressJS on top of NodeJS:

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

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

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



Express routing

# ExpressJS

However, Express is not part of the NodeJS APIs.  
If we try to use it like this, we'll get an error:

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

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

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

We need to install Express via npm.

# npm

When you install NodeJS, you also install npm:

- **npm**: Node Package Manager\*:  
Command-line tool that lets you install **packages** (libraries and tools) written in JavaScript and compatible with NodeJS
- Can find packages through the online repository:  
<https://www.npmjs.com/>

\*though the creators of "npm" say it's not an acronym (as a joke -\_-)



# npm install and uninstall

`npm install package-name`

- This downloads the *package-name* library into a `node_modules` folder.
- Now the *package-name* library can be included in your NodeJS JavaScript files.

`npm uninstall package-name`

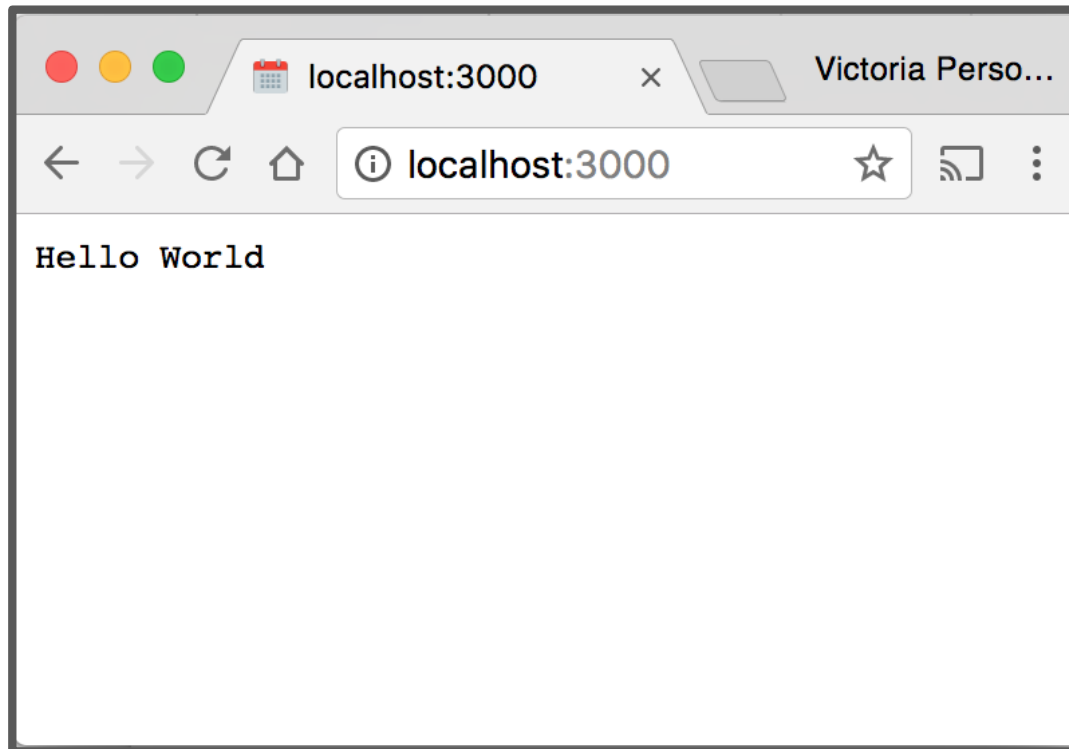
- This removes the *package-name* library from the `node_modules` folder, deleting the folder if necessary

# Express example

```
$ npm install express
```

```
$ node server.js
```

Example app listening on port 3000!



# Express routes

You can specify [routes in Express](#):

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

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

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

# Express routes

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

`app.method(path, handler)`

- Specifies how the server should handle HTTP *method* requests made to URL/*path*
- This example is saying:
  - When there's a GET request to <http://localhost:3000/hello>, respond with the text "GET hello!"

