CT449:
Phát triển ứng dụng web

Bùi Võ Quốc Bảo
(bvqbao@cit.ctu.edu.vn)

Cần Thơ, 2022

Credit

- The slides are inspired by the CS193X course created by Victoria Kirst

Servers

Server-side programming

“**Client-side**" programming:

- The code we write gets run in a browser on the user's (client's) machine.
- E.g: Javascript, VBScript

“**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
- E.g: Php, Java and JSP

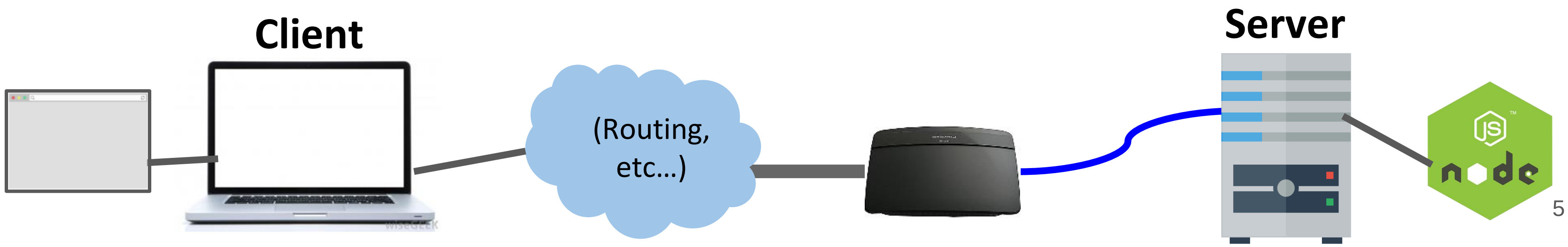
Recall...

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

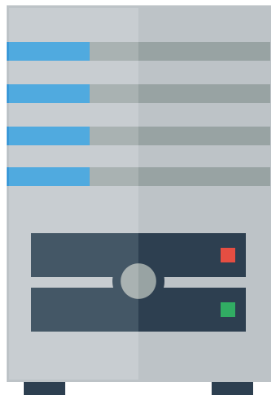


"Server"

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



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

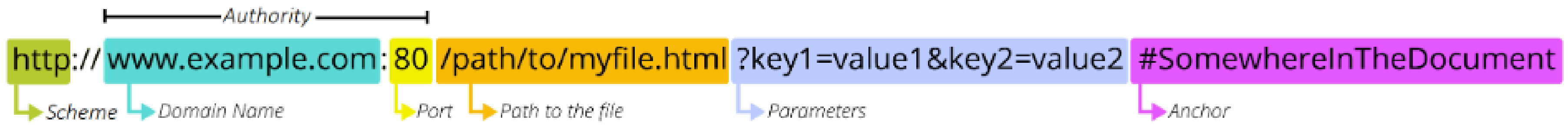
A **socket** is one end of a communication channel. You can send and receive data on sockets

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

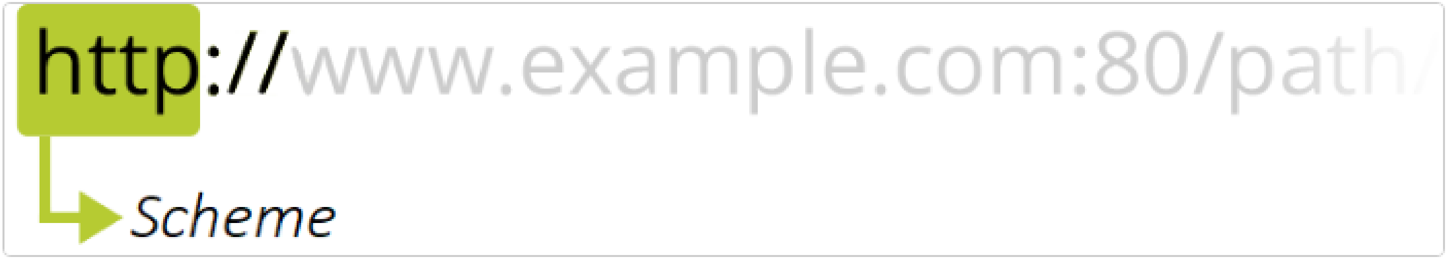
URL Syntax

A URL is composed of different parts, some mandatory and others optional.

scheme://host:port/path?query-string#fragment-id


Authority
http://www.example.com:80/path/to/myfile.html?key1=value1&key2=value2#SomewhereInTheDocument
Scheme
Domain Name
Port
Path to the file
Parameters
Anchor

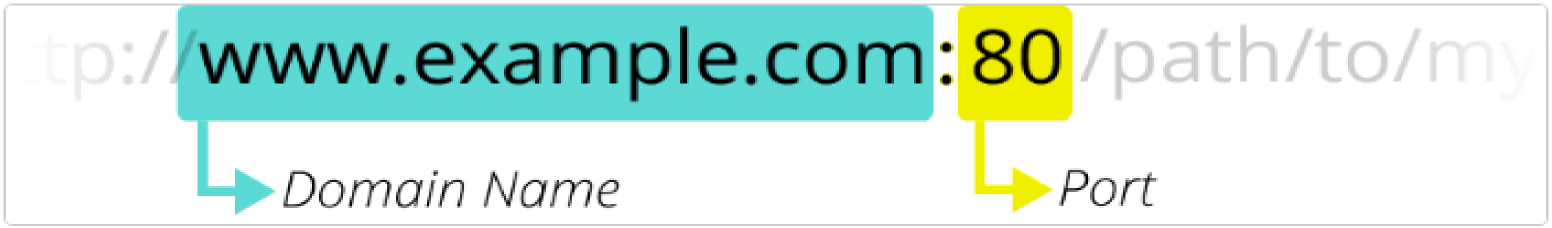
- Schema: indicates the protocol (HTTPS or HTTP, mailto, …) that the browser must use to request the resource



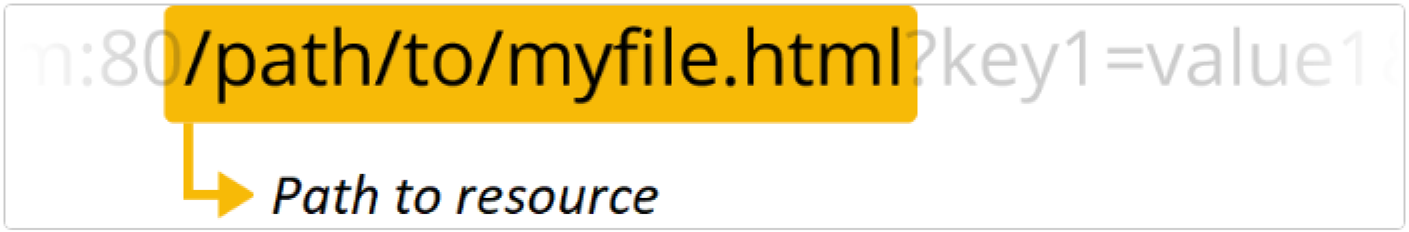
Servers

URL Syntax

- Authority: present the authority includes both the domain (e.g. www.example.com) and the port (80), separated by a colon:

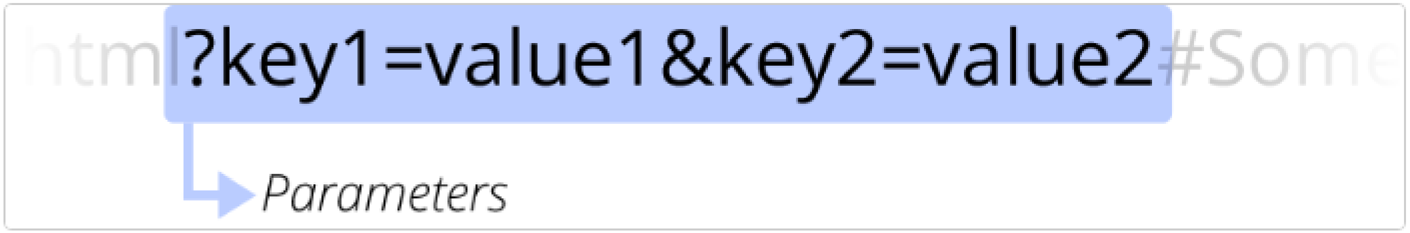


- Path to resource: the path to the resource on the Web server

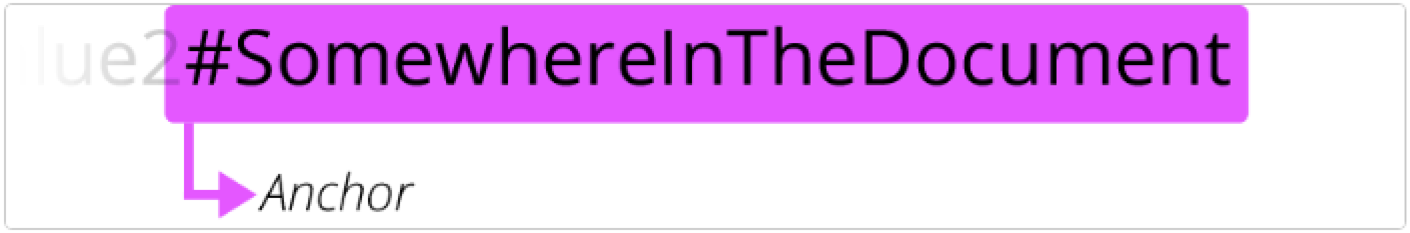


URL Syntax

- Parameters: extra parameters provided to the Web server. Those parameters are a list of key/value pairs separated with the & symbol.



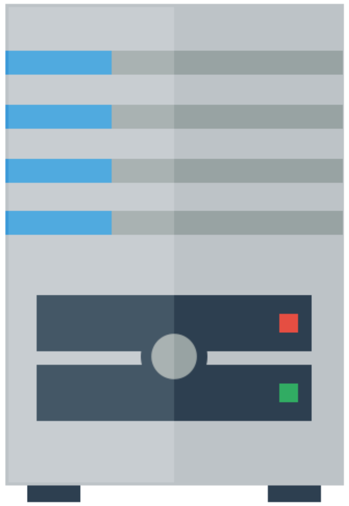
- Anchor: an anchor to another part of the resource itself.
Note: the part after the #, also known as the fragment identifier.

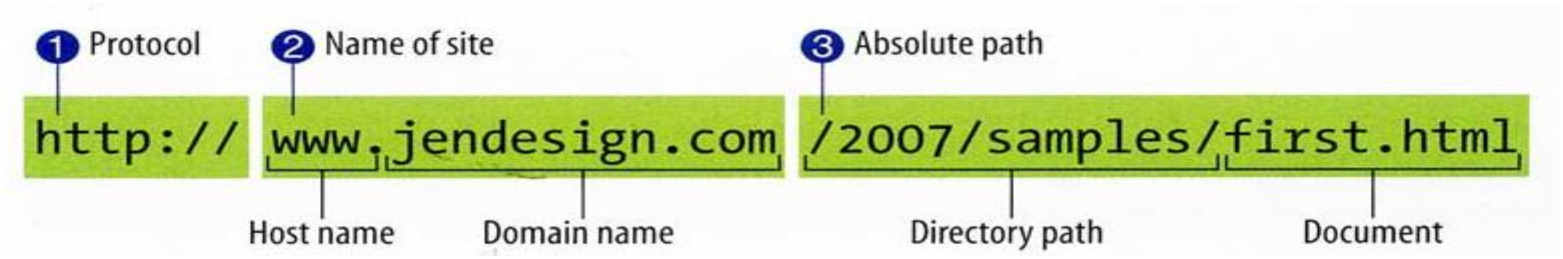


Servers

Sometimes when you type a URL in your browser, the URL is a **path to a file** on the internet:

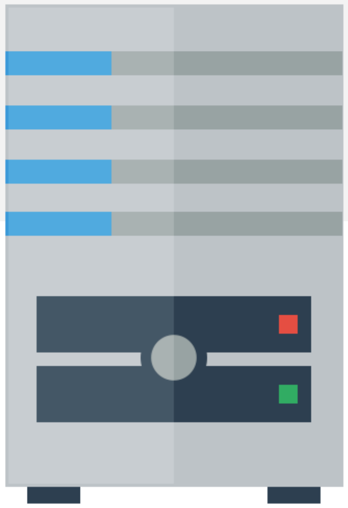
- Your browser connects to the host address and requests the given file over **HTTP**
- The web server software (e.g. Apache) grabs that file from the server's local file system, and sends back its contents to you





But that's not always the case

Web Services



Other times when you type a URL into your browser, the URL represents **an API endpoint**, and not a path to a file. That is:

- The web server does **not** grab a file from the local file system, and the URL is **not** specifying where a file is located.
- Rather, the URL represents **a parameterized request**, and the web server dynamically generates a response to that request.

Web Services

Request Parameters are part of the URL which is used to send additional data to the Server

https://www.bing.com/search?q=ToolsQA

In this URL Request parameter is represented by the "***q=ToolsQA***" part of the URL

Request parameter starts with a question mark (*?*). Request parameters follow ***"Key=Value"*** data format

Web Services

What is an API URL Path?

- An API URL Path is an address that allows you to access an API and its various features.
- There are 2 parts to any API URL
  - ➢ Base URL: is kind of like the base address for the specific API that you’re using
  - ➢ Endpoint: is a specific “point of entry” in an API.

All API endpoints are relative to the base URL. For example, assuming the base URL of `https://api.example.com/v1`, the `/users` endpoint refers to `https://api.example.com/v1/users`.

```
https://api.example.com/v1/users?role=admin&status=active
\________________________/\____/ \______________________/
        server URL        endpoint    query parameters
                              path
```

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

NodeJS

NodeJS:

Javascript runtime refers to *where* your javascript code is executed when you run it.

- Google chrome, in which case your javascript runtime is v8,
- Mozilla - it is spidermonkey,
- IE - then its chakra and

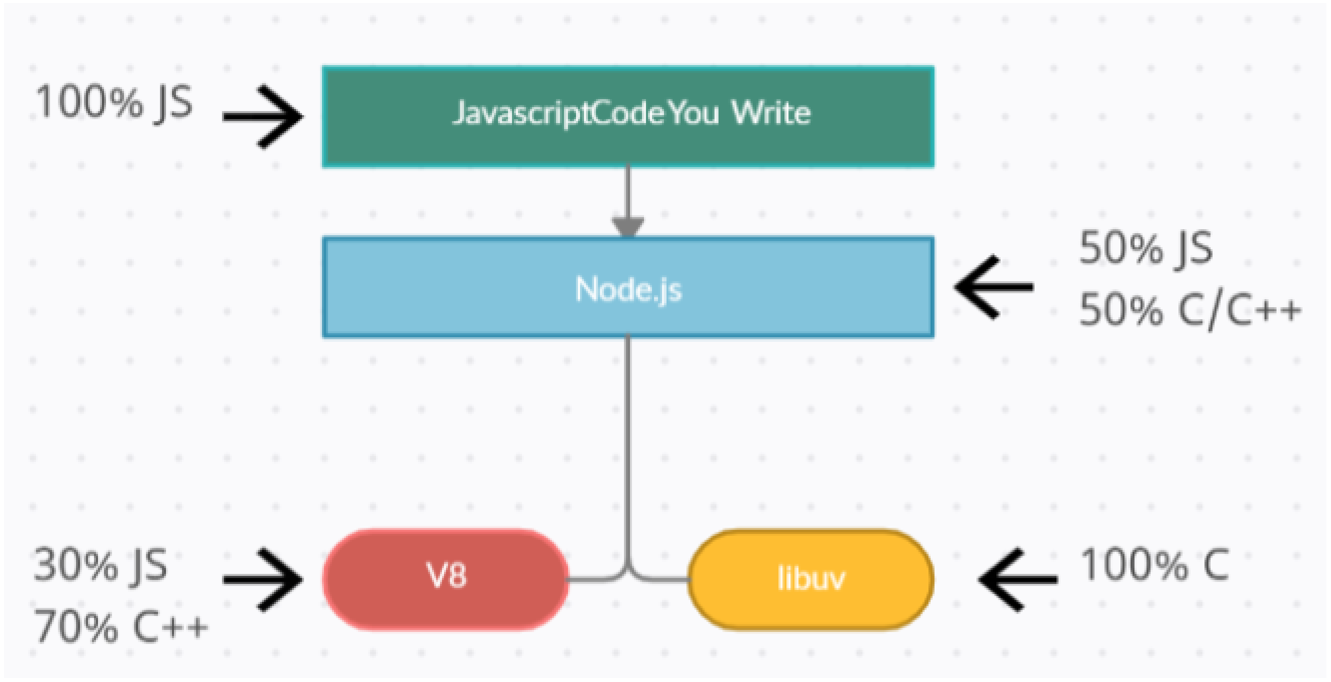
Node.js has a great portion of it written in C/C++ and a lot of its modules are actually implemented in C/C++

- V8: an open-source Javascript Engine created by google to give you the ability to execute javascript code outside of the browser
- libuv project is a C open-source project that gives Node access to the operating systems underlying file system

NodeJS (thảo luận thêm)

NodeJS:

One Important thing to note is that **V8** and **libuv** are not javascript code at all. The V8 project is like **70% C++** code and **libuv is 100% C**.



https://medium.com/swlh/node-js-c-da454904811f

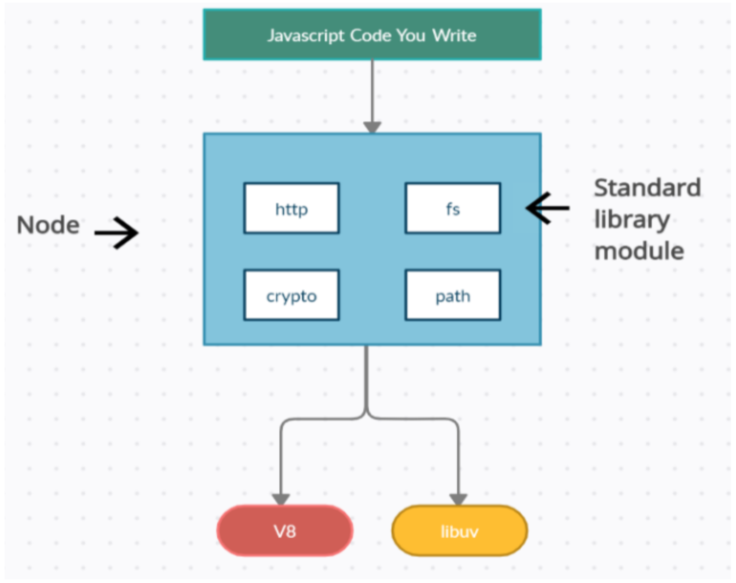
NodeJS

NodeJS:

What Is the Purpose of Node Js?

- Node provides a series of wrappers and a very unified and consistent API for us to use inside of our projects.
  - ➢ For example, Node implements the HTTP, fs, path, and crypto modules… All these modules have very consistent APIs.

→ Node is to give you a consistent API for getting access to functionality that is ultimately implemented inside V8 and libuv.



https://medium.com/swlh/node-js-c-da454904811f

First: Chrome

Chrome:

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

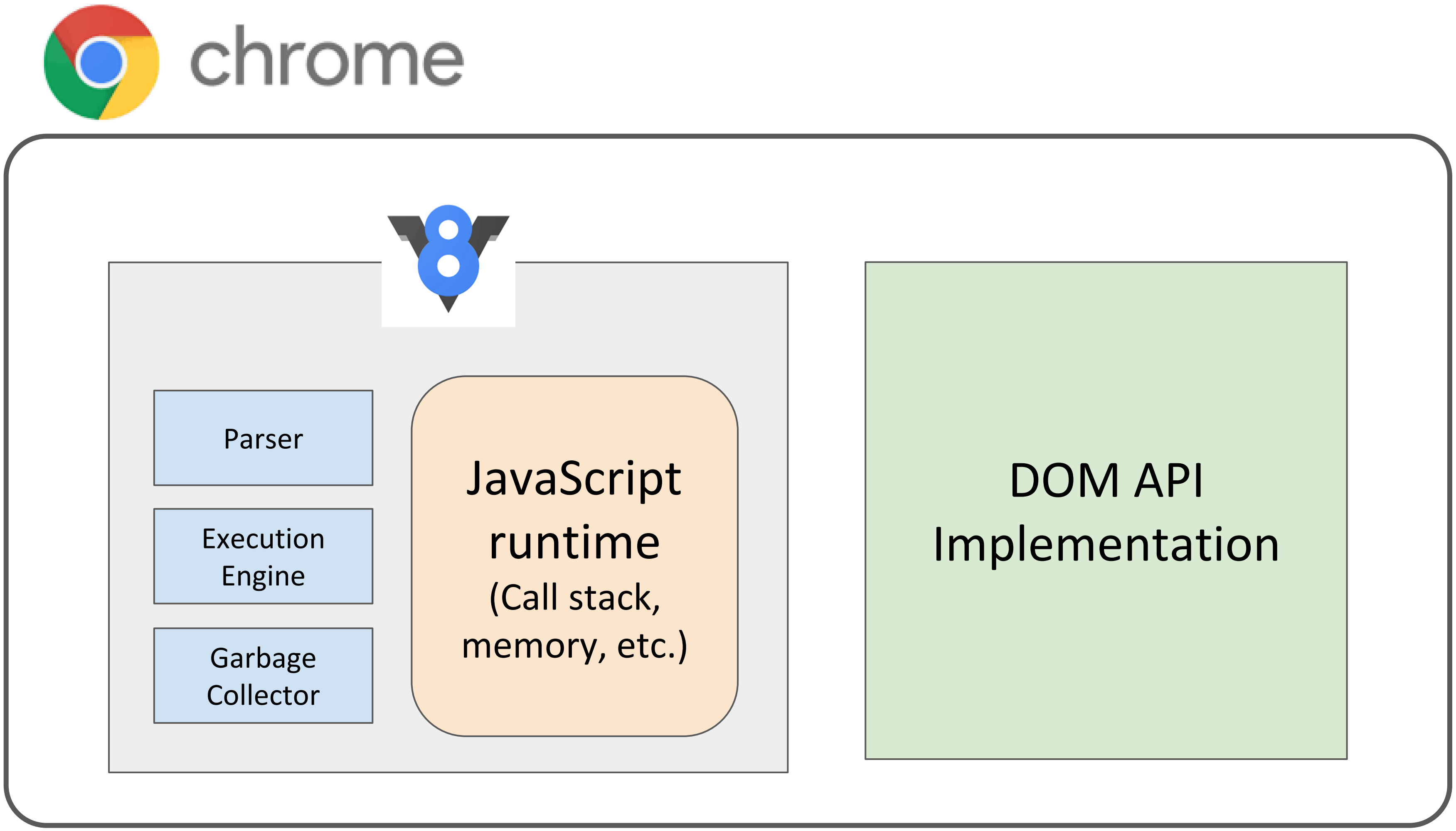
DOM APIs:

- JavaScript libraries to interact with a web page

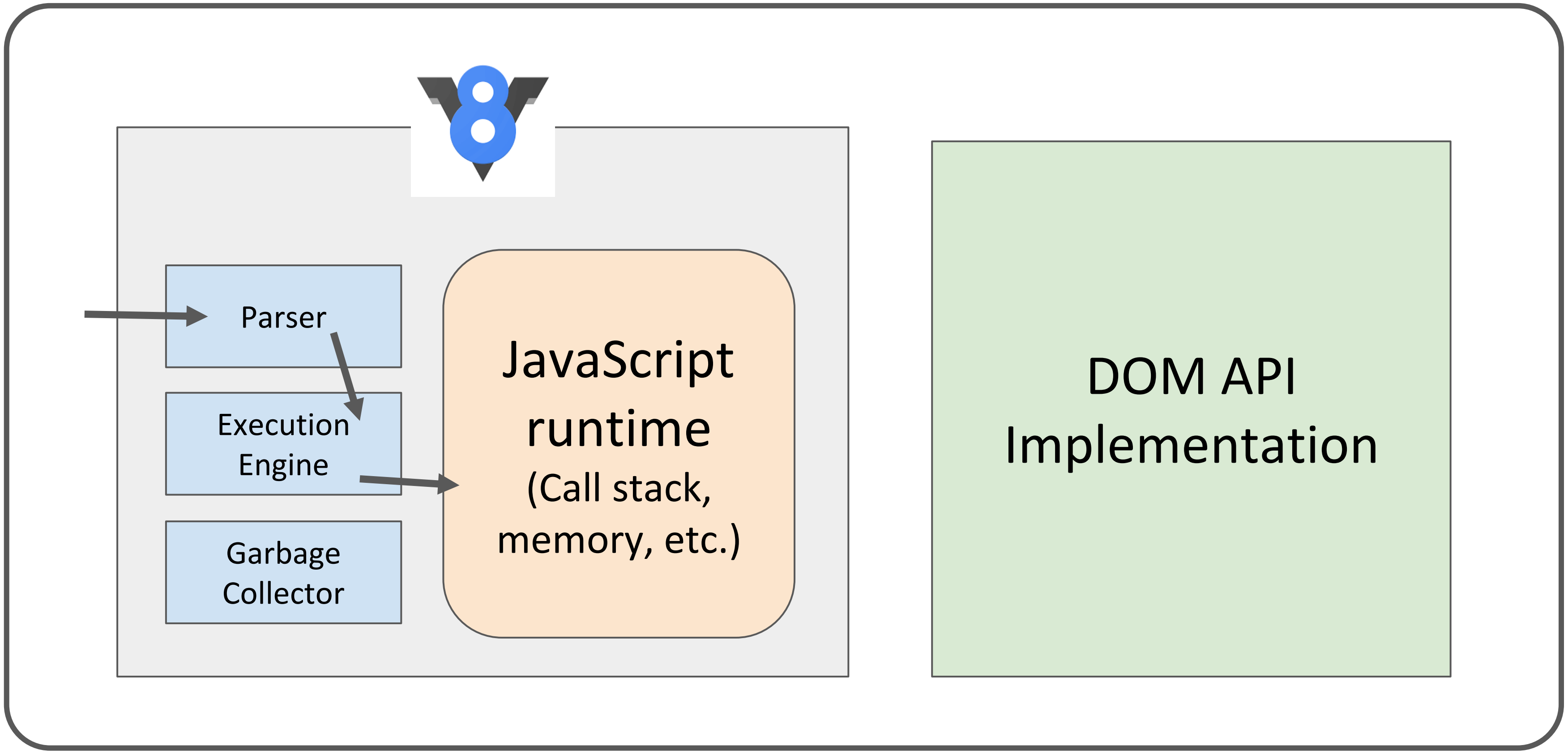
V8:

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

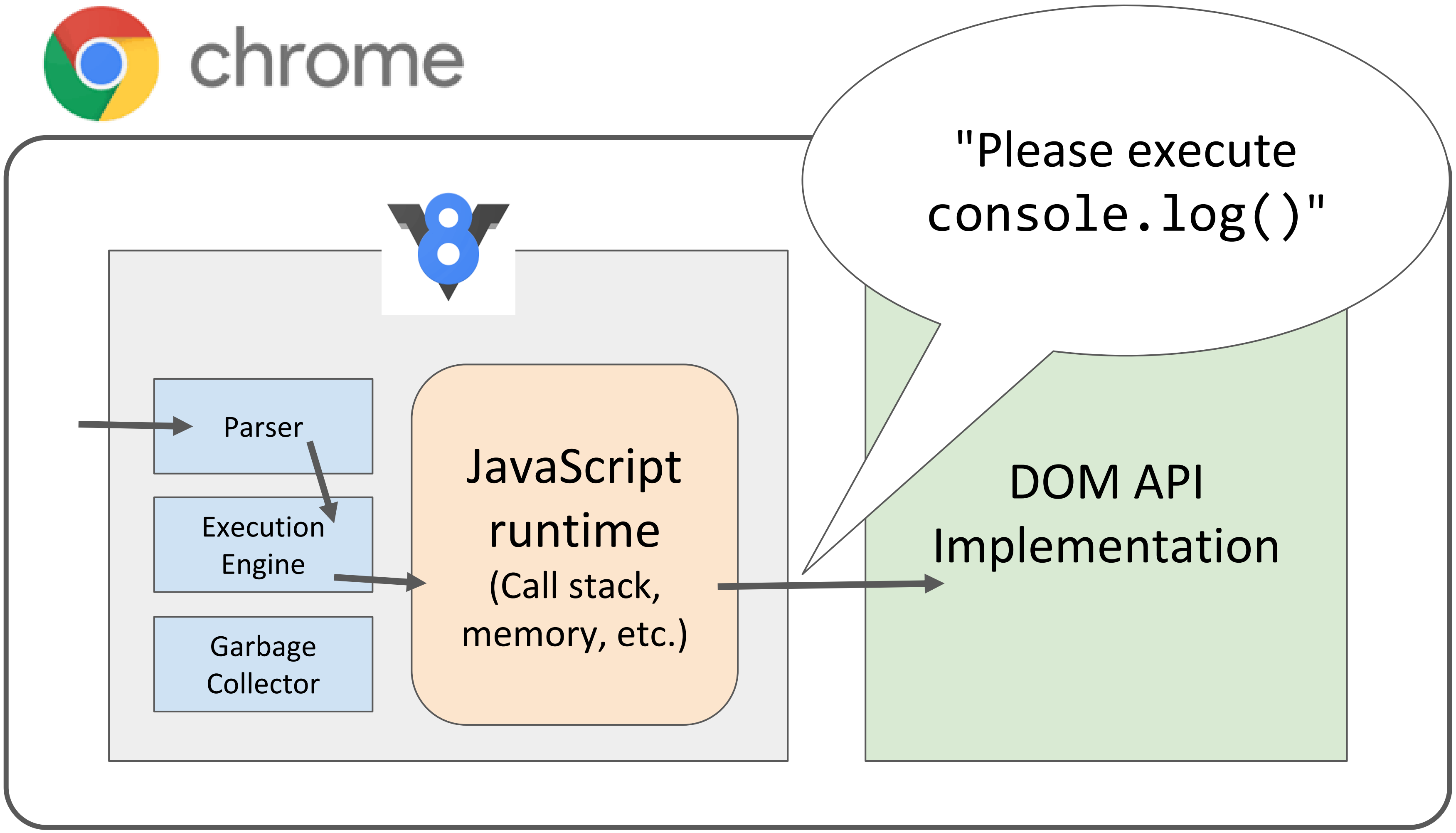
Chrome, V8, DOM



chrome

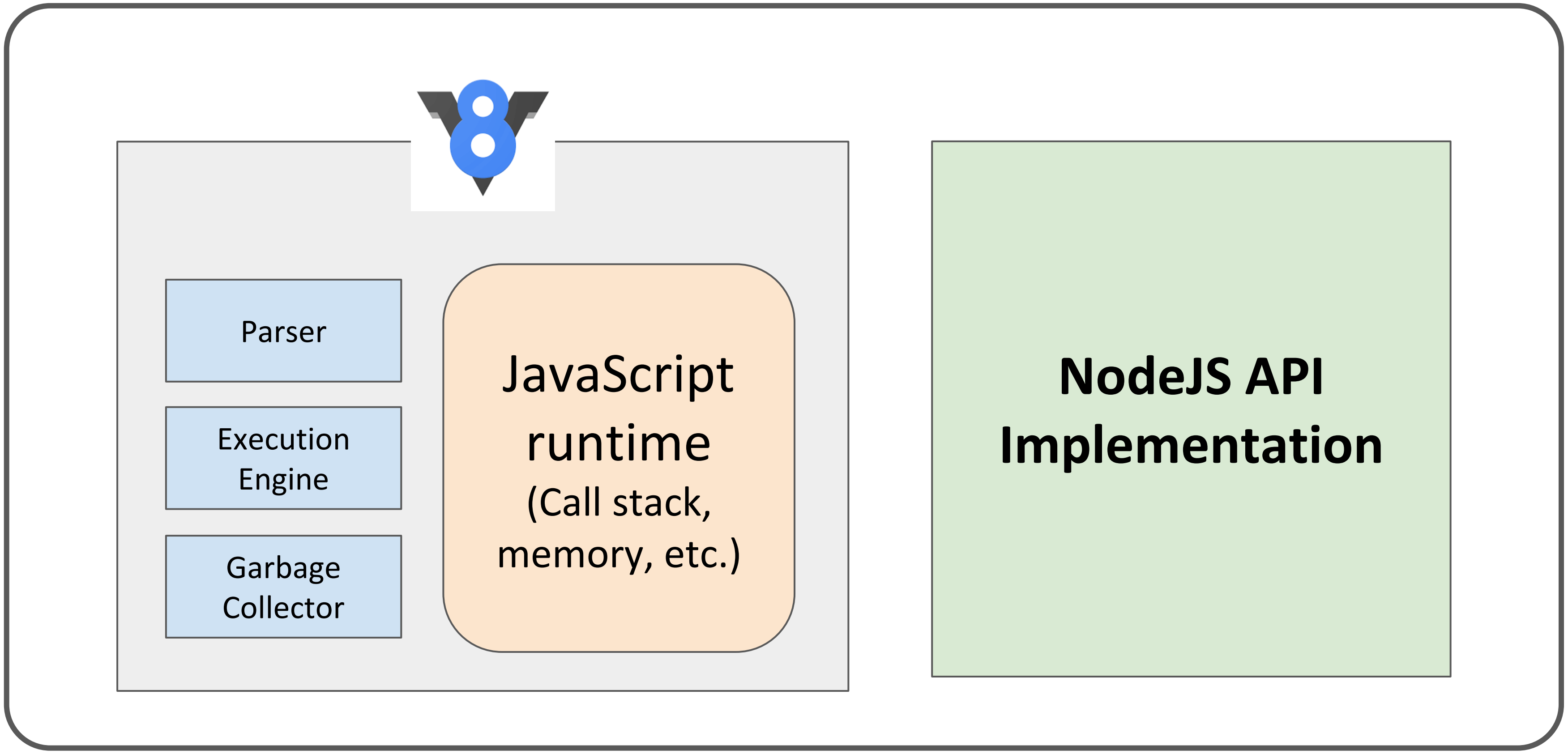


```
const name = 'V8';
```

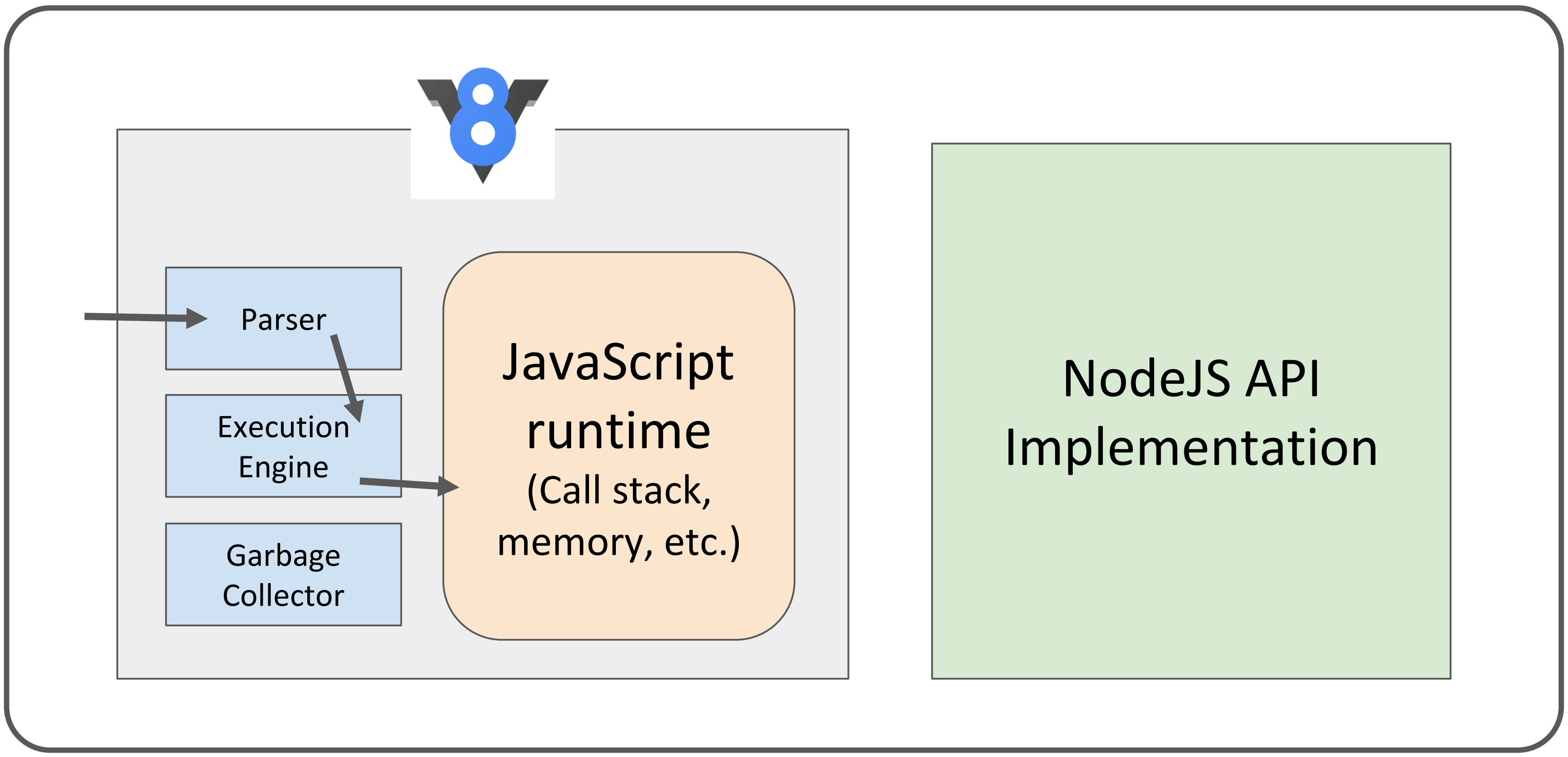


```
console.log('V8');
```

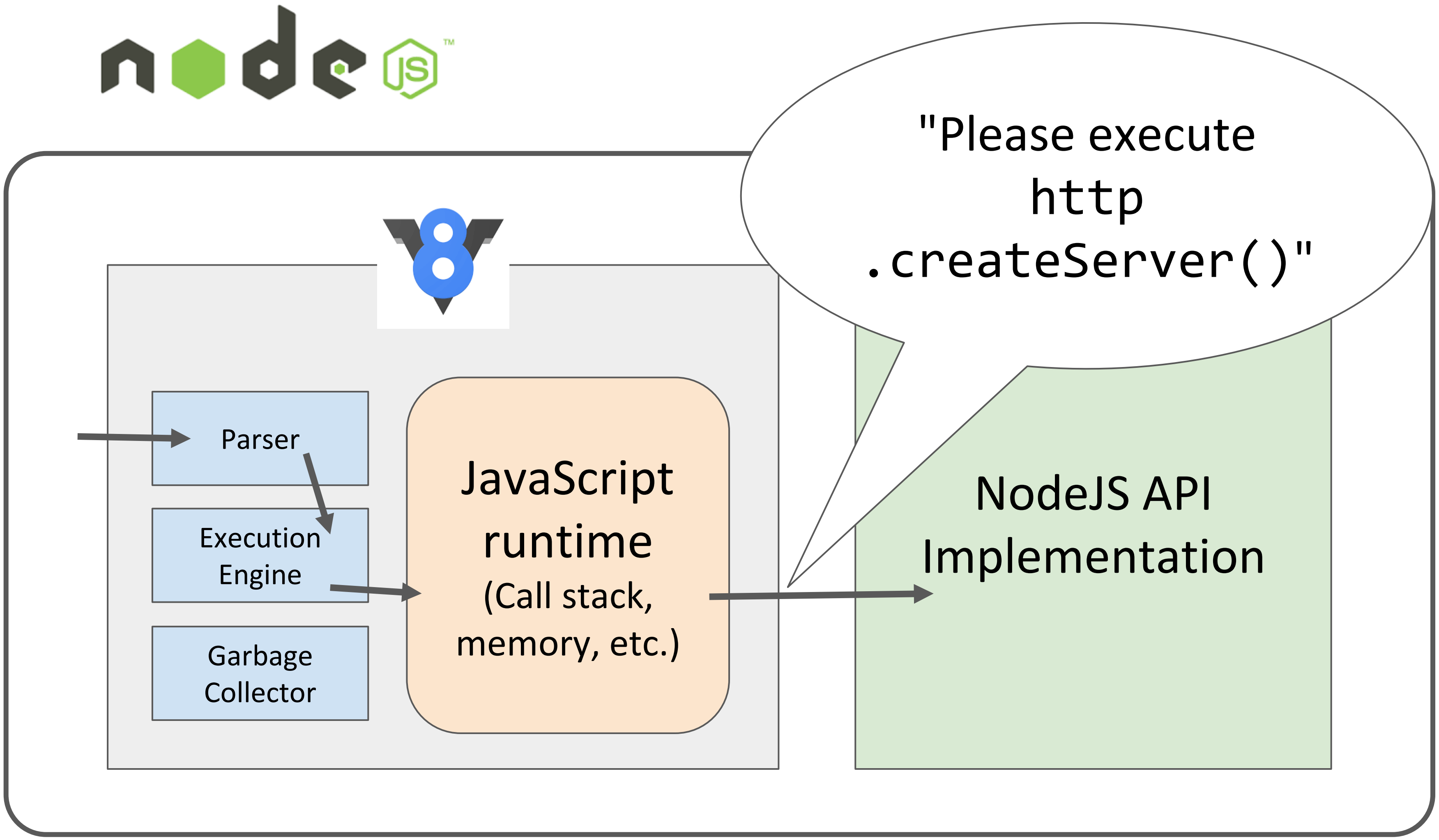
NodeJS, V8, NodeJS APIs



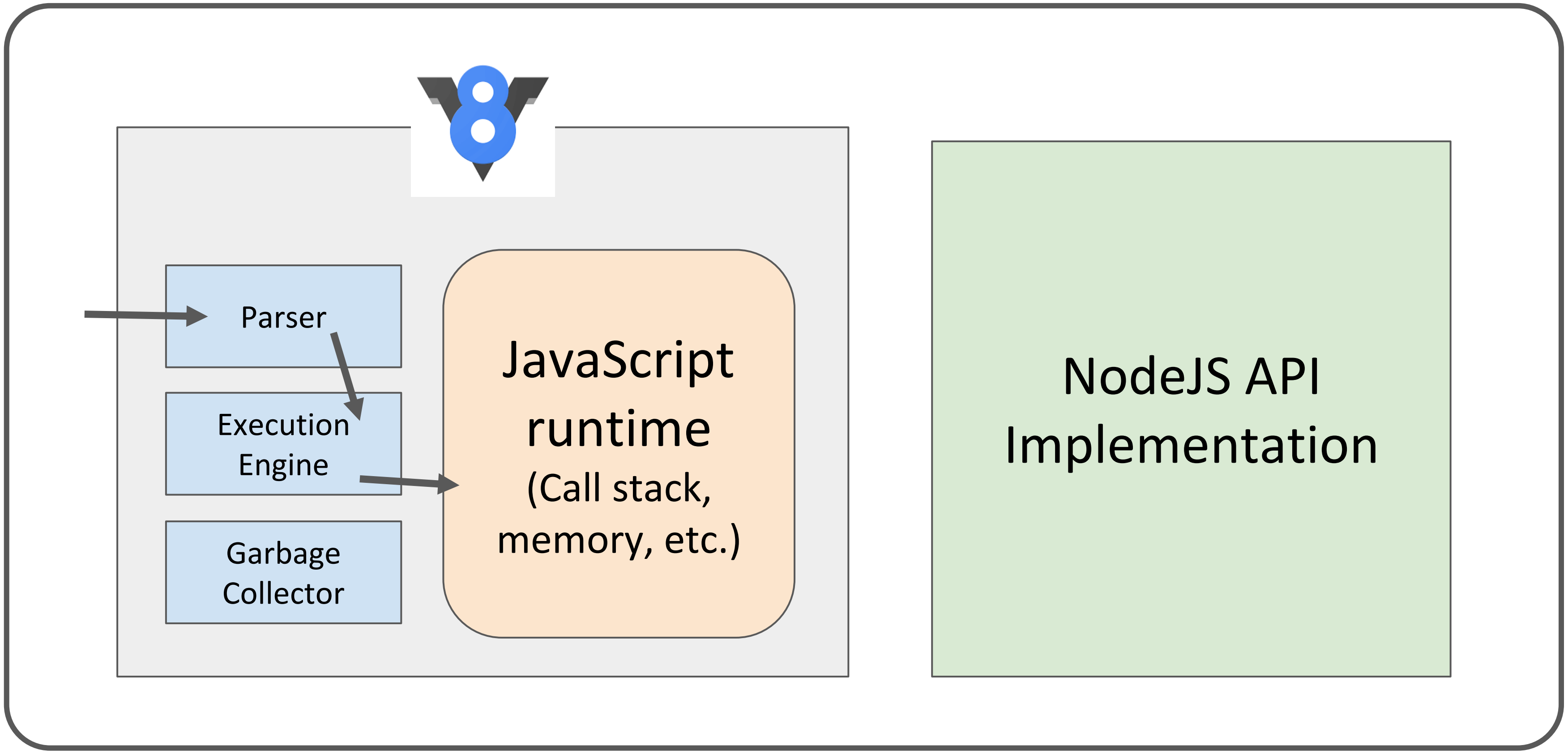
Parser
Execution Engine
Garbage Collector
JavaScript runtime (Call stack, memory, etc.)
NodeJS API Implementation


Parser
Execution Engine
Garbage Collector
JavaScript runtime
(Call stack, memory, etc.)
NodeJS API Implementation

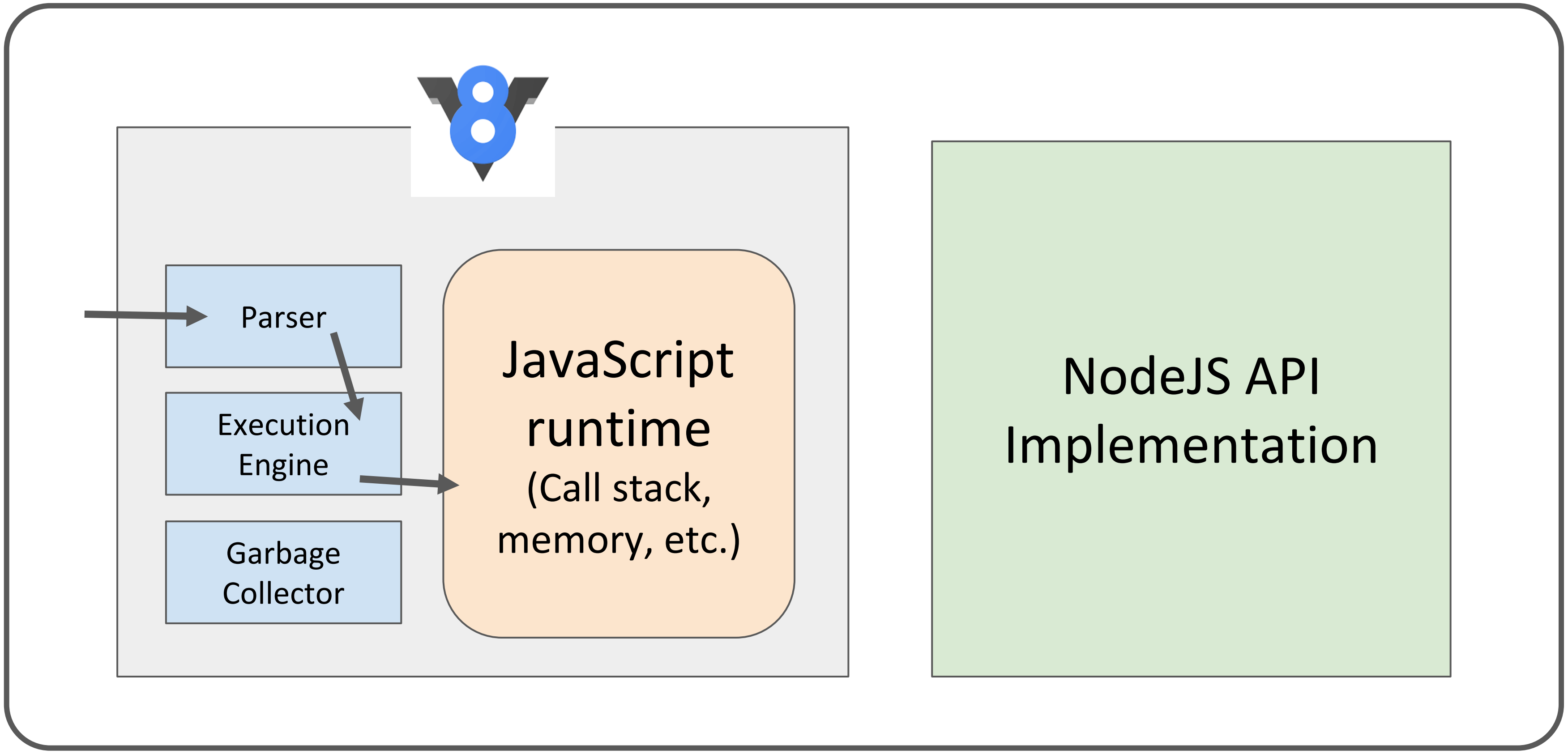
```
const x = 15;
    x++;
```


Parser
Execution Engine
Garbage Collector
JavaScript runtime (Call stack, memory, etc.)
NodeJS API Implementation
"Please execute http .createServer()"