# Handler parameters

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

Express has its own [Request](#) and [Response](#) objects:

- req is a Request object
- res is a Response object
- [res.send\(\)](#) sends an HTTP response with the given content
  - Sends content type "text/html" by default



# Querying our server

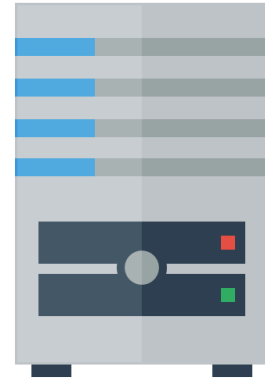
Here are three ways to send HTTP requests to our server:

1. Navigate to `http://localhost:3000/<path>` in our browser
  - a. Can only do GET requests
2. Call `fetch()` in web page
  - a. We've done GET requests so far, but can send any type of HTTP request
3. `curl` command-line tool; PostMan...

# Servers

Sometimes URL is a **path to a file** on the internet:

- The web server software (e.g. Node) **grabs that file** from the server's local file system, and **sends back** its contents to you



[HTTP](#): Hypertext Transfer Protocol, the protocol for sending files and messages through the web

# Server static data


We can instead serve our HTML/CSS/JS **statically** from the same server:

▼  fetch-to-server


▶  node\_modules

▼  public

 fetch-text.html

 fetch-text.js

 style.css

 server.js

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

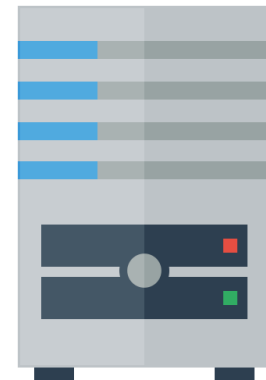
```
app.use(express.static('public'))
```

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

# Web Services

**Other times** URL represents a **parameterized request** (an API endpoint),

- the web server **dynamically generates a response** to that request.



# HTTP methods

**HTTP Methods:** the set of commands understood by a web server and sent from a browser

- **GET:** request/retrieve data

This is request sent by the browser automatically whenever you navigate to a URL!

- **POST:** send/submit data
- **PUT:** upload file
- **PATCH:** updates data
- **DELETE:** delete data
- [More HTTP methods](#)

# RESTful API

RESTful API: URL-based API that has these properties:

- Requests are sent as an **HTTP request**:
  - HTTP Methods: GET, PUT, POST, DELETE, etc
- Requests are sent to **base URL**, also known as an "**API Endpoint**"
- Requests are sent with a specified MIME/content type, such as HTML, CSS, JavaScript, plaintext, JSON, etc.

Sending data to the server

# Sending data to server

3 ways

- 1) Route params
- 2) Body message
- 3) Query params



# Route parameters

When we used the Spotify API, we saw a few ways to send information to the server via our `fetch()` request.

Example: Spotify Album API

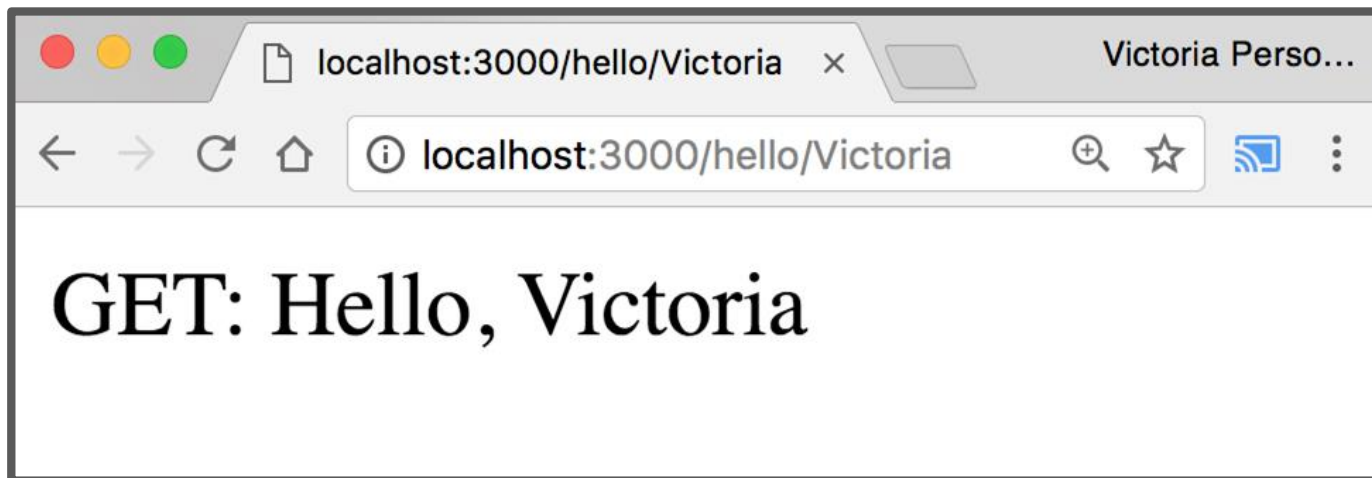
`https://api.spotify.com/v1/albums/7aDBFWp72Pz4NZEtVBANi9`

- The last part of the URL is a **parameter** representing the album id, 7aDBFWp72Pz4NZEtVBANi9

A parameter defined in the URL of the request is often called a "**route parameter**."