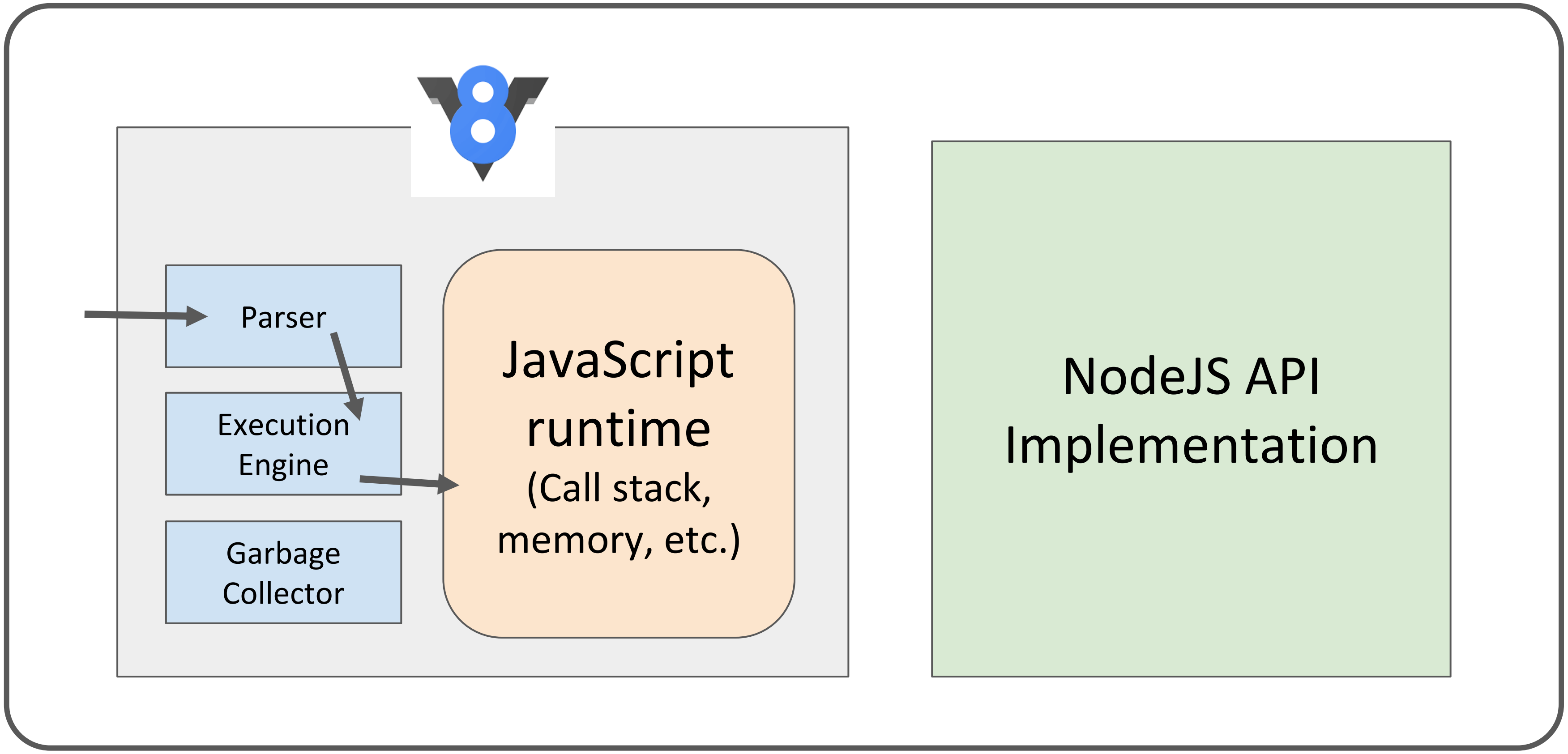
```
http.createServer();
```



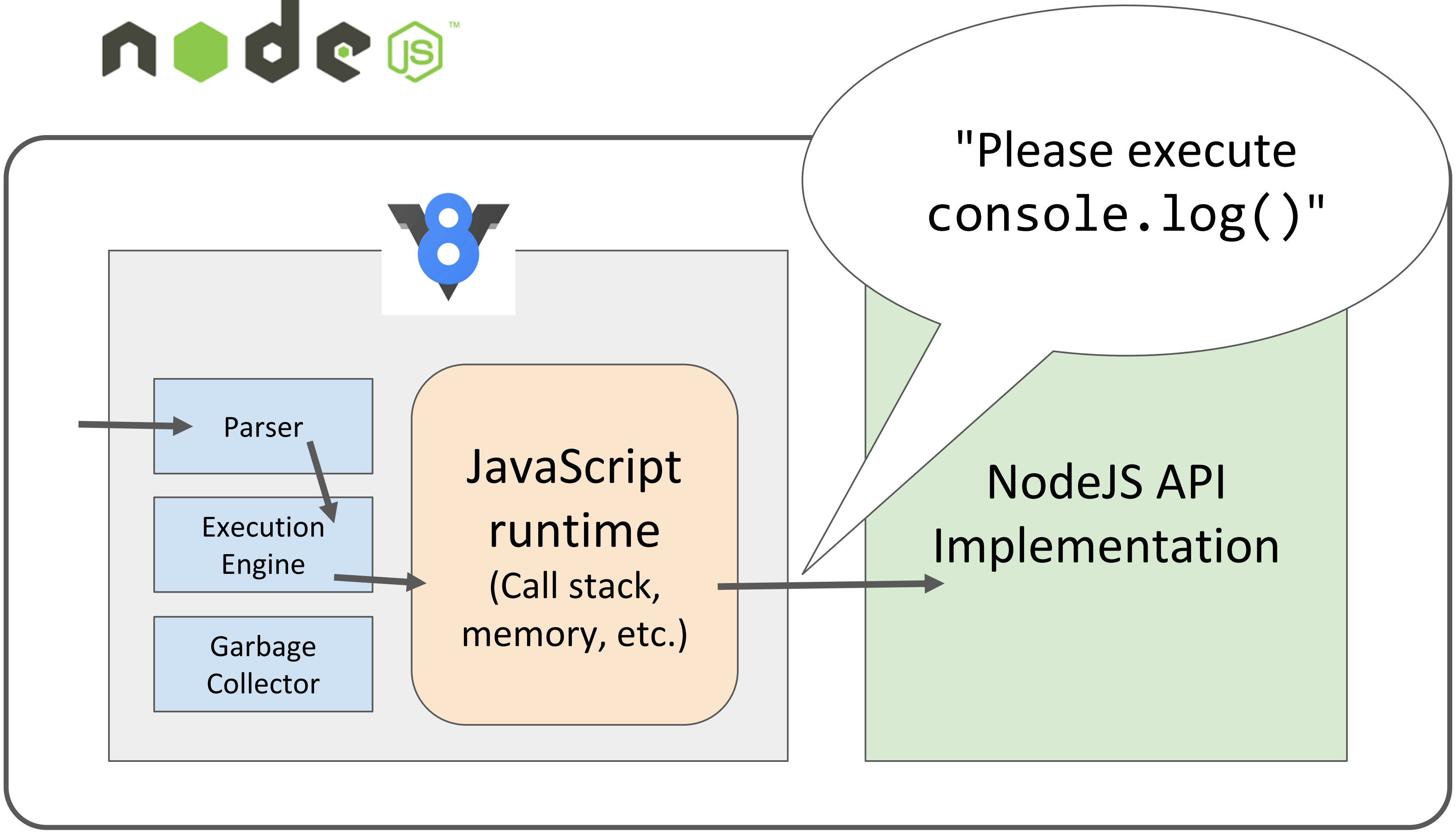
What if you tried to call
`document.querySelector('div');`
in the NodeJS runtime?



```
document.querySelector('div');
ReferenceError: document is not defined
```



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


"Please execute console.log()"
Parser
Execution Engine
Garbage Collector
JavaScript runtime (Call stack, memory, etc.)
NodeJS API Implementation

```
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

NodeJS installation:

- https://nodejs.org/en/download/
- https://github.com/nvm-sh/nvm
- https://github.com/coreybutler/nvm-windows

node command

Running node without a filename runs a read-eval-print loop (REPL)

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

```
$ node
> let x = 5;
undefined
> x++
5
> x
6
```

https://nodejs.org/api/cli.html

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/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

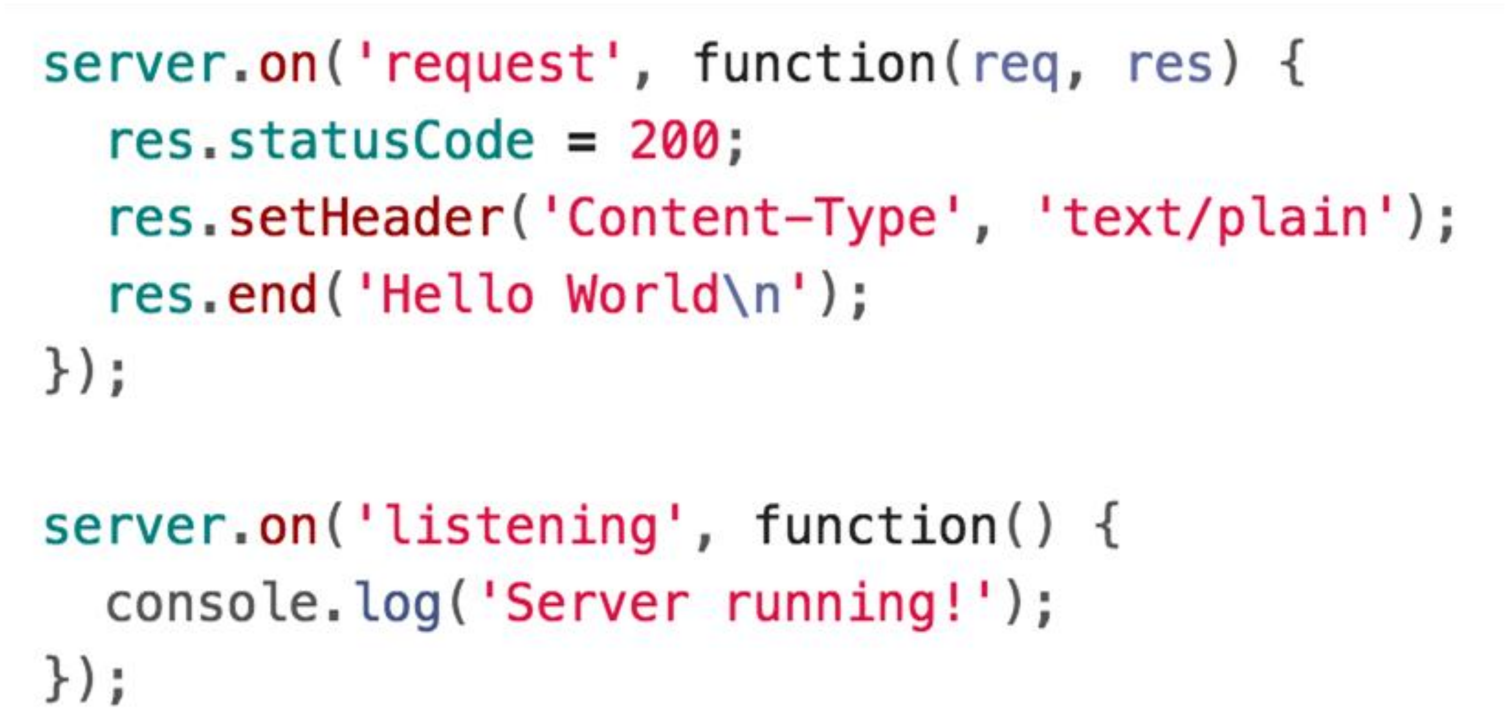
`require()`

```
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

EventEmitter.on



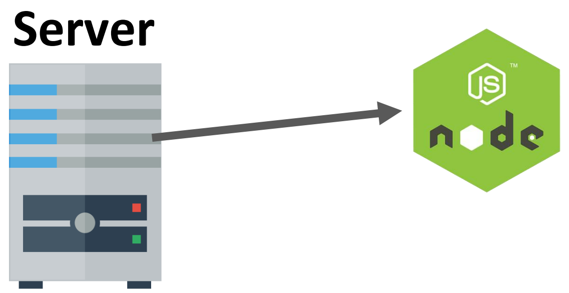
```
server.on('request', function(req, res) {
  res.statusCode = 200;
  res.setHeader('Content-Type', 'text/plain');
  res.end('Hello World\n');
});

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

The on() function is the NodeJS equivalent of `addEventListener`

EventEmitter.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



EventEmitter.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

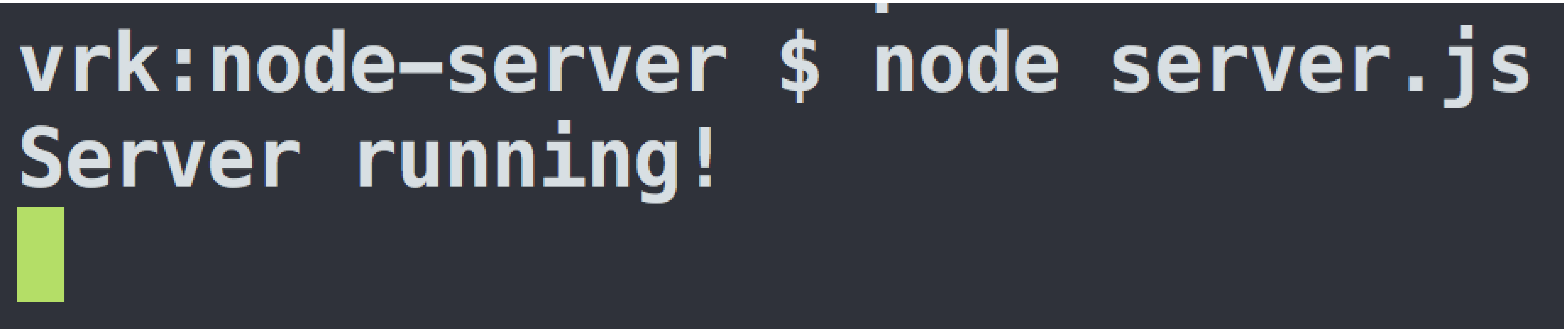
```
server.on('listening', function() {
  console.log('Server running!');
});

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

Running the server

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

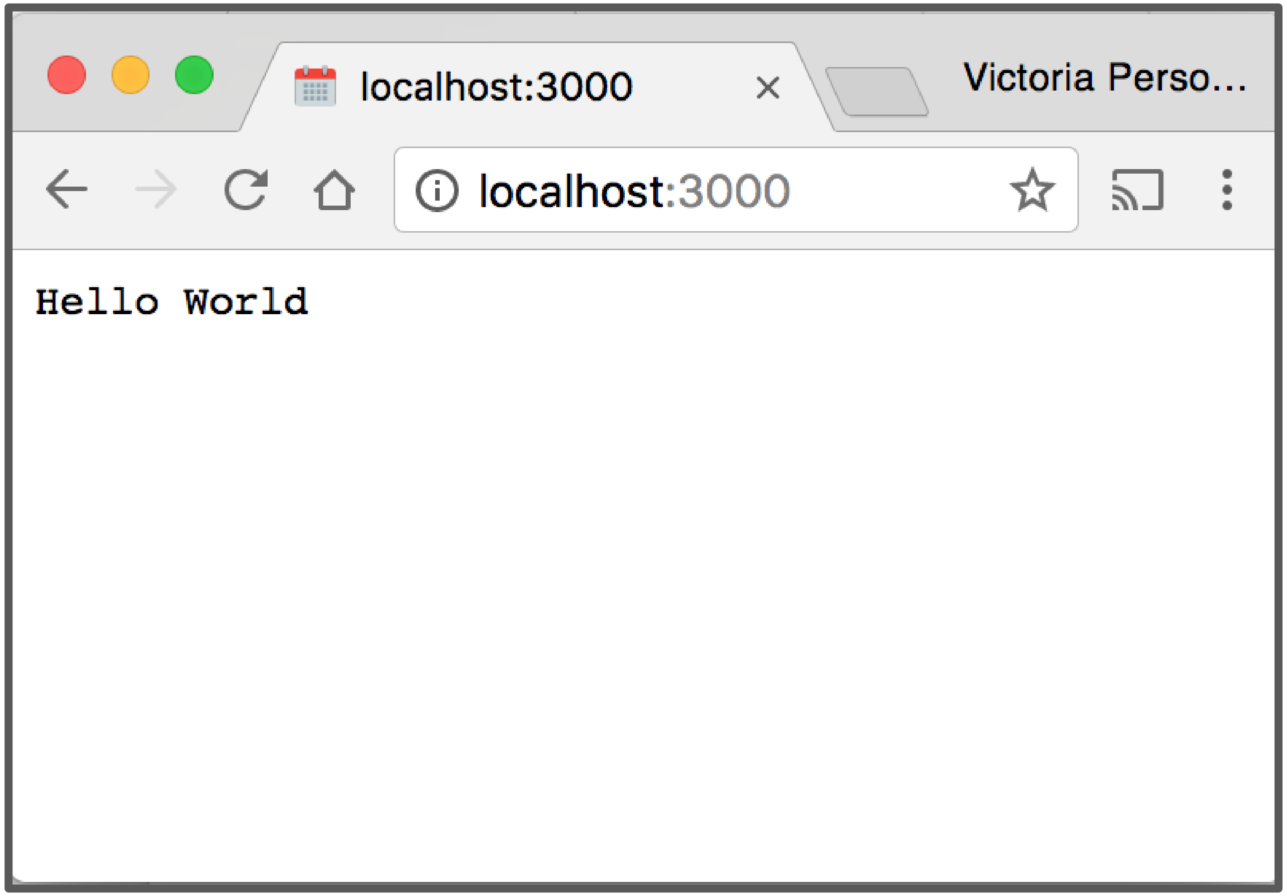


```
vrk:node-server $ node server.js
Server running!
```

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

Server response

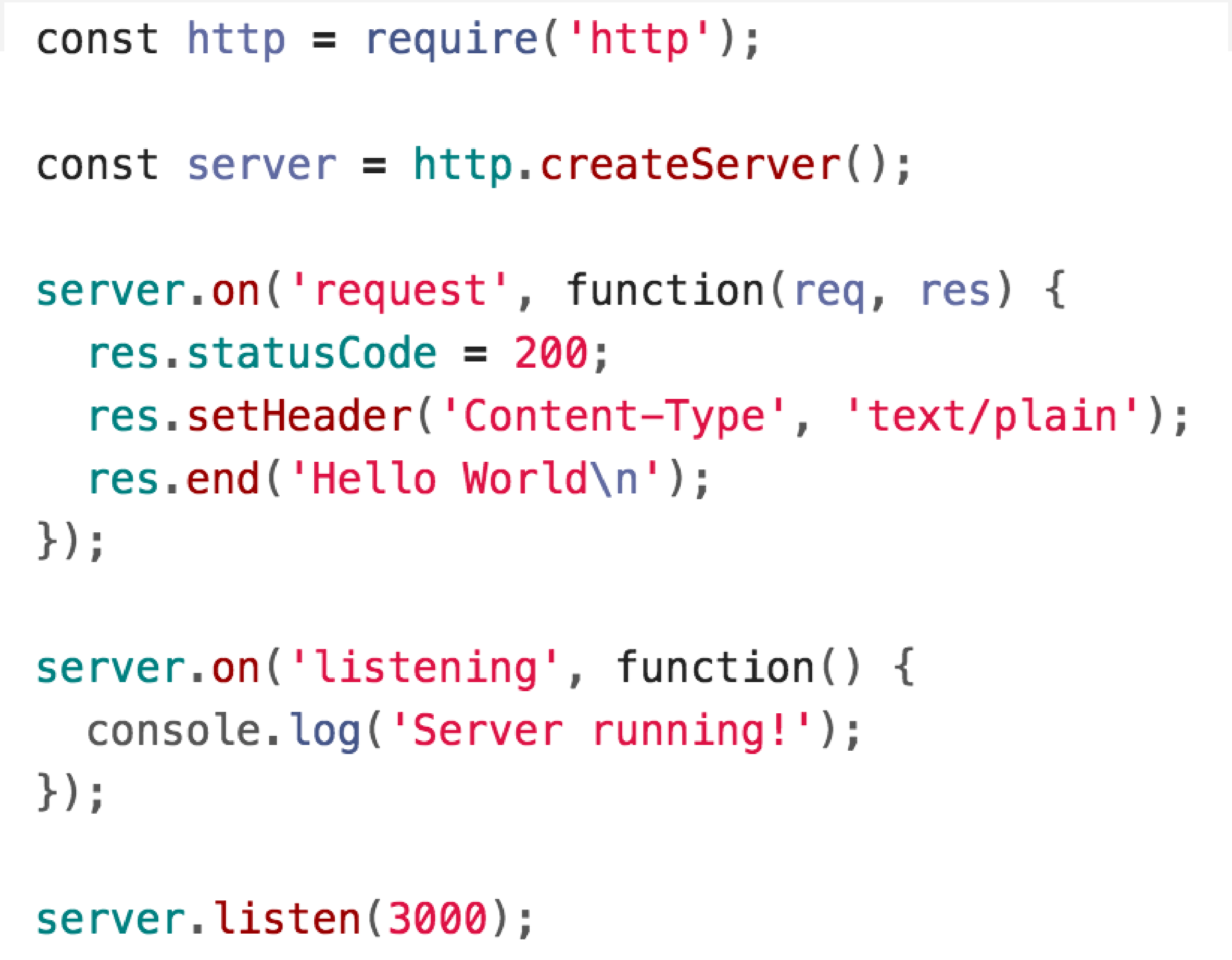
Here is the result of the request to our HTTP server:



```
Hello World
```

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

Node.js provides a bevy of low-level features you'd need to build an application:

- Low-level offerings can be verbose and difficult to use
- 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);
```

Express

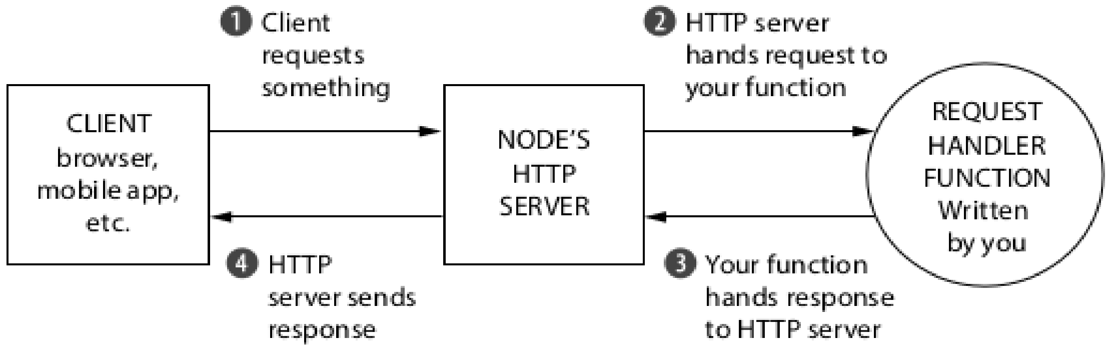
We're going to use a library called Express on top of NodeJS:

- A framework that acts as a light layer atop the Node.js web server

  → making it more pleasant to develop Node.js web applications
- ExpressJS did for NodeJS what Bootstrap did for HTML/CSS and responsive web design

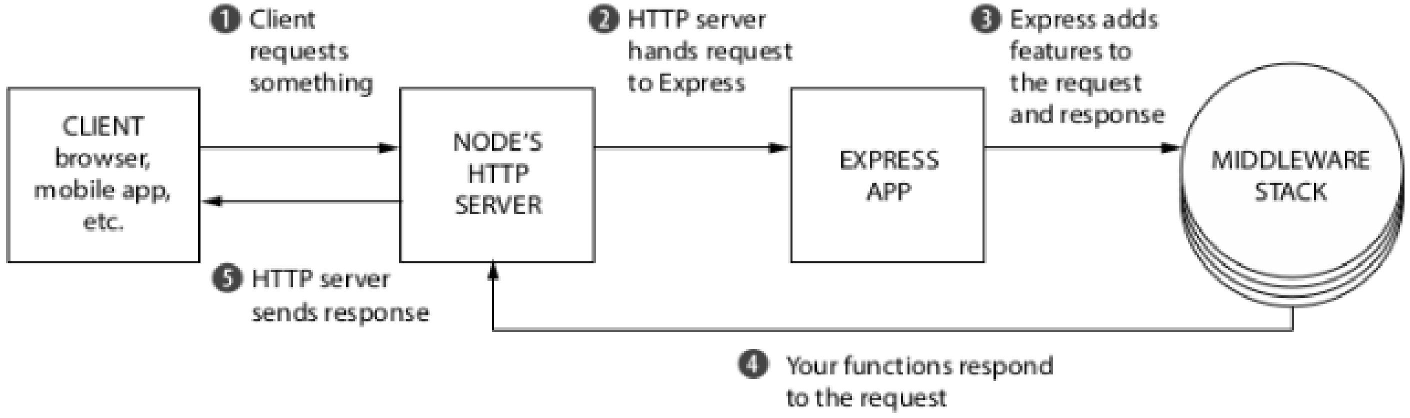
Express

When we are just using Node.js, the flow of a single request might look like this:



1 Client requests something
2 HTTP server hands request to your function
CLIENT browser, mobile app, etc.
NODE'S HTTP SERVER
REQUEST HANDLER FUNCTION Written by you
4 HTTP server sends response
3 Your function hands response to HTTP server

When we add Express, there are a couple of additional steps added to the flow of a request:



1 Client requests something
2 HTTP server hands request to Express
3 Express adds features to the request and response
CLIENT browser, mobile app, etc.
NODE'S HTTP SERVER
EXPRESS APP
MIDDLEWARE STACK
5 HTTP server sends response
4 Your functions respond to the request

Express

We're going to use a library called Express 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

```
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);
```

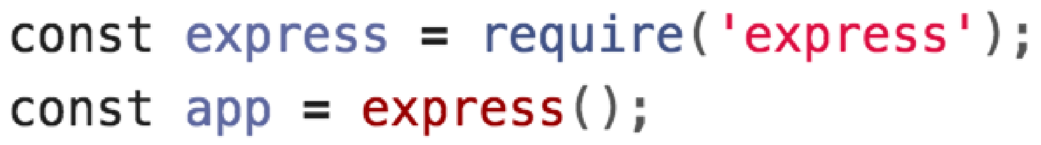
Nodejs

Express routing

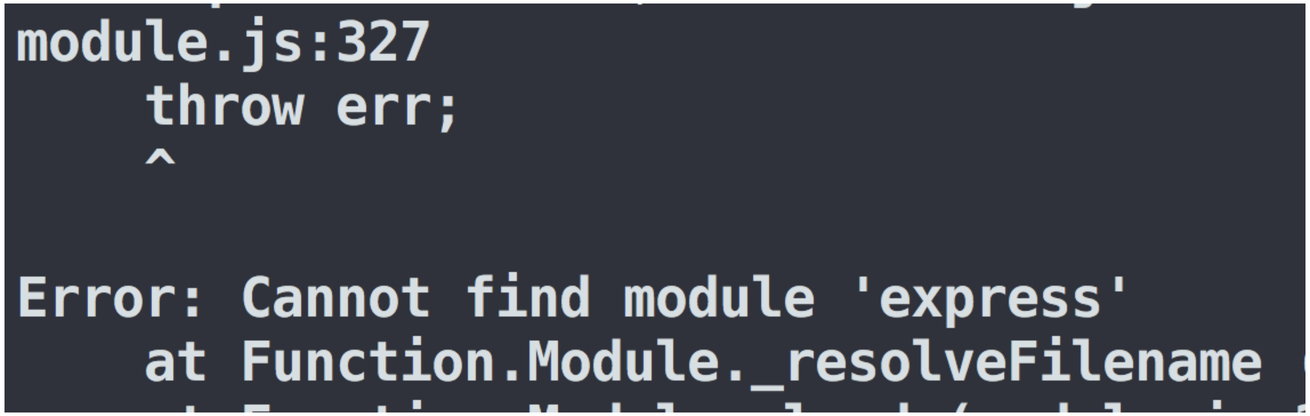
Express

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/

npm install and uninstall

`npm install` ***package-name***

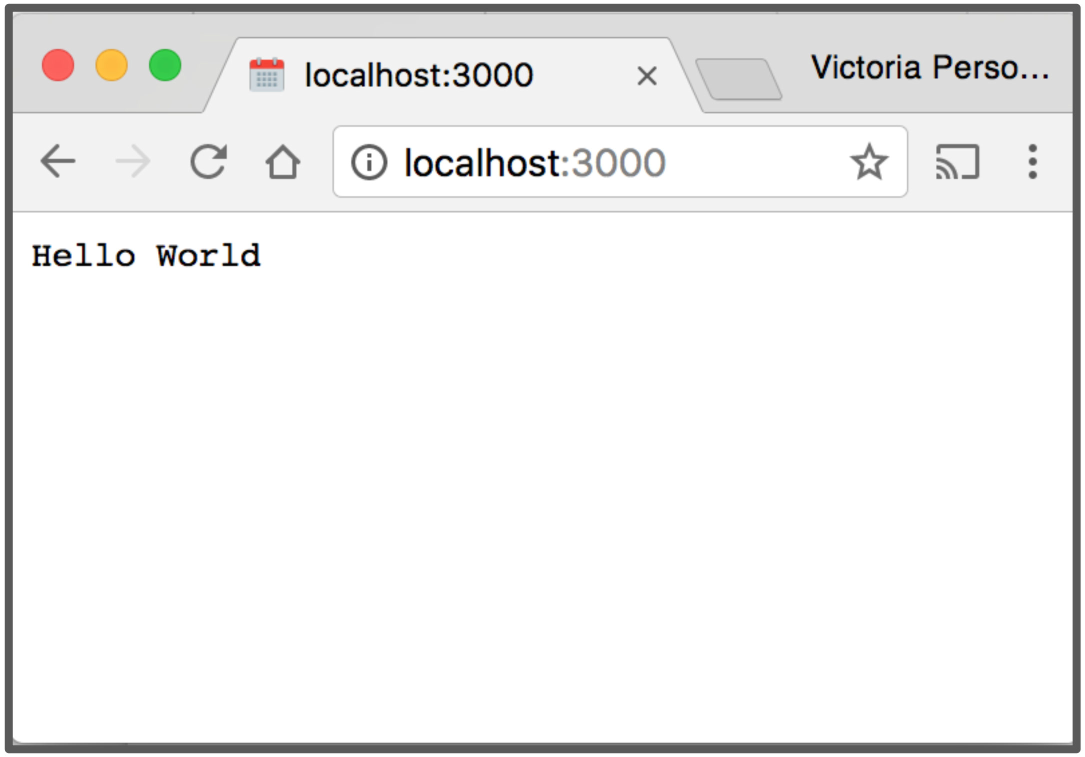
- This downloads the *package-name* library into a **`node_modules`** folder
- Now the *package-name* library can be included in your NodeJS JavaScript files

`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!
```



```
Hello World
```

Express routes

Routes is the process of determining

- what should happen when a URL (any information encoded in request URLs) is called, or
- also which parts of the application should handle a specific incoming request.
- Example, we used this code

```
app.get('/', (req, res) => { /* */ })
```

This creates a route that maps accessing the root domain URL / using the HTTP GET method to the response we want to provide.

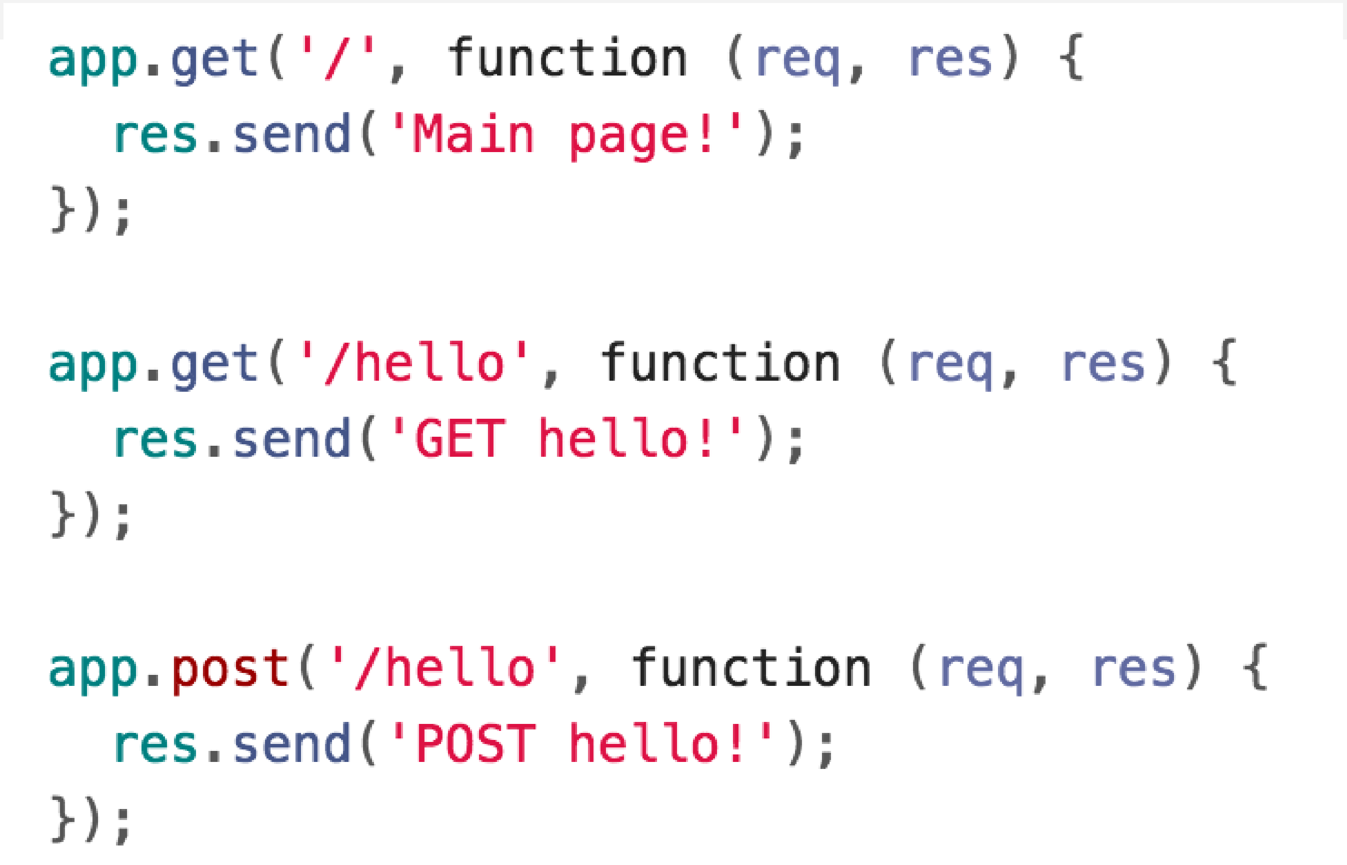
Express routes

app.***method***(***path***, ***handler***)

- Specifies how the server should handle HTTP ***method*** requests made to URL/***path***
- There is a method for every HTTP verb: get() , post() , put() , delete() , patch():

  Example:
  ```
  app.get('/', (req, res) => { /* */ })
  app.post('/', (req, res) => { /* */ })
  app.put('/', (req, res) => { /* */ })
  app.delete('/', (req, res) => { /* */ })
  app.patch('/', (req, res) => { /* */ })
  ```

Express routes

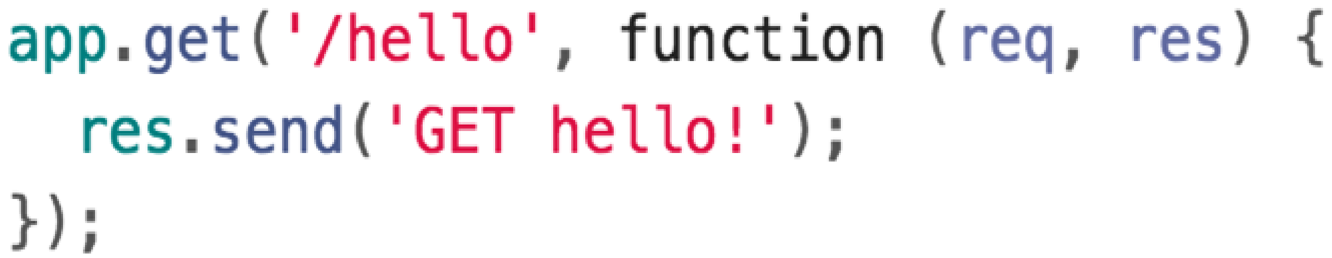


```
app.get('/', function (req, res) {
  res.send('Main page!');
});

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

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

Express routes

app.***method***(***path***, ***handler***)

- This example is saying:
    - When there's a GET request to http://localhost:3000/hello, respond with the text "GET hello!"



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

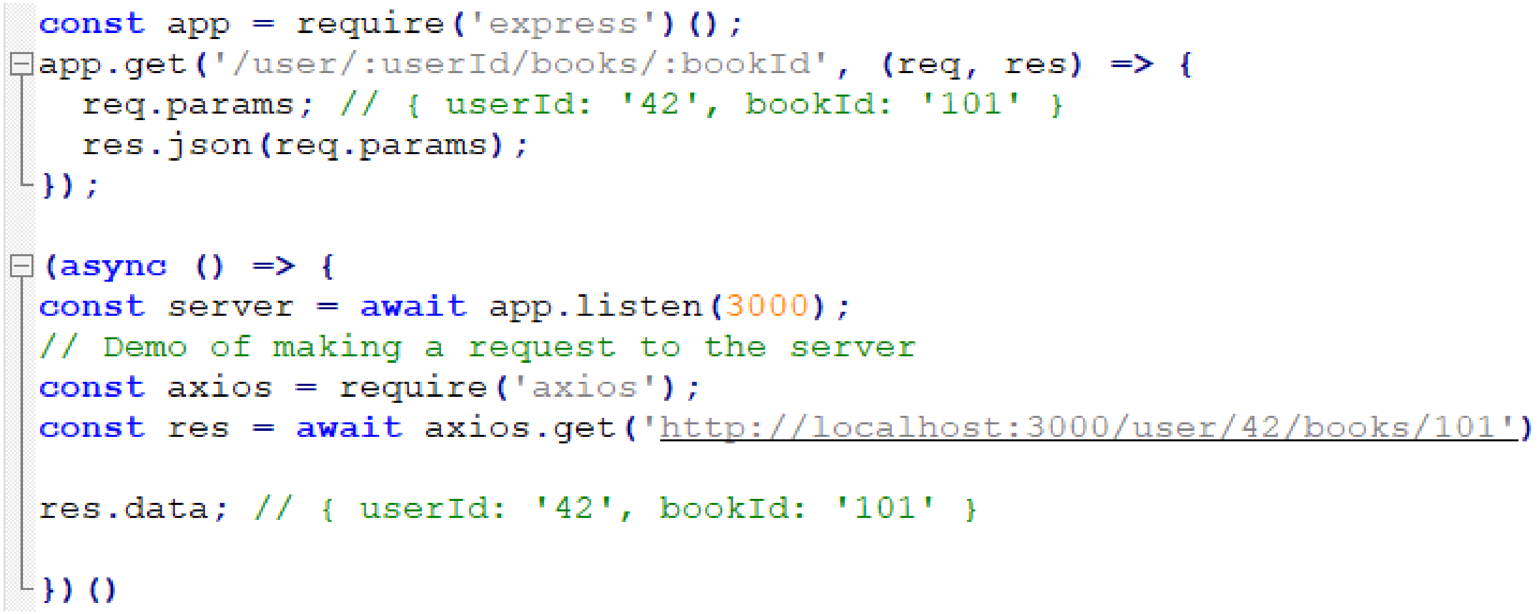
Express routes

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
  - If you pass in a string, it sets the Content-Type header to text/html by default
  - if you pass in an object or an array, it sets the application/json Content-Type header, and parses that parameter into JSON.
    - ➢ send() automatically sets the Content-Length HTTP response header.
    - ➢ send() also automatically closes the connection

Handler parameters

- In Express,
  - Route parameters are essentially variables derived from named sections of the URL.
  - Express captures the value in the named section and stores it in the req.params property



```
const app = require('express')();
app.get('/user/:userId/books/:bookId', (req, res) => {
  req.params; // { userId: '42', bookId: '101' }
  res.json(req.params);
});

(async () => {
const server = await app.listen(3000);
// Demo of making a request to the server
const axios = require('axios');
const res = await axios.get('http://localhost:3000/user/42/books/101')

res.data; // { userId: '42', bookId: '101' }

})()
```

Install axios to run above example*: npm install axios*

Using regular expressions to match routes

- In Express,
  - Regular expressions can used to match multiple paths with one statement

```
app.get(/post/, (req, res) => { /* */ })
```

will match `/post` , `/post/first` , `/thepost` , `/posting/something` , and so on.

https://www.npmjs.com/package/path-to-regexp

Callback

- Callback is called when task get completed and is asynchronous equivalent for a function
- All the APIs of Node are written in such a way that they support callbacks
- Method below accept a callback function, which is called when a request is started, and we need to handle it

```
app.get('/', (req, res) => res.send('Hello World!'))
```

callback function

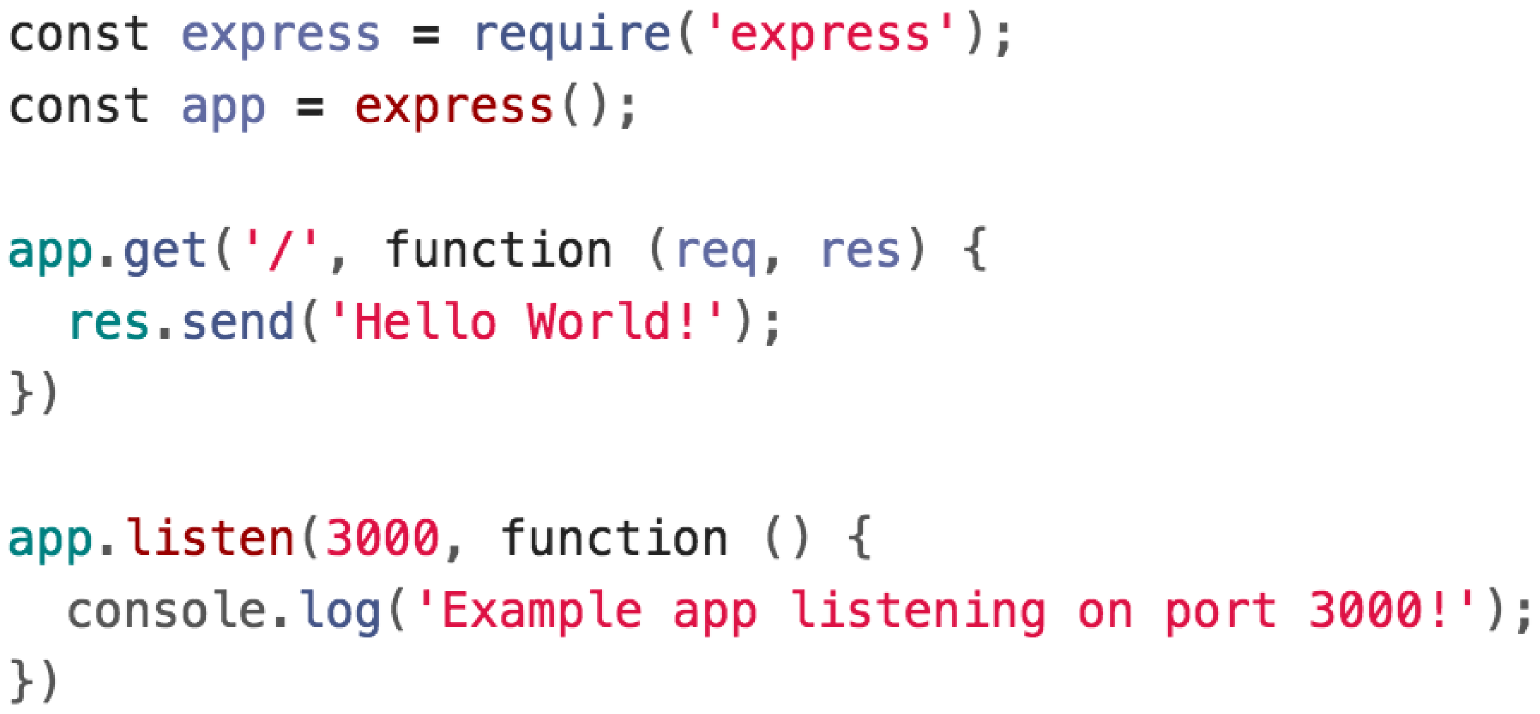
-- > Express sends us two objects in this callback, which we called `req` and `res` , that represent the Request and the Response objects.

Note: In Express, callback function is middleware takes three arguments (usually named as shown: req, res, next)

Querying our server

HTTP requests

Our server is written to respond to HTTP requests (GitHub):



```
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!');
})
```

Q: How do we sent HTTP requests to our server?

Querying our server

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

1. Navigate to http://localhost:3000/<path> in our browser
   - **Caveat:** Can only do GET requests (actually, can do POST requests with a form)

2. `Postman` Web/HTTP API client

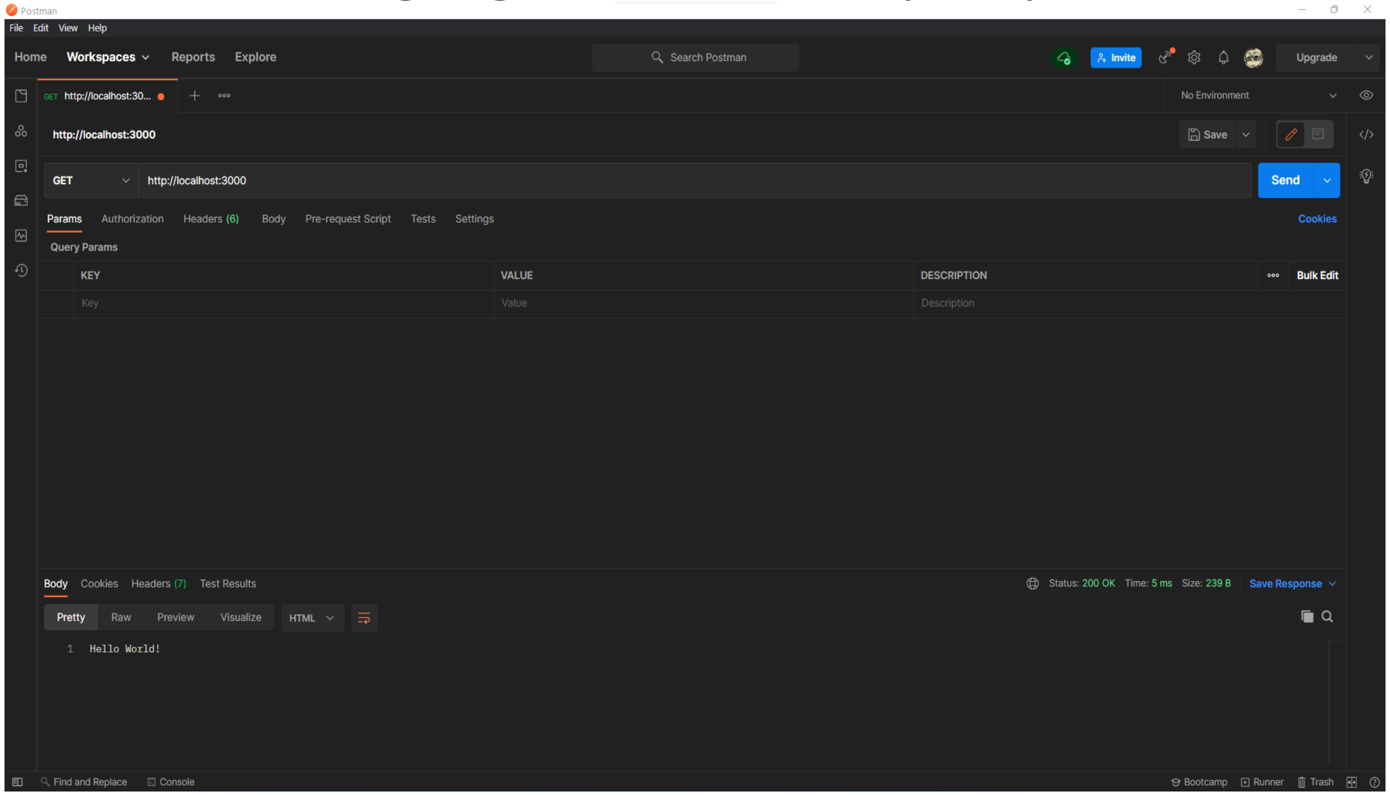
3. `curl` command-line tool

4. Call `fetch()` in web page

Postman

Postman is an application used for API testing

- Tests HTTP requests: GET, POST, PUT, PATCH, DELETE
- saving environments for later use, converting the API to code for various languages(like JavaScript, Python).


Postman
File Edit View Help
Home Workspaces Reports Explore
Search Postman
Invite
Upgrade
GET http://localhost:30...
No Environment
Save
Send
Params Authorization Headers (6) Body Pre-request Script Tests Settings
Cookies
Query Params
KEY VALUE DESCRIPTION Bulk Edit
Key Value Description
Body Cookies Headers (7) Test Results
Save Response
Pretty Raw Preview Visualize HTML
Find and Replace Console
Bootcamp Runner Trash

http://localhost:3000

GET http://localhost:3000

Status: 200 OK Time: 5 ms Size: 239 B

```
Hello World!
```

https://www.postman.com/

curl

`curl`:
- Command-line tool to send and receive data from a server
- using one of the supported protocols (HTTP, HTTPS, FTP, FTPS, GOPHER, DICT, TELNET, LDAP or FILE) (Manual)

`curl –d '…' –H '…'  -X` ***METHOD url***

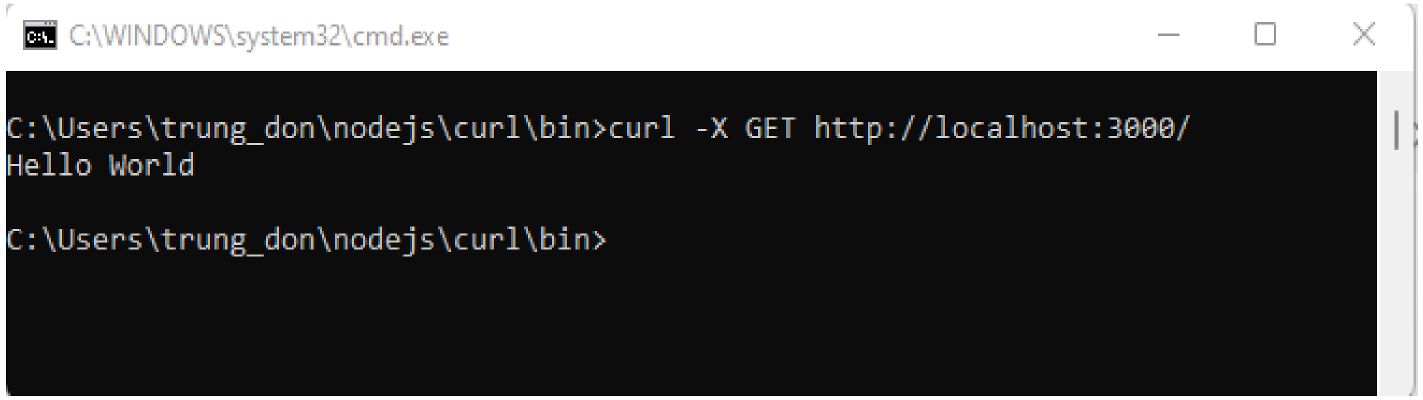
e.g.