# Route parameters

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

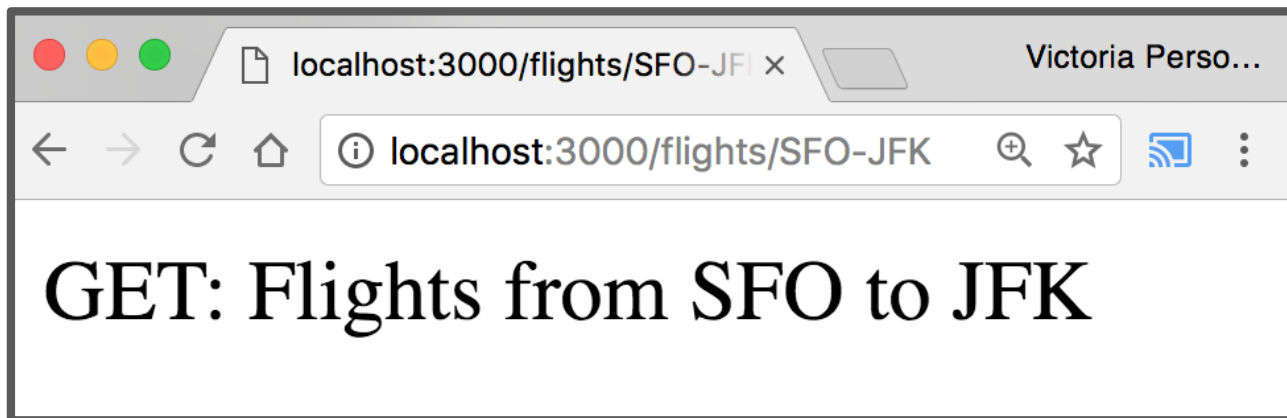


[GitHub](#)

# Route parameters

You can define multiple route parameters in a URL ([docs](#)):

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



Query parameters

# Query parameters

The Spotify Search API was formed using query parameters:

Example: Spotify Search API

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

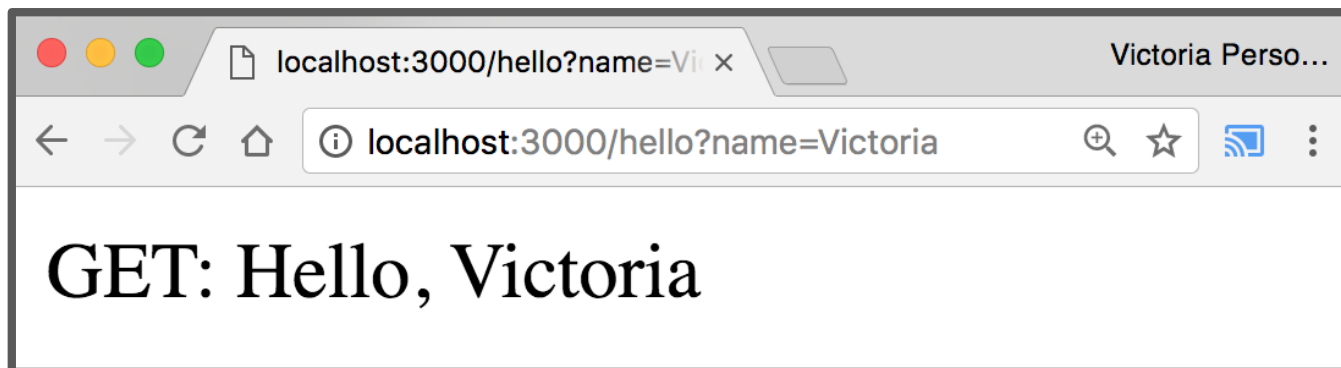
- There were two query parameters sent to the Spotify search endpoint:
  - type, whose value is album
  - q, whose value is beyonce

# Query parameters

**Q: How do we read query parameters in our server?**

**A: We can access query parameters via `req.query`:**

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



# Selected topic \*: Express Routes

\* To understand code in some next slides

# Recall: ExpressJS routes

We've been seeing ExpressJS routes that look like this, with an anonymous function parameter:

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



# ExpressJS routes

Of course, they can also be written like this, with a named function parameter:

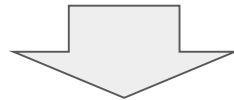
```
function onGet(req, res) {  
  // ...  
}  
app.get('/', onGet);
```

One more random thing:  
Template Literals

# Template literals

[Template literals](#) allow you to embed expressions in JavaScript strings:

```
const port = process.env.PORT || 3000;
app.listen(port, function () {
  console.log('Server listening on port ' + port + '!');
});
```



```
const port = process.env.PORT || 3000;
app.listen(port, function () {
  console.log(`Server listening on port ${port}!`);
});
```

Returning JSON from the server

# JSON response

If we want to return a JSON response, we should use `res.json(object)` instead:

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

The parameter we pass to `res.json()` should be a JavaScript object.

# Example: Dictionary lookup

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

Posting data

# POST message body: `fetch()`

## Client-side:

You should specify a **message body** in your `fetch()` call:

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



# Server-side

**Server-side:** Handling the message body in NodeJS/Express is a little messy ([GitHub](#)):

**// WE WON'T CODE LIKE THIS**

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

# body-parser (extended)

We can use the [body-parser library](#) to help:

```
// process form & json data
app.use(express.urlencoded({extended: true}));
app.use(express.json());
```

This is included in ExpressJS, which we can then pass to routes whose message bodies we want parsed as JSON.

# POST message body

We can access the message body through `req.body`:

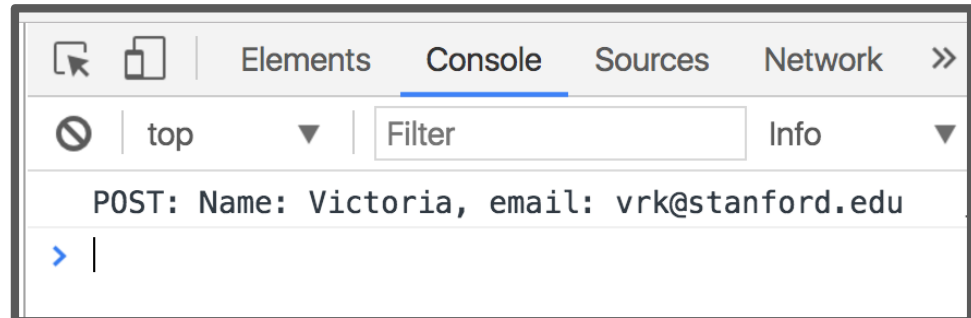
```
app.post('/helloparsed', function (req, res) {  
  const body = req.body;  
  const name = body.name;  
  const email = body.email;  
  res.send('POST: Name: ' + name + ', email: ' + email);  
});
```

[GitHub](#)

# POST message body

Finally, we need to add JSON content-type headers on the `fetch()`-side ([GitHub](#)):

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



# Recap

You can deliver parameterized information to the server in the following ways:

1. Route parameters
2. GET request with query parameters  
(**DISCOURAGED**: POST with query parameters)
3. POST request with message body

**Q: When do you use route parameters vs query parameters vs message body?**

# GET vs POST

- Use [GET](#) requests for retrieving data, not writing data
- Use [POST](#) requests for writing data, not retrieving data

You can also use more specific HTTP methods:

- PATCH: Updates the specified resource
- DELETE: Deletes the specified resource

There's nothing technically preventing you from breaking these rules, but you should use the HTTP methods for their intended purpose.

# Route params vs Query params

Generally follow these rules:

- Use **route parameters** for required parameters for the request
- Use **query parameters** for:
  - Optional parameters
  - Parameters whose values can have spaces

These are conventions and are not technically enforced, nor are they followed by every REST API.

# Example: Spotify API

The Spotify API mostly followed these conventions:

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

- The Album ID is required and it is a route parameter.

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

- q is required but might have spaces, so it is a query parameter
- limit is optional and is a query parameter
- type is required but is a query parameter (breaks convention)

Notice both searches are GET requests, too