```
$ curl -X GET http://localhost:3000/
```

curl

curl:

```
$ curl -X GET http://localhost:3000/
```

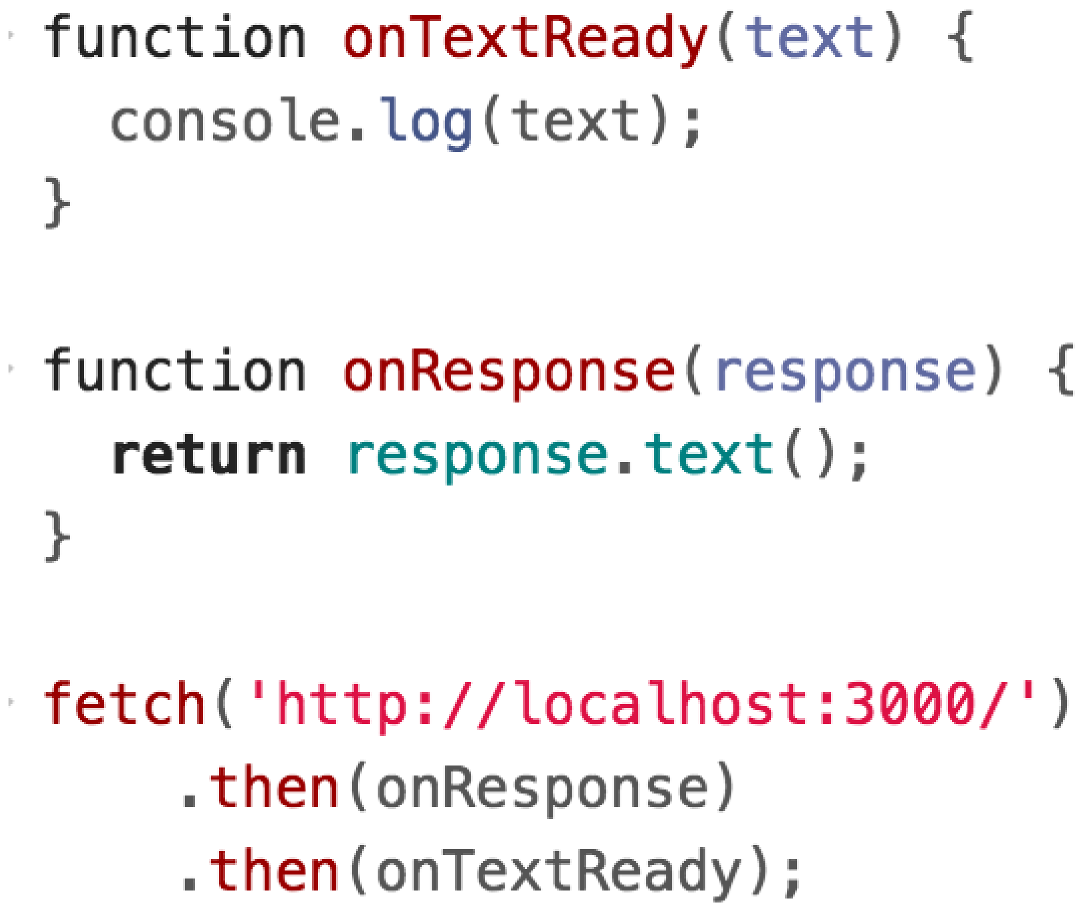


```
C:\Users\trung_don\nodejs\curl\bin>curl -X GET http://localhost:3000/
Hello World

C:\Users\trung_don\nodejs\curl\bin>
```

Querying with fetch()

The Fetch API is a modern interface that allows you to make HTTP requests to servers from web browsers

JavaScript client
code in a web page:



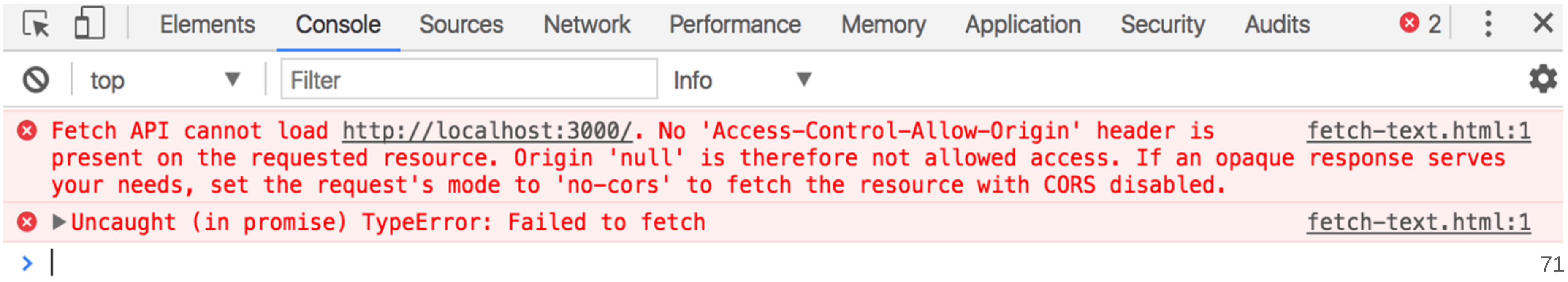
```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('http://localhost:3000/')
    .then(onResponse)
    .then(onTextReady);
```

fetch() to localhost

But if we try fetching to localhost from file://

```
fetch('http://localhost:3000')
    .then(onResponse)
    .then(onTextReady);
```

We get this CORS error:



```
Fetch API cannot load http://localhost:3000/. No 'Access-Control-Allow-Origin' header is      fetch-text.html:1
present on the requested resource. Origin 'null' is therefore not allowed access. If an opaque response serves
your needs, set the request's mode to 'no-cors' to fetch the resource with CORS disabled.
▶Uncaught (in promise) TypeError: Failed to fetch                                               fetch-text.html:1
```

CORS

CO**R**S: **C**ross-**O**rigin **R**esource **S**haring (wiki)

- Browser policy for what resources a web page can load
- An origin = protocol + host + port
    - Two URLs have the same origin if the ***protocol, port (if specified), and host*** are the same for both.

http://example.com/app1/index.html http://example.com/app2/index.html	Giống nhau
http://Example.com:80 http://example.com	Giống nhau
http://example.com/app1 https://example.com/app2	Khác nhau
http://example.com http://www.example.com http://myapp.example.com	Khác nhau
http://example.com http://example.com:8080	Khác nhau

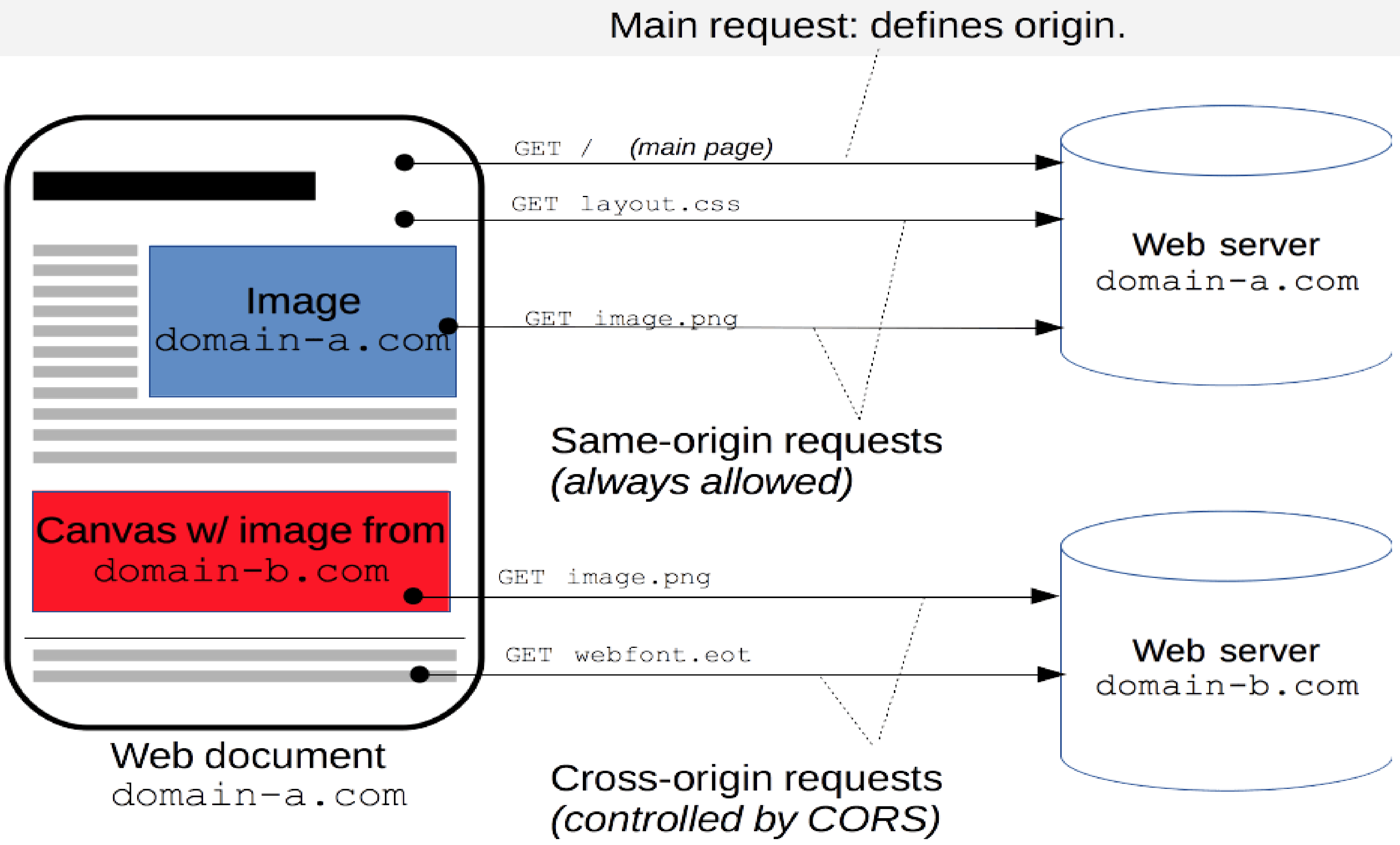
CORS

CORS: **C**ross-**O**rigin **R**esource **S**haring (wiki)

- You **cannot** make cross-origin requests by default for:
  - Resources loaded via `fetch()` or XHR
- The problem is that we are trying to `fetch()` **http://localhost:3000** from **file:///**

Since the two resources have different origins, this is disallowed by default CORS policy

CORS



https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

Cross-origin solutions

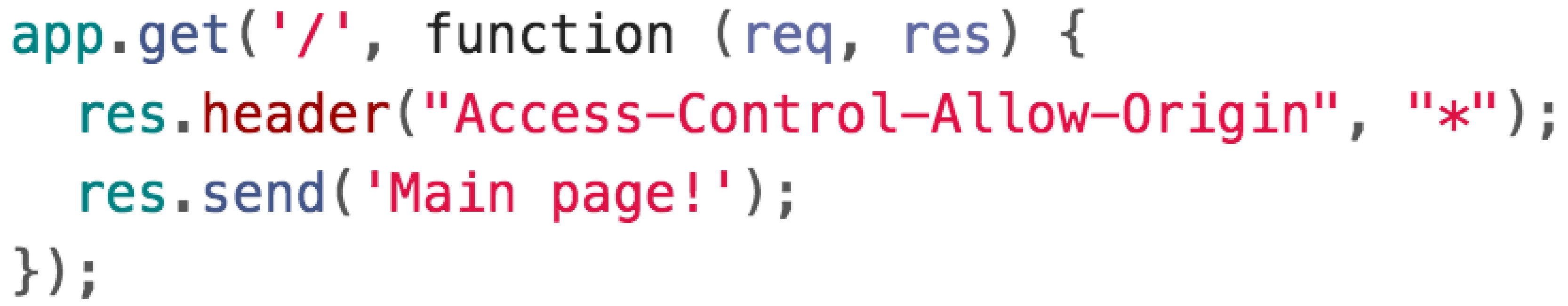
The problem is that we are trying to `fetch()` **http://localhost:3000** from **file:///**

Two ways to solve this:

1. Change the server running on localhost:3000 to allow cross-origin requests, i.e. to allow requests from different origins (such as file:///)

2. **Preferred solution:** Load the frontend code statically from the same server, so that the request is from the same origin

Solution 1: Enable CORS



```
app.get('/', function (req, res) {
  res.header("Access-Control-Allow-Origin", "*");
  res.send('Main page!');
});
```

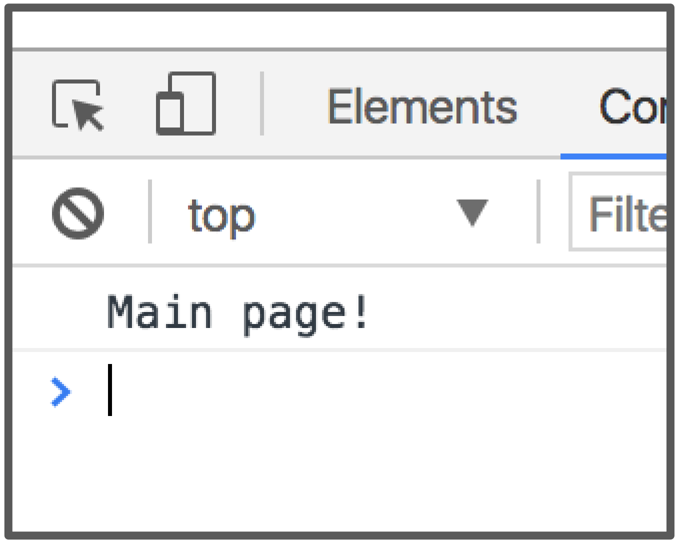
You can set an Access-Control-Allow-Origin HTTP header before sending your response

- This is the server saying to the browser in its response: "Hey browser, I'm totally fine with websites of any origin requesting this file"

Solution 1: Enable CORS

Now the fetch will succeed:

```
function onTextReady(text) {
  console.log(text);
}

function onResponse(response) {
  return response.text();
}

fetch('http://localhost:3000/')
    .then(onResponse)
    .then(onTextReady);
```



```
Main page!
>
```

Cross-origin solutions

However, you wouldn't have to enable CORS at all if you were making requests from the same origin

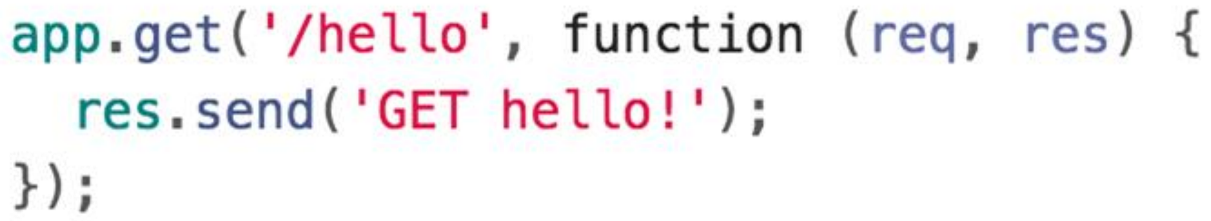
Preferred solution: Load the frontend code statically from the same server, so that the request is from the same origin

Recall: Web services

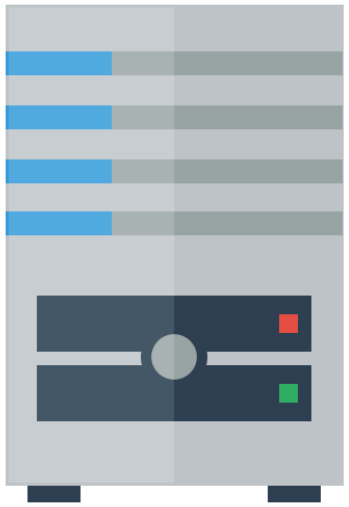
Sometimes when you type a URL into your browser, the URL represents **an API endpoint**

That is, the URL represents **a parameterized request**, and the web server dynamically generates a response to that request

That's how our NodeJS server treats routes defined like this:



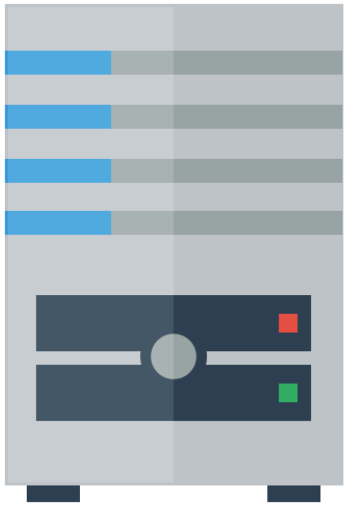
```
app.get('/hello', function (req, res) {
  res.send('GET hello!');
});
```



Recall: Servers

Other times when you type a URL in your browser, the URL is a **path to a file** on the hard drive of the server:

- The web server software grabs that file from the server's local file system, and sends back its contents to you



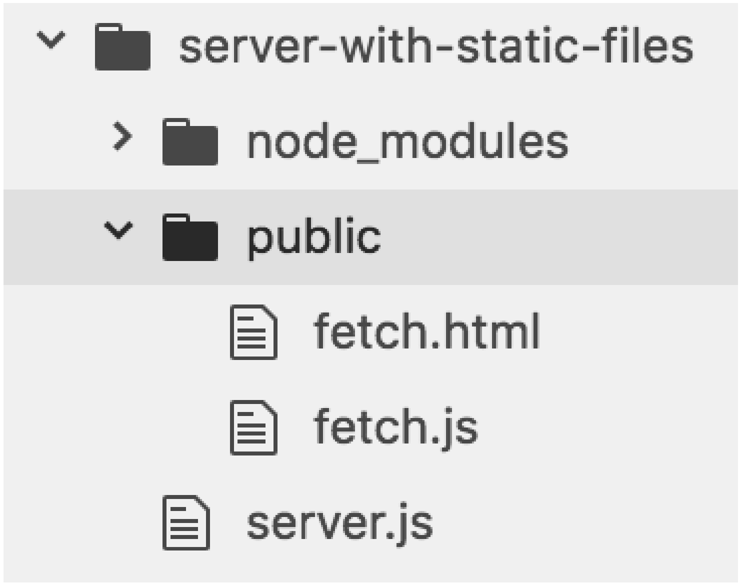
We can make our NodeJS server also sometimes serve files "statically," meaning instead of treating **all** URLs as API endpoints, some URLs will be treated as file paths

Solution 2: Statically served files

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

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

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

This line of code makes our server now start serving the files in the 'public' directory directly

Server static data


```
server-with-static-files
  node_modules
  public
    fetch.html
    fetch.js
  server.js
```

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

Now Express will serve:

http://localhost:3000/fetch.html

http://localhost:3000/fetch.js

Express looks up the files relative to the static directory, so the name of the static directory ("public" in this case) is not part of the URL

Sending data to the server

Route parameters

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

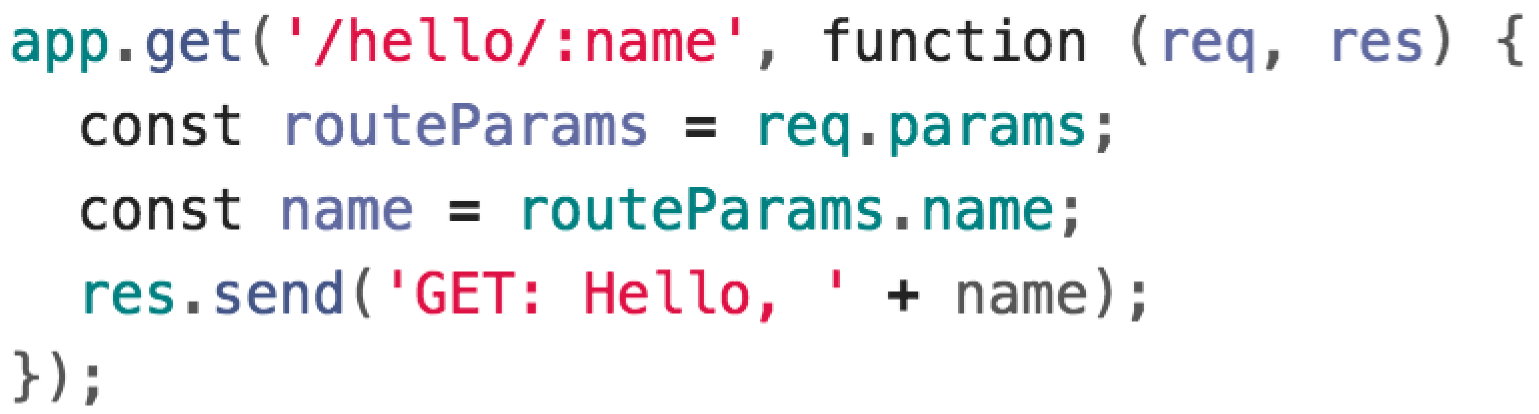
Example:
`https://jsonplaceholder.typicode.com/users/10`

The last part of the URL is a **parameter** representing the user id, which is 10

Route parameters

Q: How do we read route parameters in our server?

A: We can use the **:*variableName*** syntax in the path to specify a route parameter (Express docs):

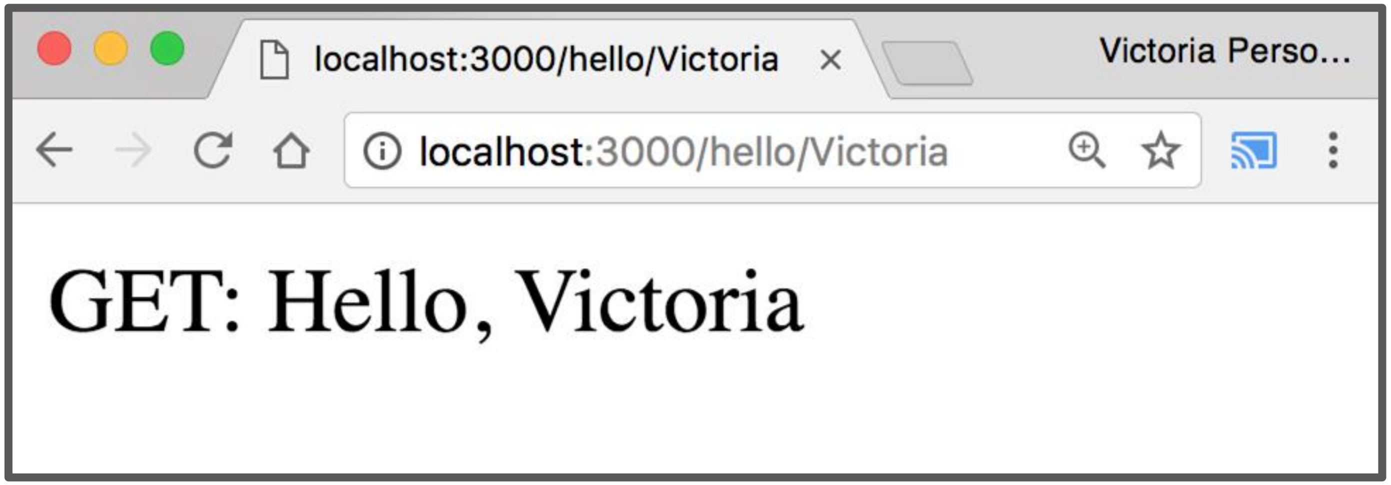


```
app.get('/hello/:name', function (req, res) {
  const routeParams = req.params;
  const name = routeParams.name;
  res.send('GET: Hello, ' + name);
});
```

We can access the route parameters via **`req.params`**

Route parameters

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

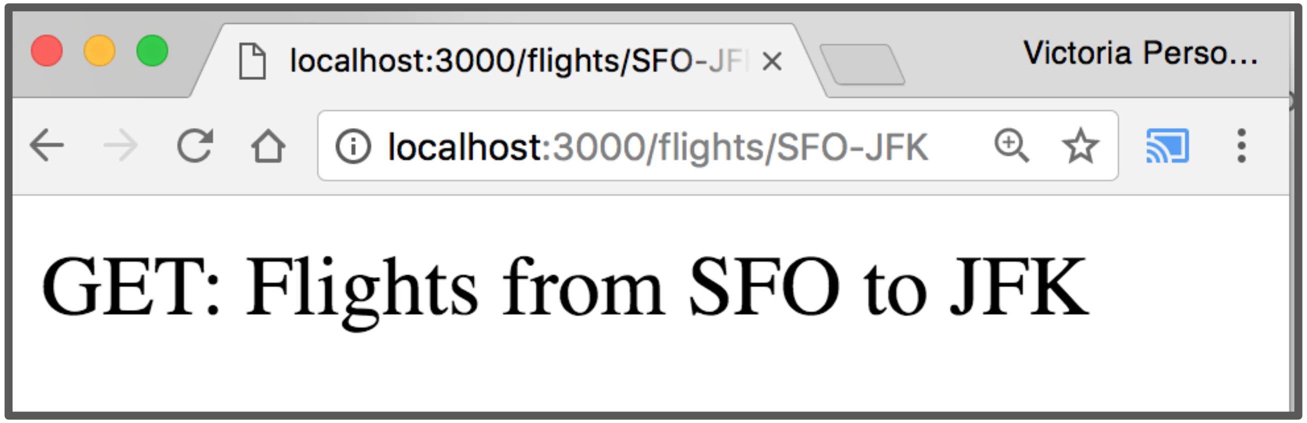


GET: Hello, Victoria

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);
});
```



localhost:3000/flights/SFO-JFK

GET: Flights from SFO to JFK

Query parameters

Example:

`https://jsonplaceholder.typicode.com/posts?userId=1`

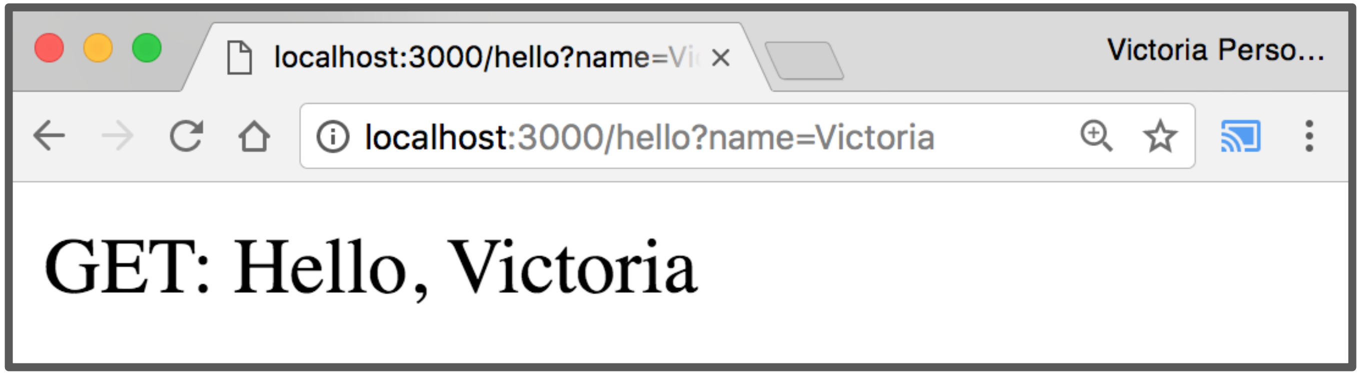
The query parameter sent to the server endpoint is userId, whose value is 1

Query parameters

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

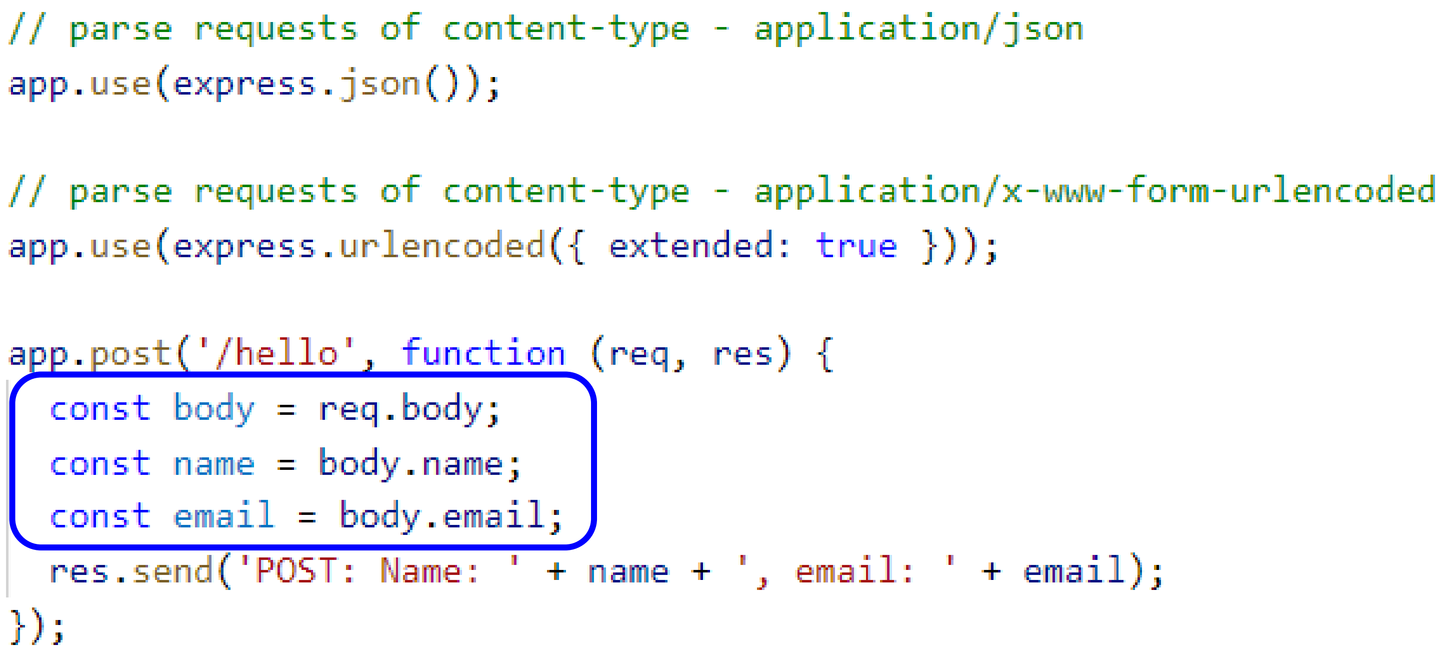
A: We can access query parameters via **`req.query`**:

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


localhost:3000/hello?name=Vi
Victoria Perso...
localhost:3000/hello?name=Victoria

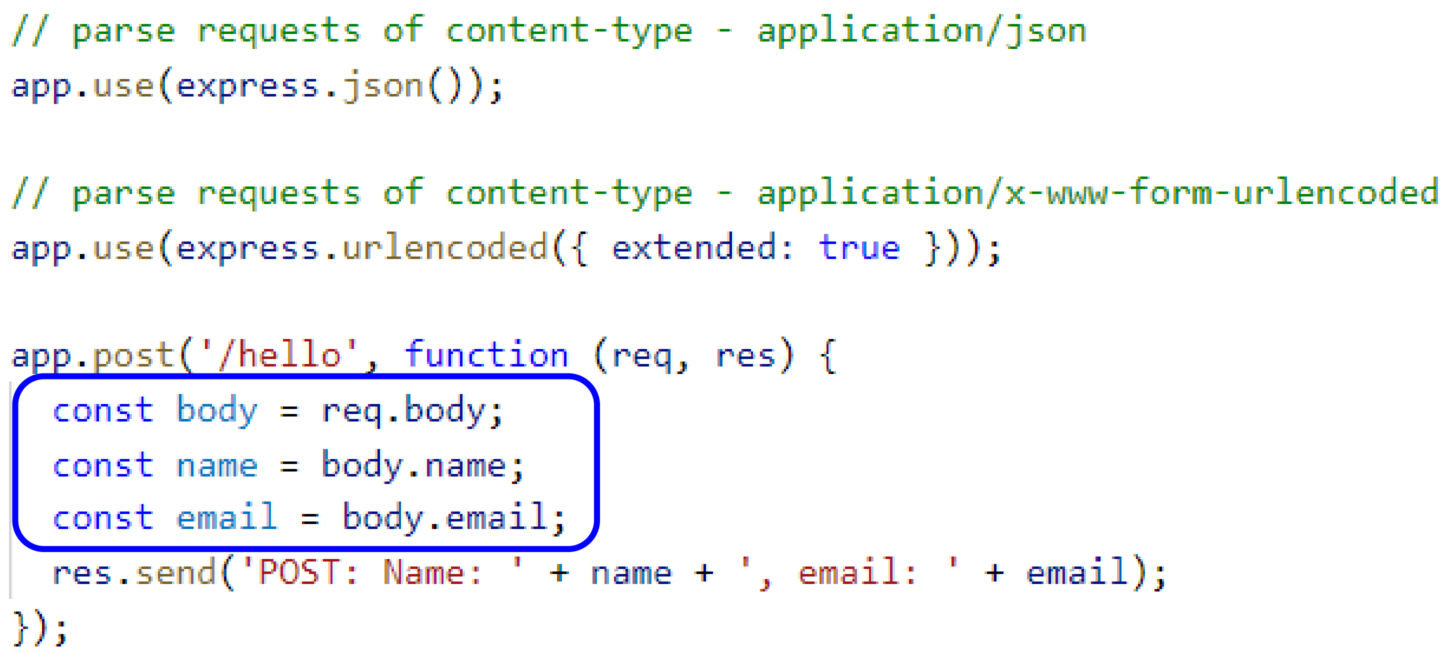
GET: Hello, Victoria

POST message body



```
// parse requests of content-type - application/json
app.use(express.json());

// parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));

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

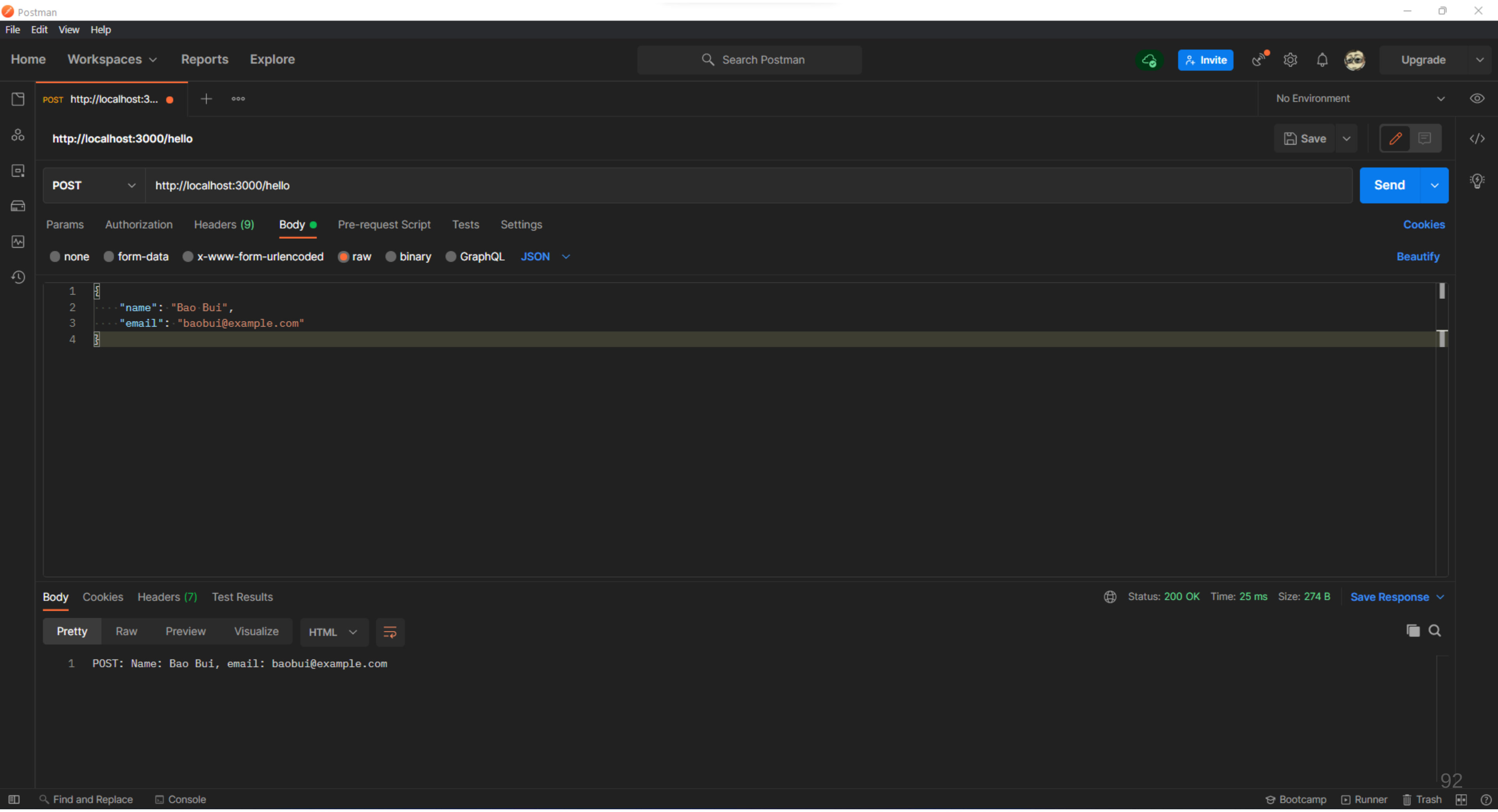
POST message body



```
// parse requests of content-type - application/json
app.use(express.json());

// parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));

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

```
Content-Type: application/json
{"name": "Bao Bui", "email": "bao@example.com"}

Content-Type: application/x-www-form-urlencoded
name=Bao%20Bui&email=bao%40example.com
```

POST message body


Postman
File Edit View Help
Home Workspaces Reports Explore
Search Postman
Invite
Upgrade
No Environment
Save
Send
Params Authorization Headers (9) Body Pre-request Script Tests Settings
Cookies
Beautify
Pretty Raw Preview Visualize HTML
Find and Replace Console
Bootcamp Runner Trash

POST http://localhost:3000/hello

Body: none / form-data / x-www-form-urlencoded / [✓ raw] / binary / GraphQL — JSON

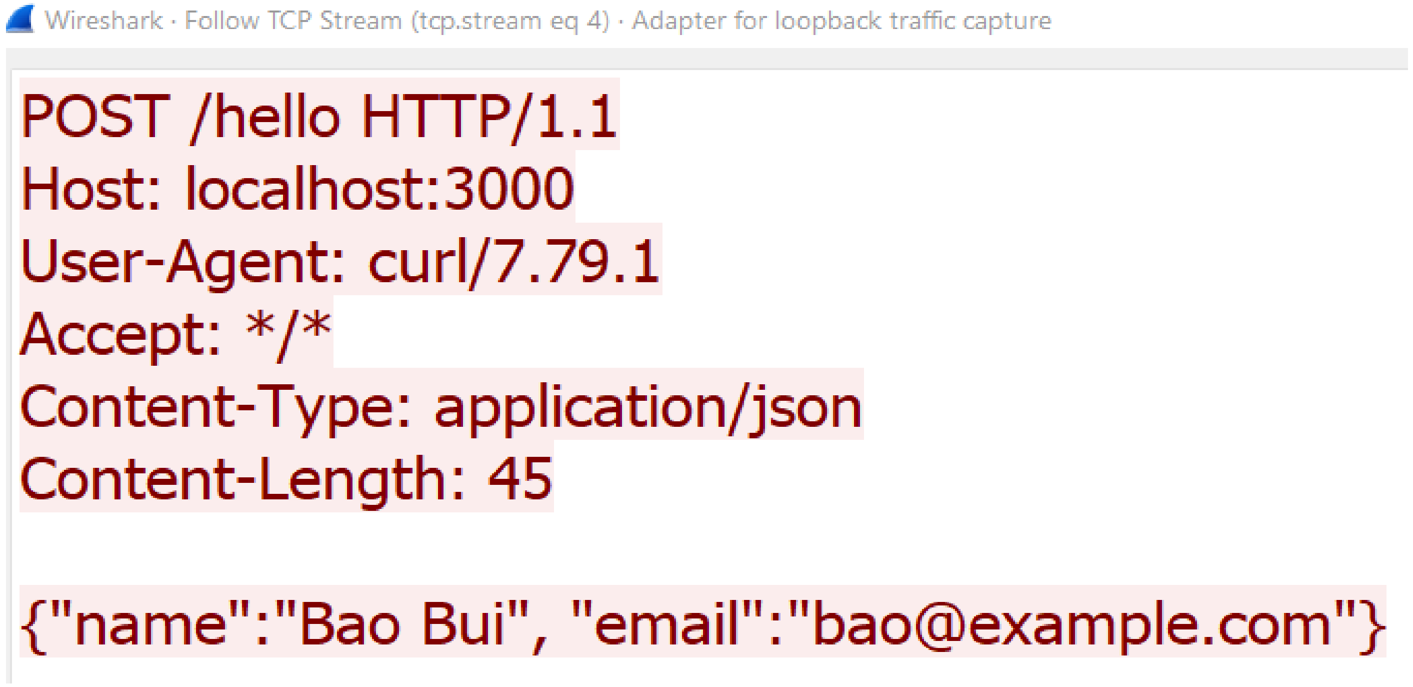
```
{
    "name": "Bao Bui",
    "email": "baobui@example.com"
}
```

Body Cookies Headers (7) Test Results — Status: 200 OK Time: 25 ms Size: 274 B Save Response

```
POST: Name: Bao Bui, email: baobui@example.com
```

POST message body

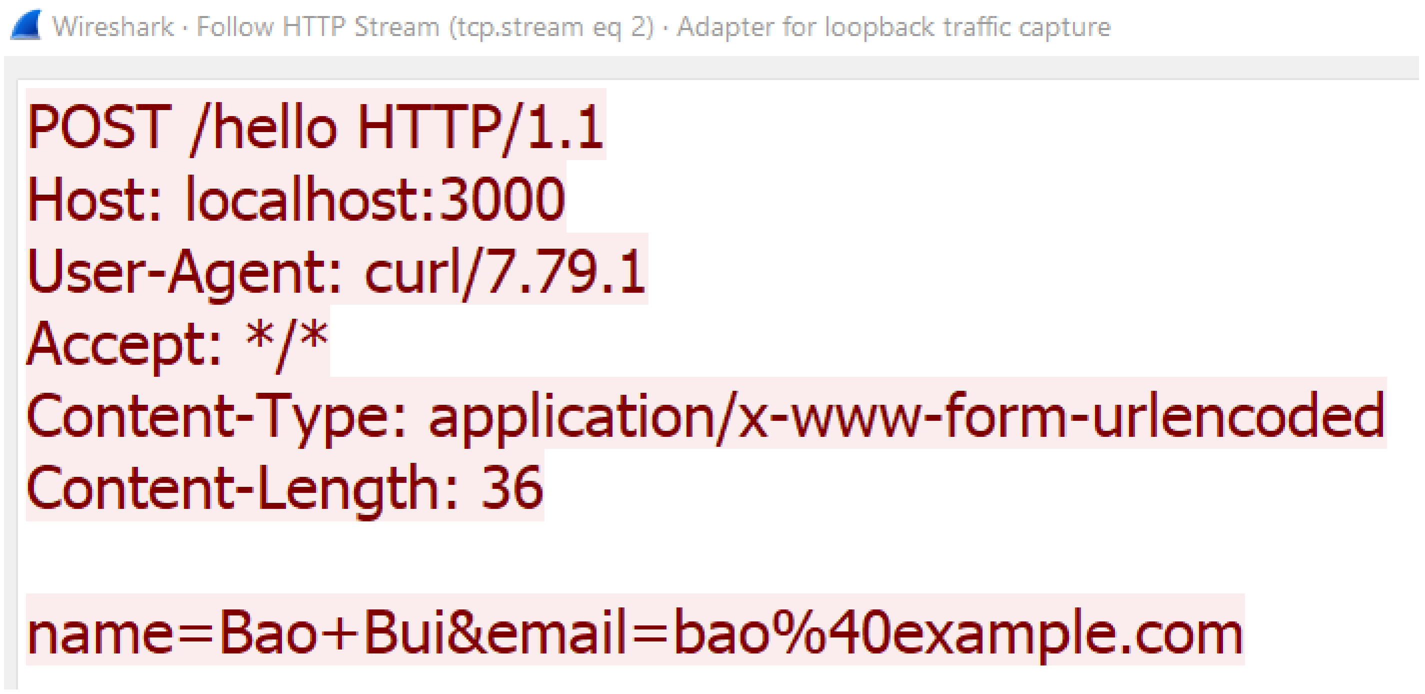
```
$ curl -d '{"name":"Bao Bui", "email":"bao@example.com"}'
-H 'Content-Type: application/json' -X POST
http://localhost:3000/hello
```


Wireshark · Follow TCP Stream (tcp.stream eq 4) · Adapter for loopback traffic capture

```
POST /hello HTTP/1.1
Host: localhost:3000
User-Agent: curl/7.79.1
Accept: */*
Content-Type: application/json
Content-Length: 45

{"name":"Bao Bui", "email":"bao@example.com"}
```

POST message body

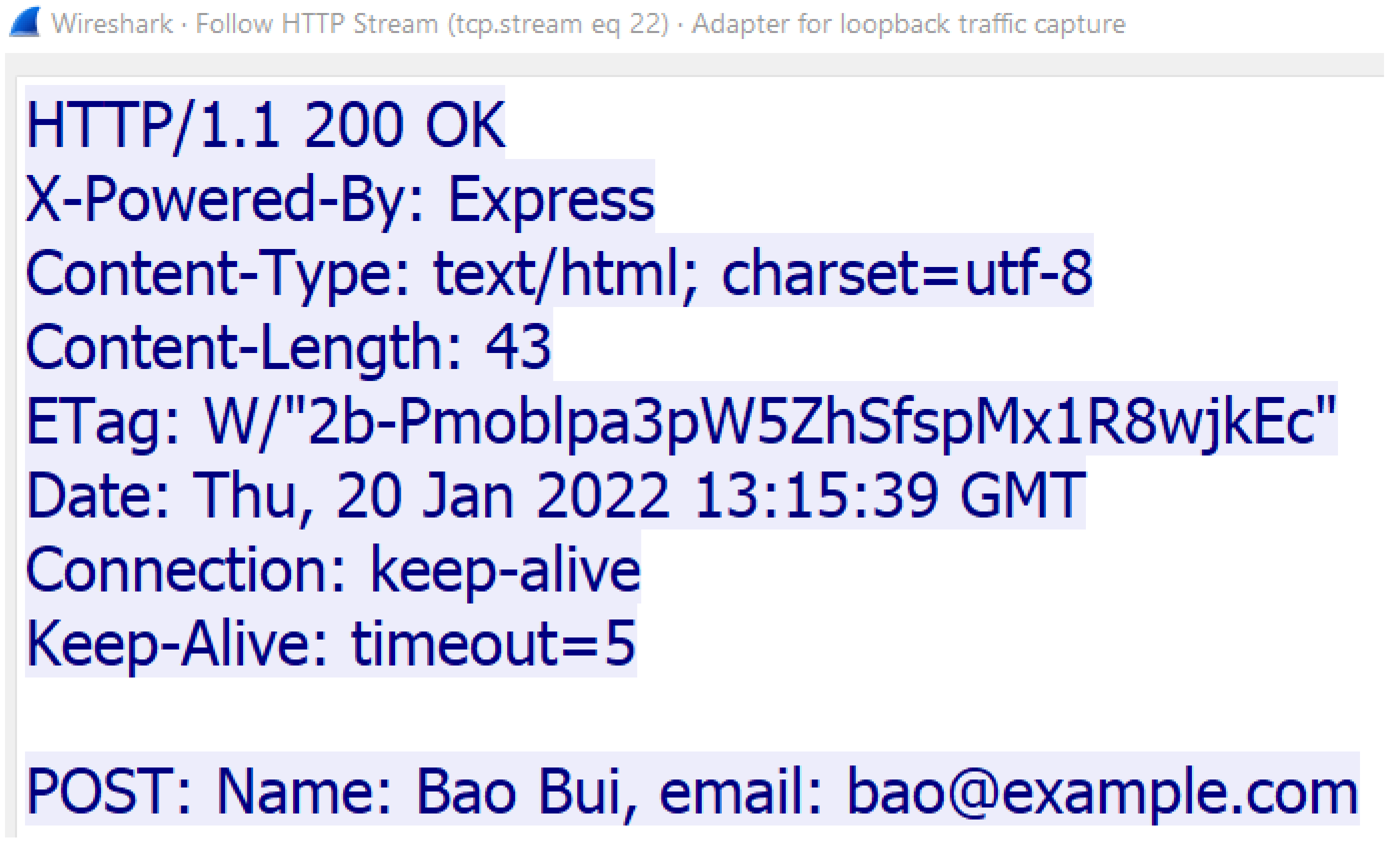
```
$ curl --data-urlencode 'name=Bao Bui' --data-urlencode
'email=bao@example.com' -H 'Content-Type:
application/x-www-form-urlencoded' -X POST
http://localhost:3000/hello
```


Wireshark · Follow HTTP Stream (tcp.stream eq 2) · Adapter for loopback traffic capture

```
POST /hello HTTP/1.1
Host: localhost:3000
User-Agent: curl/7.79.1
Accept: */*
Content-Type: application/x-www-form-urlencoded
Content-Length: 36

name=Bao+Bui&email=bao%40example.com
```

POST message body

HTTP Response Message


Wireshark · Follow HTTP Stream (tcp.stream eq 22) · Adapter for loopback traffic capture

```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: text/html; charset=utf-8
Content-Length: 43
ETag: W/"2b-Pmoblpa3pW5ZhSfspMx1R8wjkEc"
Date: Thu, 20 Jan 2022 13:15:39 GMT
Connection: keep-alive
Keep-Alive: timeout=5

POST: Name: Bao Bui, email: bao@example.com
```

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:
  - PUT/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 HTTP API

Middleware

Middleware

Middleware are functions that Express

- Executes in the middle after the incoming request
- Then produces an output which could either be the final output or be used by the next middleware

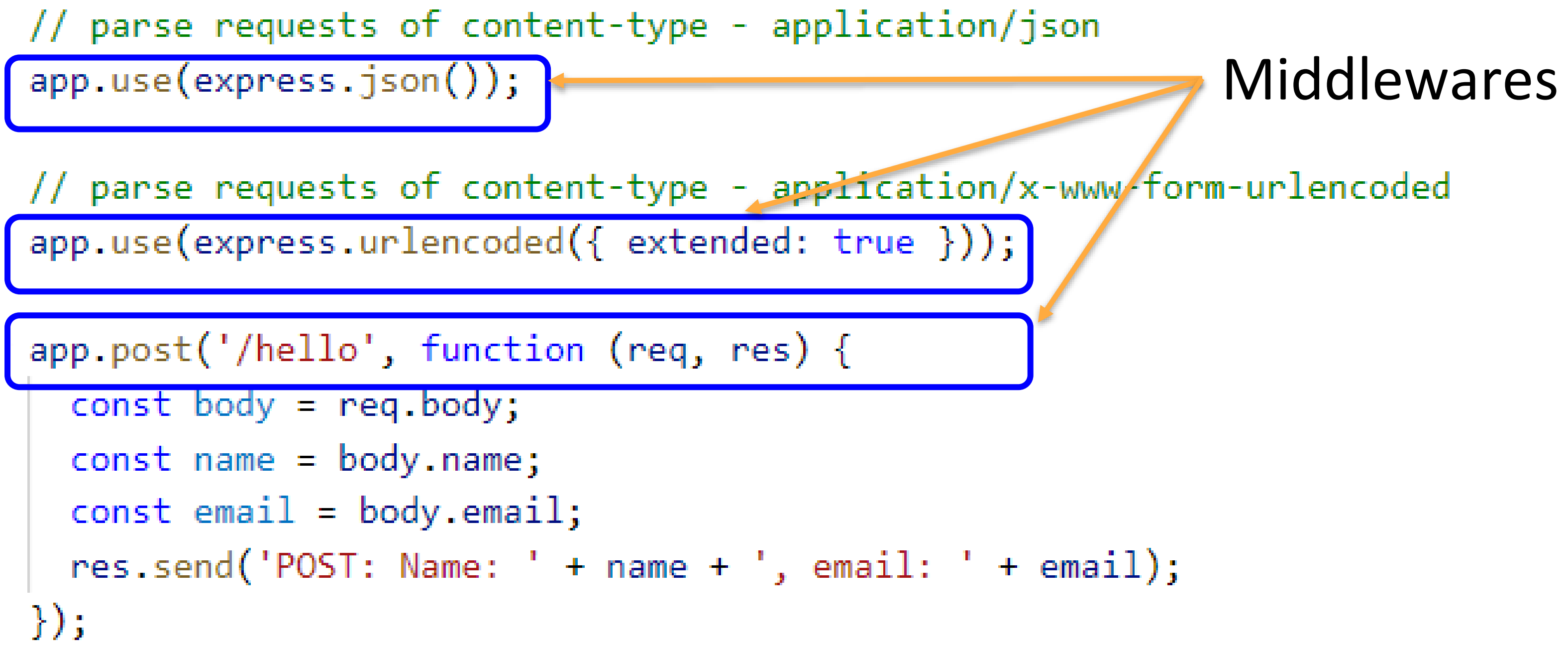
Middleware stack is a structure storing middleware in specified order and resolve these middleware into a single handler.

Middleware functions can perform the following tasks:

- Execute any code.
- Make changes to the request and the response objects.
- End the request-response cycle.
- Call the next middleware function in the stack.

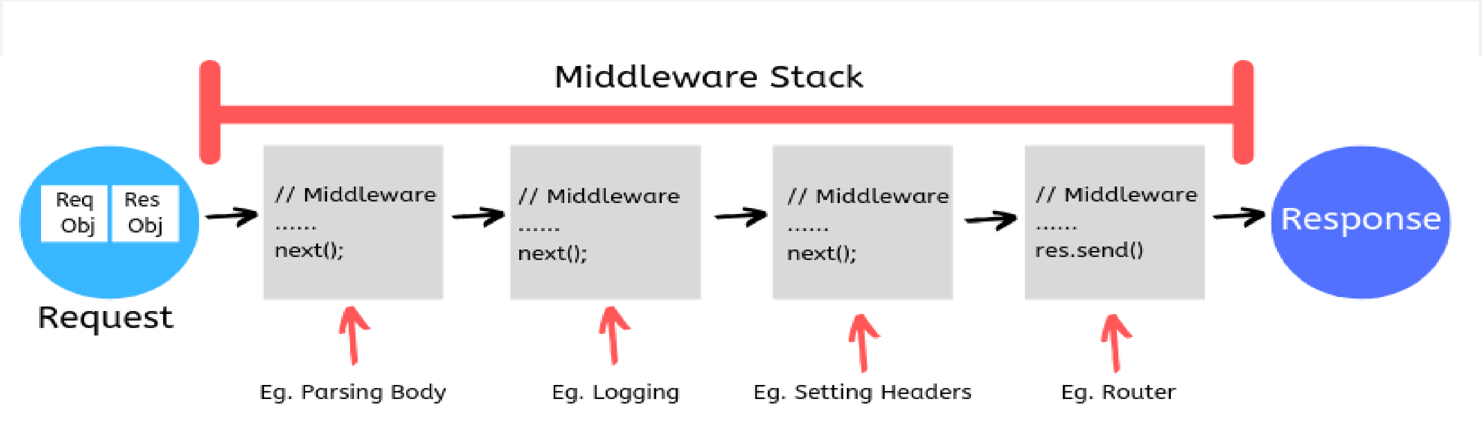
Middleware

when we have *app.use(…),* we are in fact applying middlewares to Express

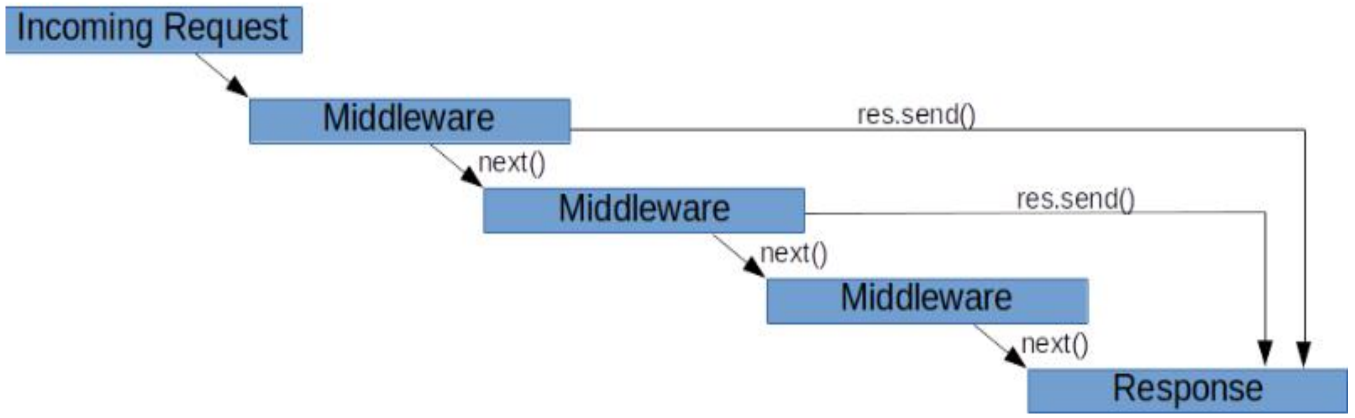


```
// parse requests of content-type - application/json
app.use(express.json());

// parse requests of content-type - application/x-www-form-urlencoded
app.use(express.urlencoded({ extended: true }));

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

Middleware



The middleware stack follows **the order of middlewares placed in the code**



*Nguồn: https://iq.opengenus.org/middlewares-in-express/

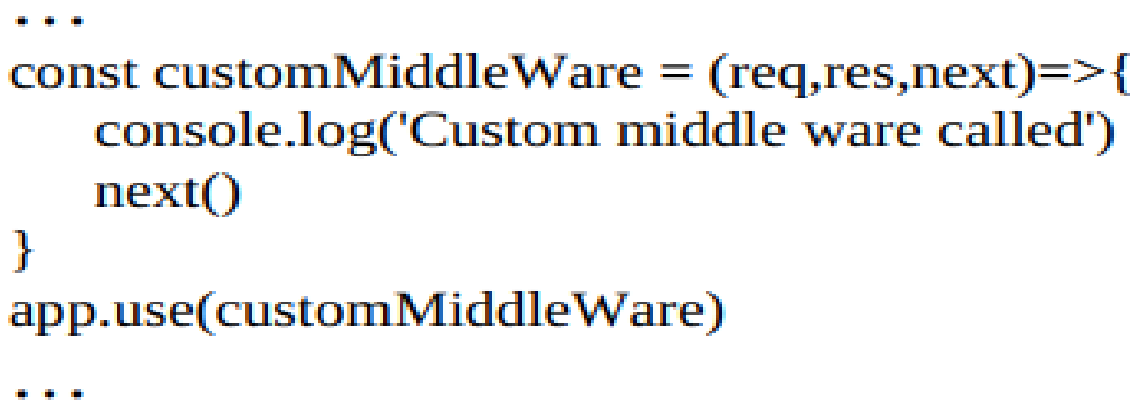
Middleware

Custom middleware

function(req, res, next) { ... };

Every function in this stack takes three arguments: *req, res, next*

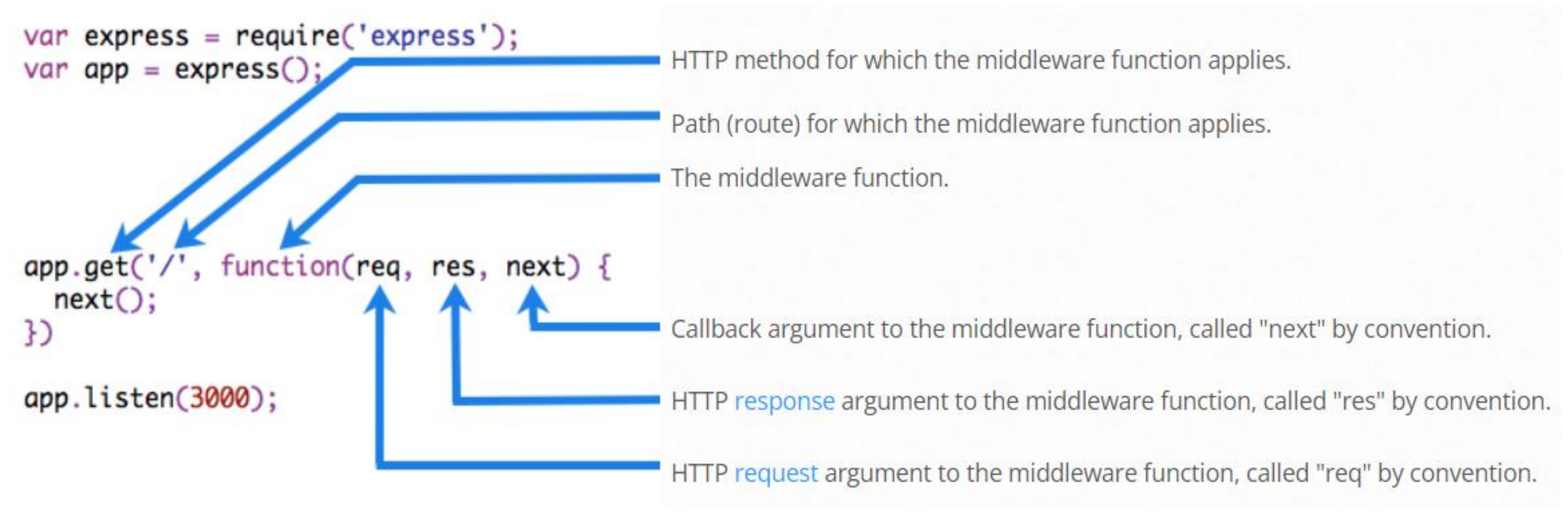
- Two arguments: *req* (a object for the request), *res* (a object for the response)
- The third argument: *next*
  - ➢ is itself a function
  - ➢ is called, Express will go on to the next function in the stack



```
...
const customMiddleWare = (req,res,next)=>{
    console.log('Custom middle ware called')
    next()
}
app.use(customMiddleWare)
...
```

Middleware

The following figure shows the elements of a middleware function call:



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

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

app.listen(3000);
```

HTTP method for which the middleware function applies.
Path (route) for which the middleware function applies.
The middleware function.
Callback argument to the middleware function, called "next" by convention.
HTTP response argument to the middleware function, called "res" by convention.
HTTP request argument to the middleware function, called "req" by convention.

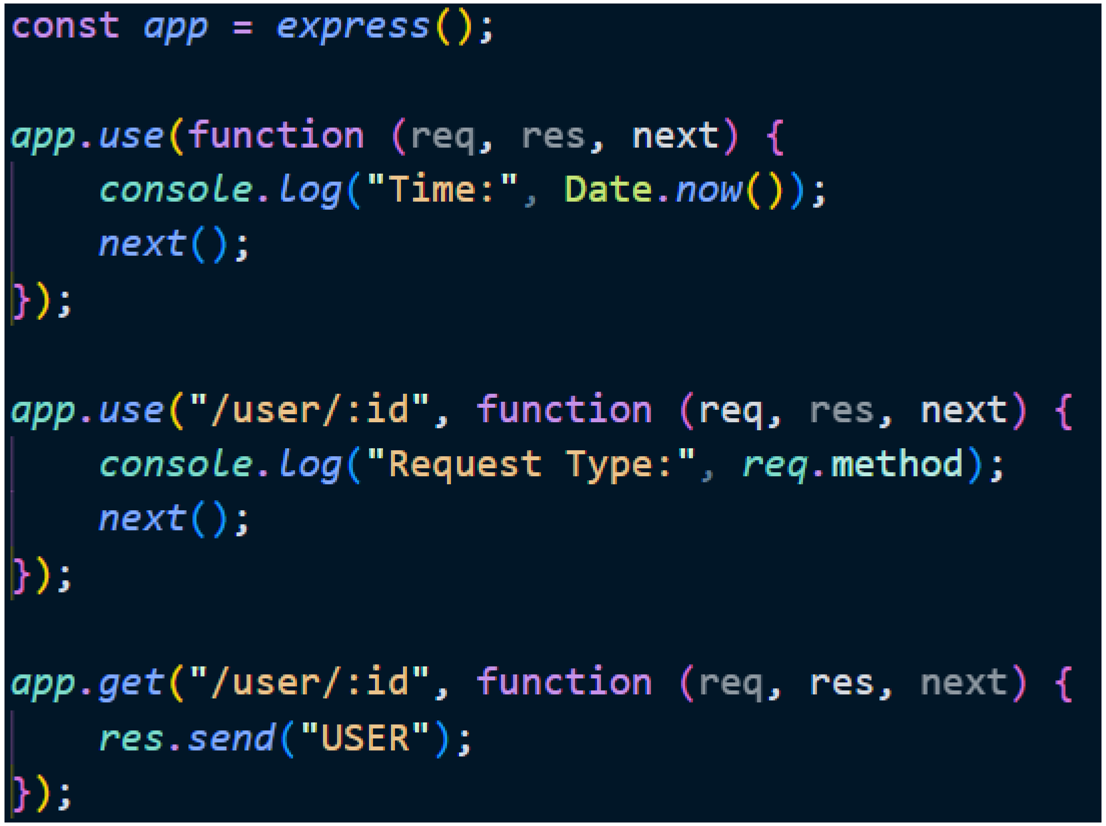
Middleware

Types of middleware:

- Application-level middleware
- Router-level middleware
- Error handling middleware
- Built-in middleware
- External middleware (requires `npm install`)

Middleware

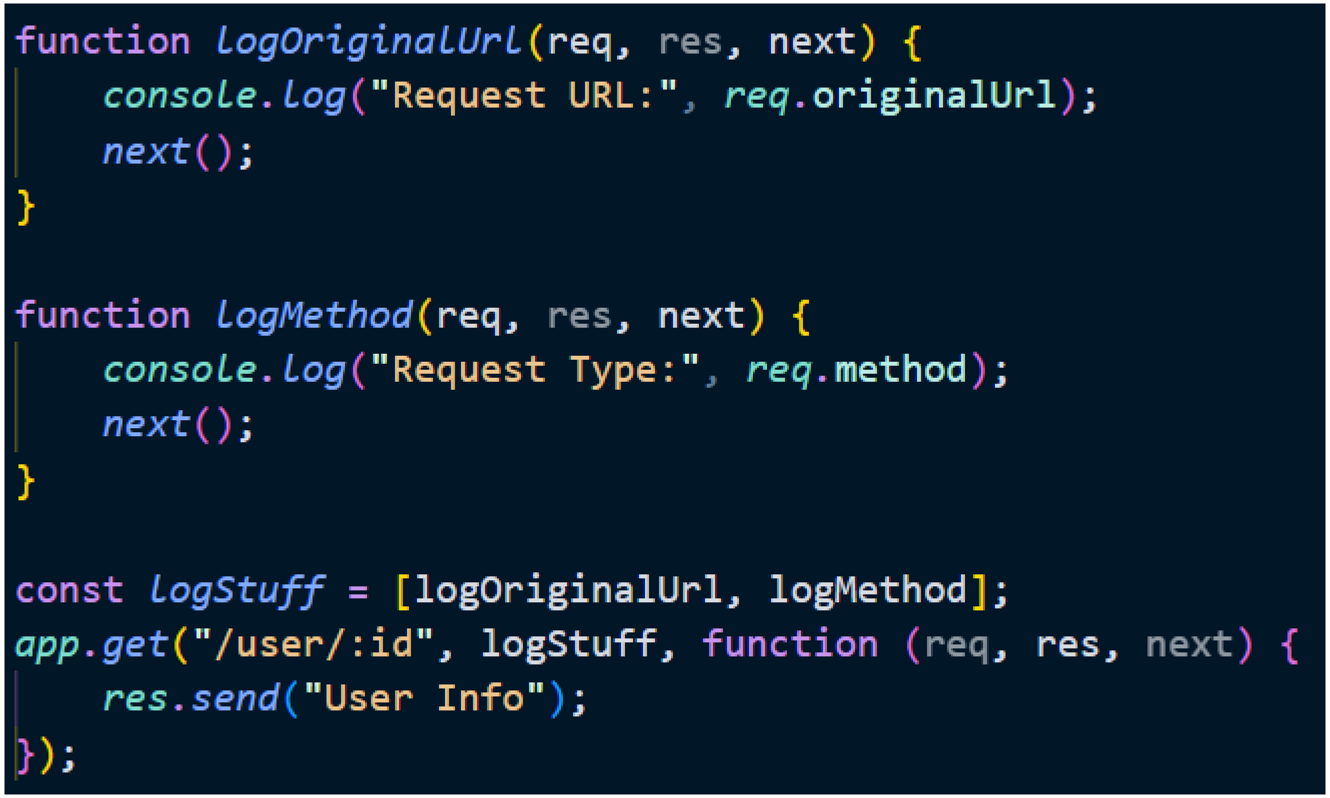
Application-level middleware: bound to the app instance by using `app.use()` or `app.METHOD()` functions



```
const app = express();

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

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

app.get("/user/:id", function (req, res, next) {
    res.send("USER");
});
```

Middleware

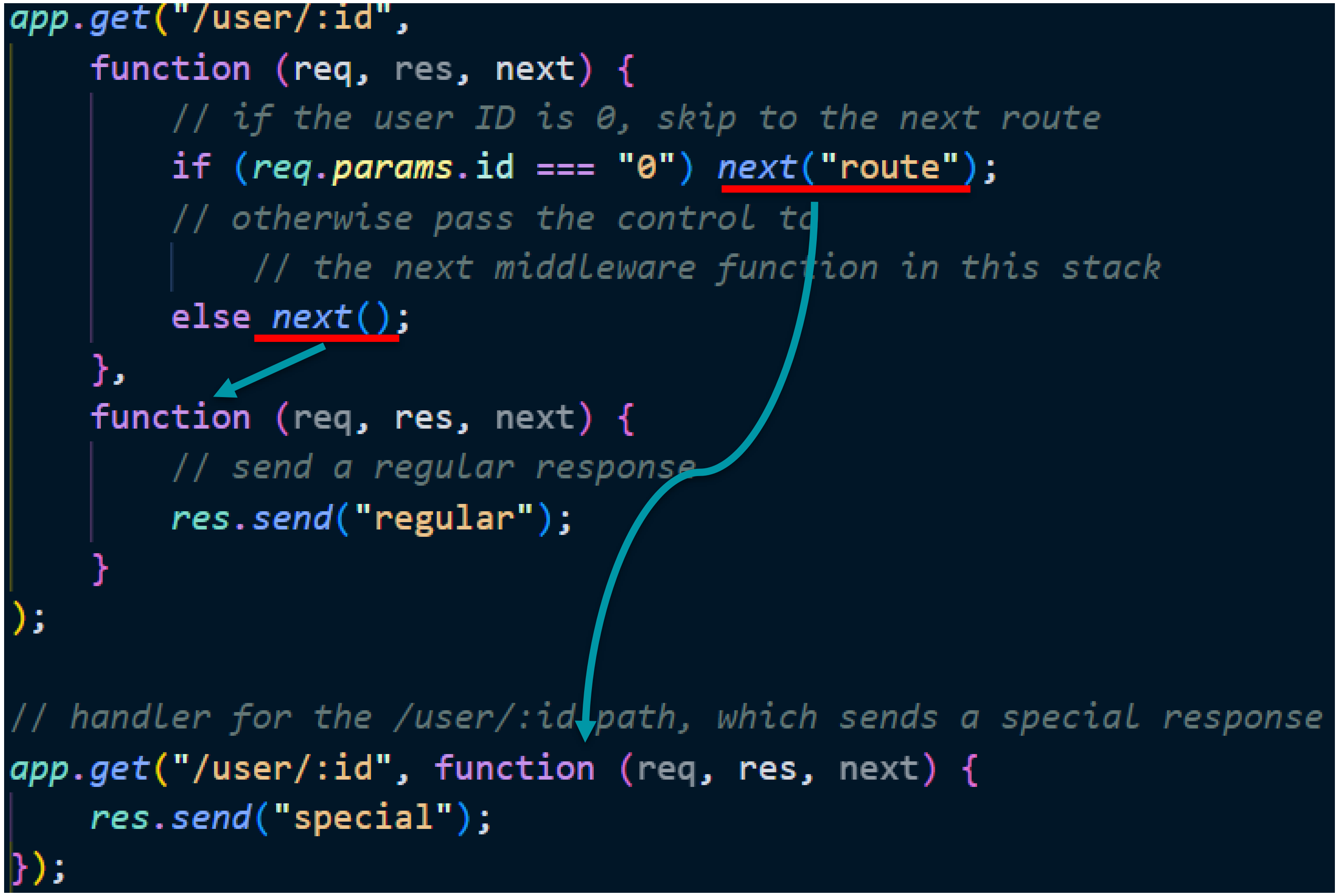
Application-level middleware: bound to the app instance by using `app.use()` or `app.METHOD()` functions



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

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

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

Middleware

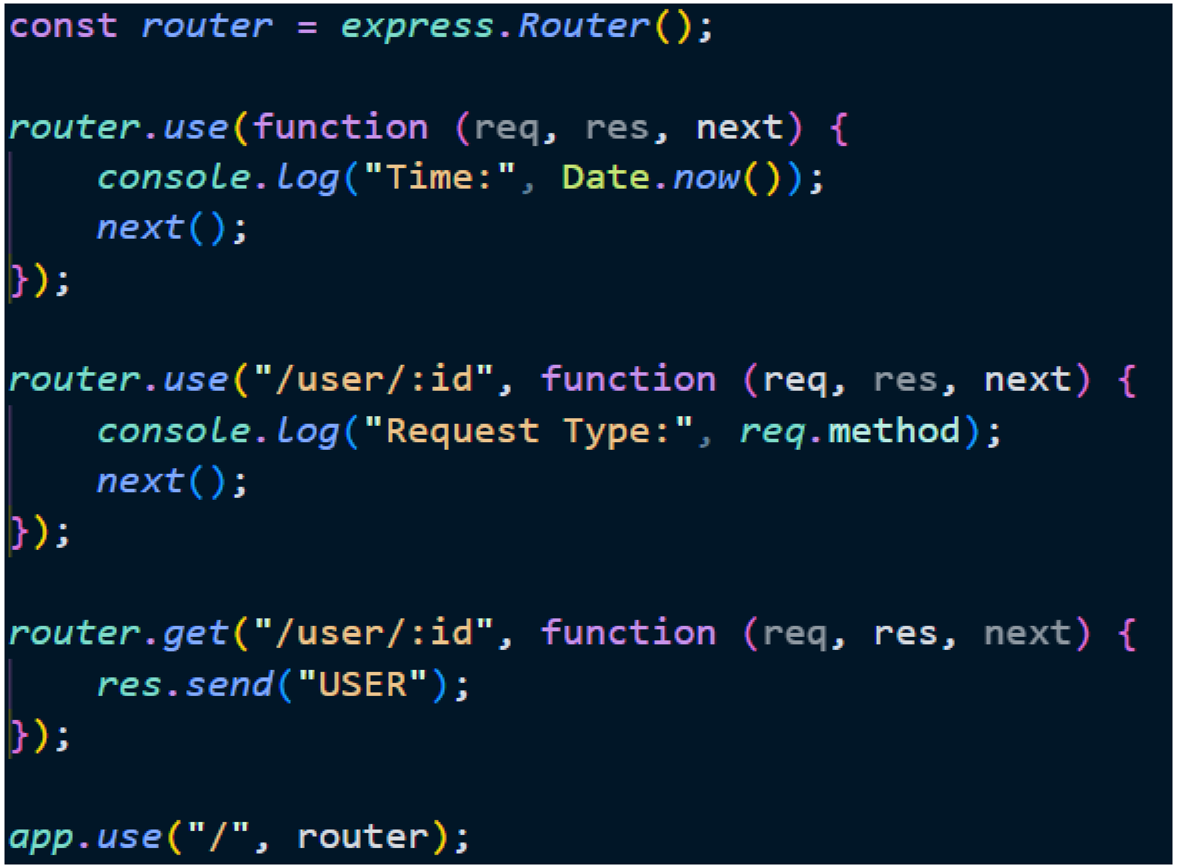
Application-level middleware



```
app.get("/user/:id",
    function (req, res, next) {
        // if the user ID is 0, skip to the next route
        if (req.params.id === "0") next("route");
        // otherwise pass the control to
            // the next middleware function in this stack
        else next();
    },
    function (req, res, next) {
        // send a regular response
        res.send("regular");
    }
);

// handler for the /user/:id path, which sends a special response
app.get("/user/:id", function (req, res, next) {
    res.send("special");
});
```

Middleware

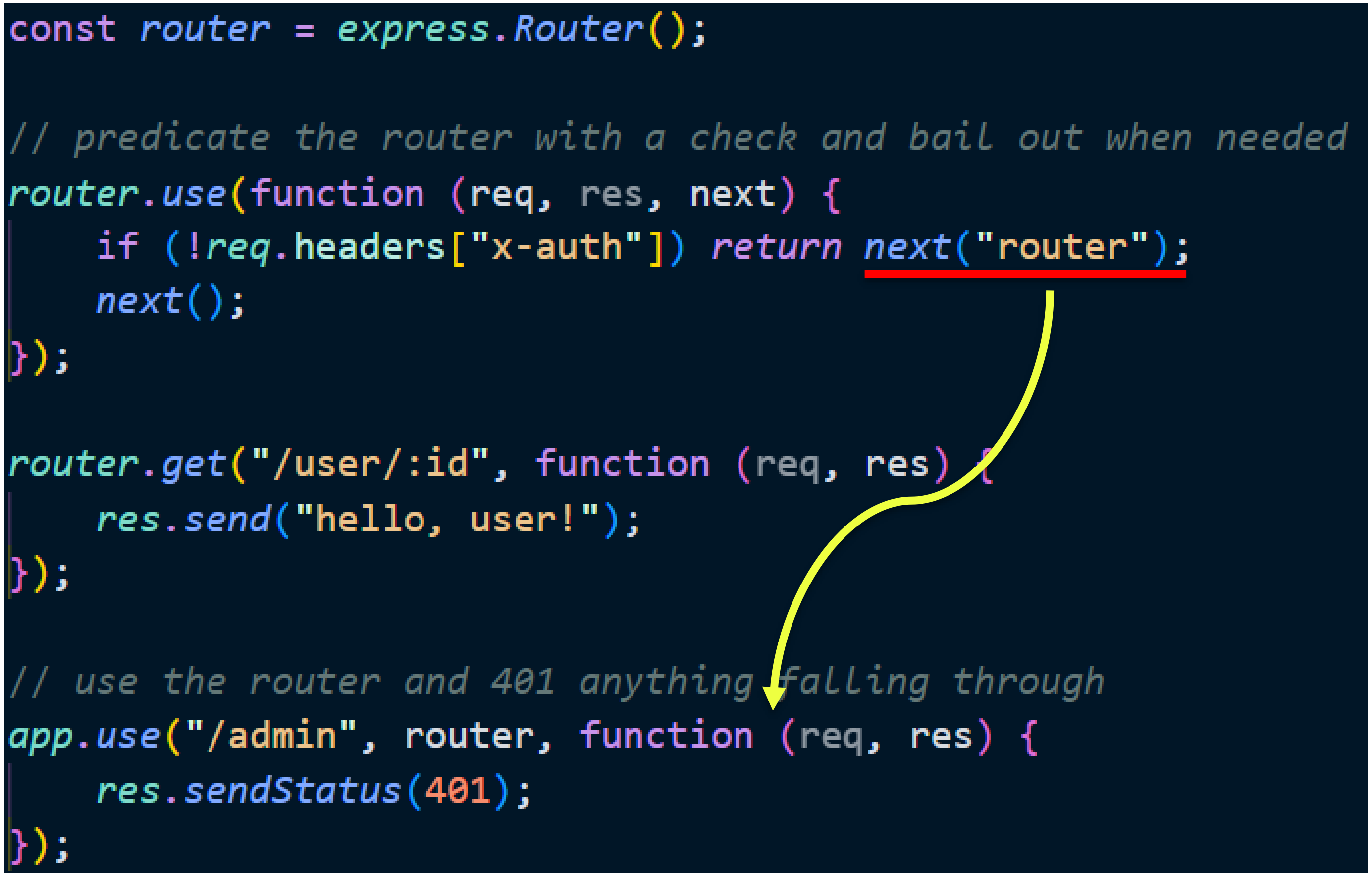
Router-level middleware: bound to an instance of express.Router()



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

router.use(function (req, res, next) {
    console.log("Time:", Date.now());
    next();
});

router.use("/user/:id", function (req, res, next) {
    console.log("Request Type:", req.method);
    next();
});

router.get("/user/:id", function (req, res, next) {
    res.send("USER");
});

app.use("/", router);
```

Middleware

Router-level middleware: bound to an instance of express.Router()



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

// predicate the router with a check and bail out when needed
router.use(function (req, res, next) {
    if (!req.headers["x-auth"]) return next("router");
    next();
});

router.get("/user/:id", function (req, res) {
    res.send("hello, user!");
});

// use the router and 401 anything falling through
app.use("/admin", router, function (req, res) {
    res.sendStatus(401);
});
```

Middleware

Error handling:

- Express comes with a built-in error handler

  + that takes care of any errors that might be encountered in the app

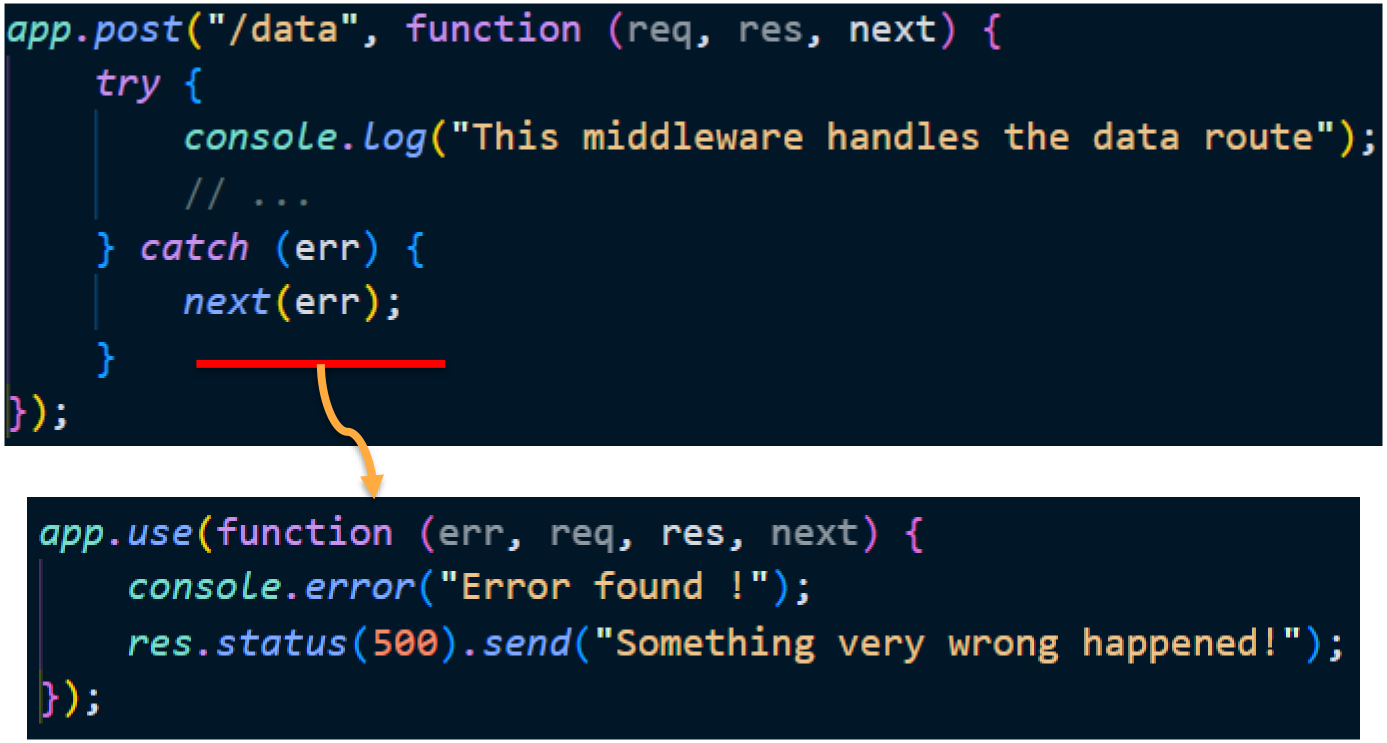
  +This default error-handling middleware function is added at the end of the middleware function stack

- The built-in error handler will be handled unless you have a custom error handler.

Middleware

Writing error handlers:

- Error-handling functions have four arguments instead of three: (err, req, res, next)
- Define error-handling middleware last, after other app.use() and routes calls



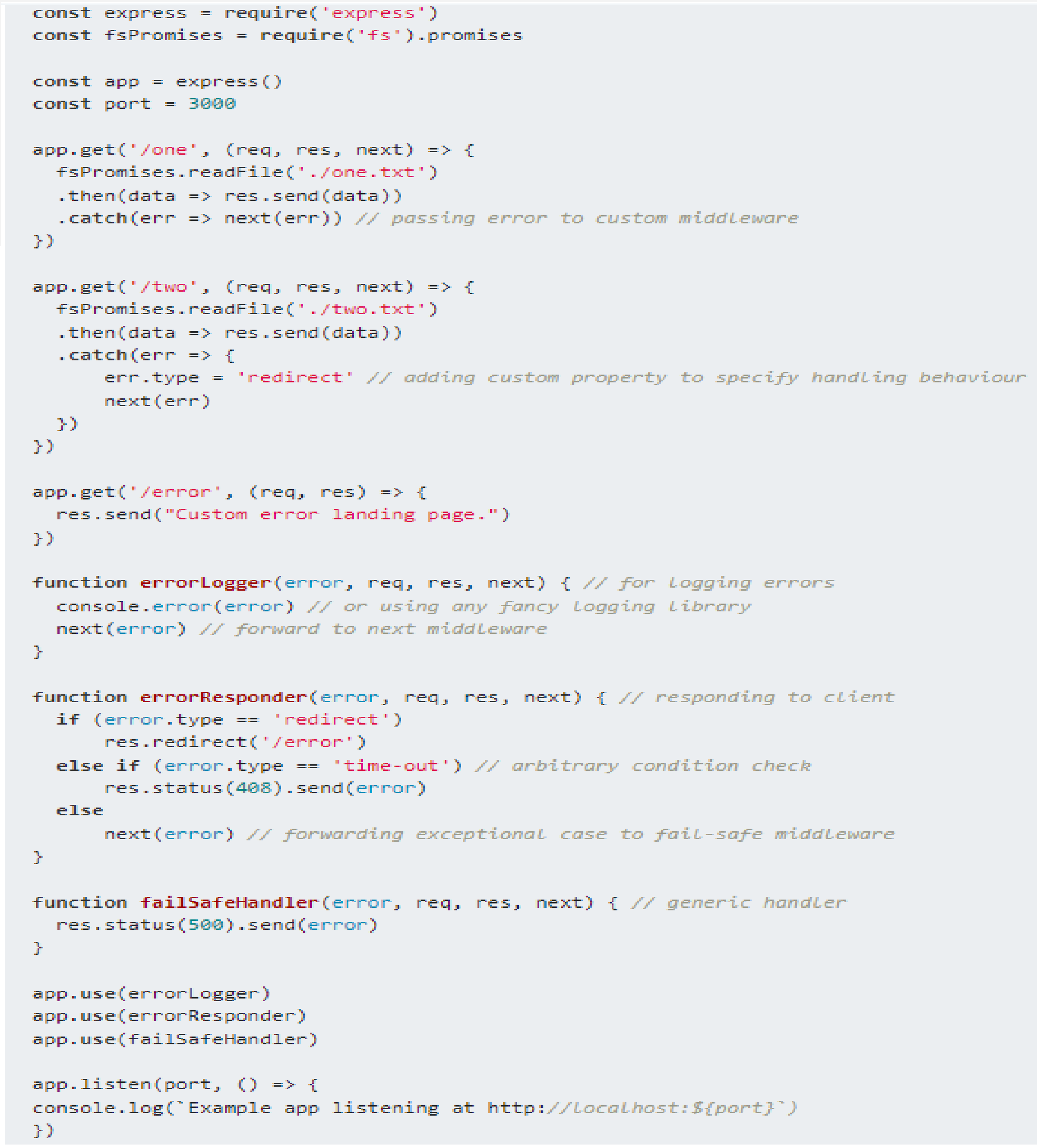
```
app.post("/data", function (req, res, next) {
    try {
        console.log("This middleware handles the data route");
        // ...
    } catch (err) {
        next(err);
    }
});
```

```
app.use(function (err, req, res, next) {
    console.error("Error found !");
    res.status(500).send("Something very wrong happened!");
});
```

Middleware

Writing error handlers:

- If an error occurs, all middleware that is meant to handle errors will be called in order until one of them does not call the next() function call



```
const express = require('express')
const fsPromises = require('fs').promises

const app = express()
const port = 3000

app.get('/one', (req, res, next) => {
  fsPromises.readFile('./one.txt')
  .then(data => res.send(data))
  .catch(err => next(err)) // passing error to custom middleware
})

app.get('/two', (req, res, next) => {
  fsPromises.readFile('./two.txt')
  .then(data => res.send(data))
  .catch(err => {
      err.type = 'redirect' // adding custom property to specify handling behaviour
      next(err)
  })
})

app.get('/error', (req, res) => {
  res.send("Custom error landing page.")
})

function errorLogger(error, req, res, next) { // for logging errors
  console.error(error) // or using any fancy logging library
  next(error) // forward to next middleware
}

function errorResponder(error, req, res, next) { // responding to client
  if (error.type == 'redirect')
      res.redirect('/error')
  else if (error.type == 'time-out') // arbitrary condition check
      res.status(408).send(error)
  else
      next(error) // forwarding exceptional case to fail-safe middleware
}

function failSafeHandler(error, req, res, next) { // generic handler
  res.status(500).send(error)
}

app.use(errorLogger)
app.use(errorResponder)
app.use(failSafeHandler)

app.listen(port, () => {
console.log(`Example app listening at http://localhost:${port}`)
})
```

Middleware

Built-in middleware

– `express.static()`: serves static assets (HTML files, images,…)

– `express.json()`: parses incoming requests with JSON payloads (Express >= 4.16.0)

– `express.urlencoded()`: parses incoming requests with URL-encoded payloads (Express >= 4.16.0)

Middleware

(Some) External middleware

- `morgan`: HTTP request logger middleware for node.js
- cors: middleware that can be used to enable CORS with various options
- `cookie-parser`: parses Cookie header and populates `req.cookies` with an object keyed by the cookie
- `multer`: middleware for handling `multipart/form-data` (file uploads)

Middleware

(Some) External middleware

```
npm install morgan
```

```
const morgan = require("morgan");
app.use(morgan("dev"));
```

package.json

Installing dependencies

In our examples, we had to install the Express npm packages

```
$ npm install express
```

These get written to the `node_modules` directory

Uploading server code

When you upload NodeJS code to a GitHub repository (or any code repository), you should **not** upload the `node_modules` directory:

- You shouldn't be modifying code in the `node_modules` directory, so there's no reason to have it under version control
- This will also increase your repo size significantly

Q: But if you don't upload the node_modules directory to your code repository, how will anyone know what libraries they need to install?

Managing dependencies

If we don't include the `node_modules` directory in our repository, we need to somehow tell other people what npm modules they need to install

npm provides a mechanism for this: [`package.json`]

package.json

You can put a file named `package.json` in the root directory of your NodeJS project to specify metadata about your project

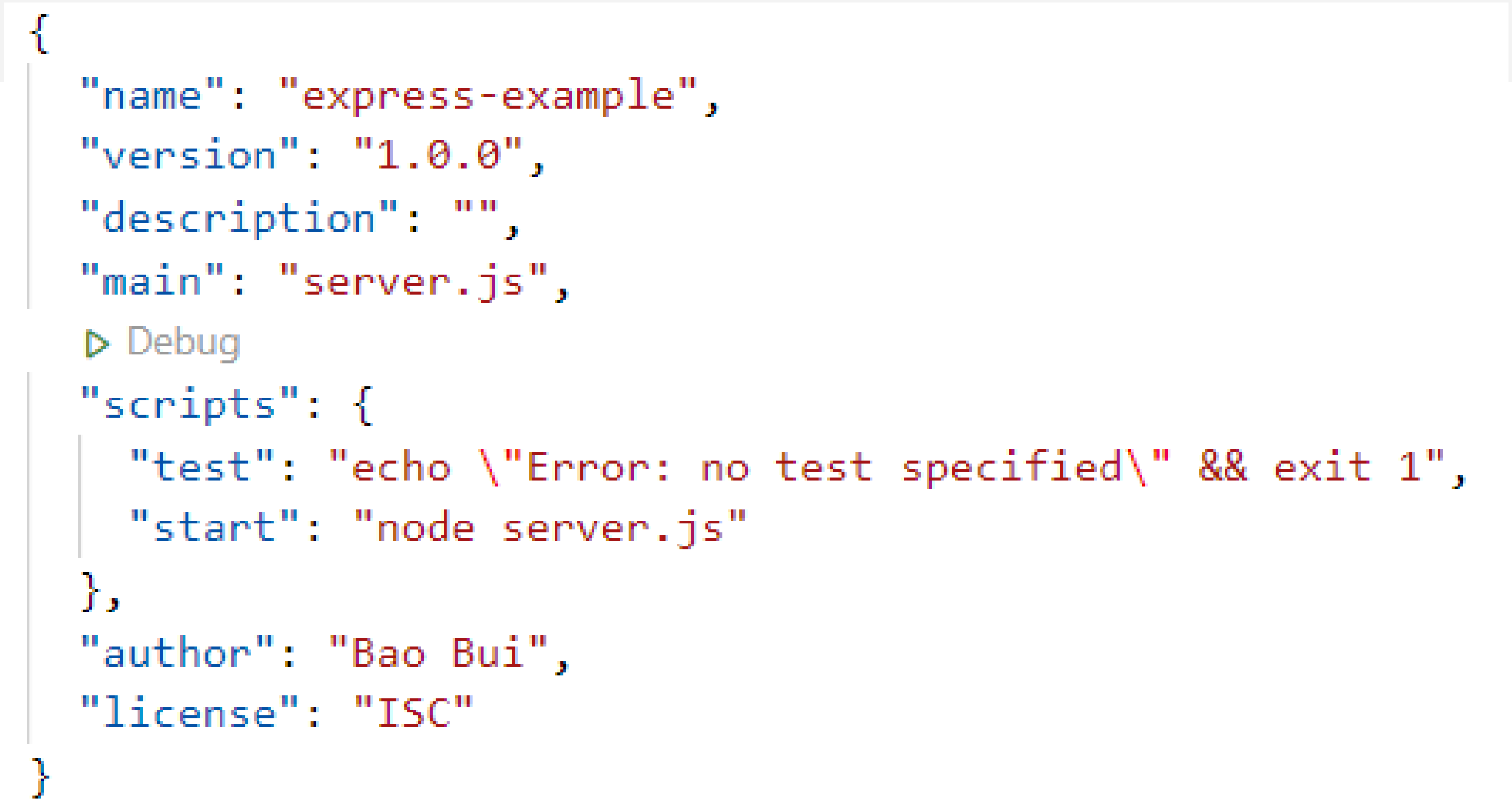
Create a `package.json` file using the following command:

```
$ npm init
```

This will ask you a series of questions then generate a `package.json` file based on your answers

- Add `-y` option to get a package.json file with default values

Auto-generated package.json



```
{
  "name": "express-example",
  "version": "1.0.0",
  "description": "",
  "main": "server.js",
  ▷ Debug
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1",
    "start": "node server.js"
  },
  "author": "Bao Bui",
  "license": "ISC"
}
```

Saving deps to package.json

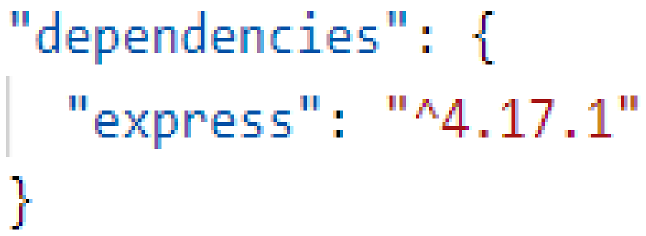
Now when you install packages:

```
$ npm install express
```

Or

```
$ npm i express
```

An entry for this library is added in package.json



```
"dependencies": {
  "express": "^4.17.1"
}
```

Saving deps to package.json

If you remove the node_modules directory:

```
$ rm -rf node_modules
```

You can install your project dependencies again via:

```
$ npm install
```

- This also allows people who have downloaded your code from GitHub to install all your dependencies with one command instead of having to install all dependencies individually

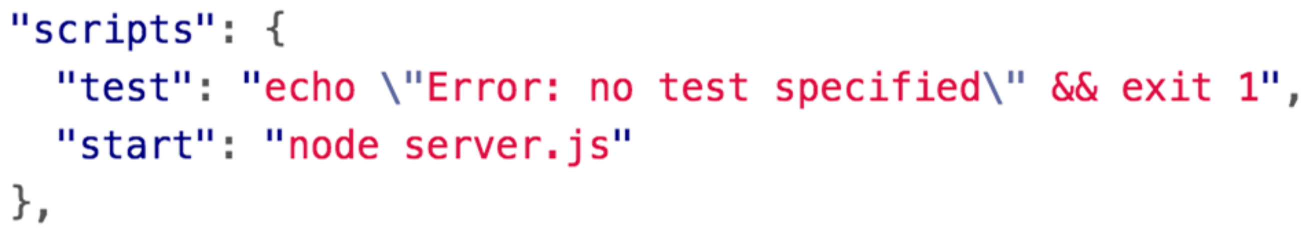
package-lock.json

package-lock.json is auto generated for any operations where npm modifies either the *node_modules* tree, or *package.json*

It describes the exact tree that was generated

npm scripts

Your package.json file also defines scripts:



```
"scripts": {
  "test": "echo \"Error: no test specified\" && exit 1",
  "start": "node server.js"
},
```

You can run these scripts using `$ npm [run]` ***scriptName***

E.g. the following command runs "node server.js"

```
$ npm start
```

nodemon

Automatically restart the node application when file changes

```
$ npm i -D nodemon
```

`-D` option: package will appear in *devDependencies* section

dependencies: packages required to run
devDependencies: only for development

`npm install`: install packages listed in both sections
`npm install --production`: only packages in *dependencies* are installed

nodemon

Automatically restart the node application when file changes

In package.json, use nodemon to start the server:

```
"scripts": {
    "start": "nodemon server.js"
}
```

Then run `$ npm start`

npx: an npm package runner

npx makes it easy to use CLI tools and other executables hosted on the registry

Using locally-installed tools without npm run-script:

```
$ npm i –D cowsay
$ npx cowsay hello!
```

Executing one-off commands:

```
$ npx cowsay hello!
```

Node.js Event Loop

Event loop

The **Event loop** is only a watchdog that ensures that the Call Stack and Callback Queue are in constant communication.

- It first determines whether the call stack is free, and then informs the user of the callback queue.
- The call stack is a LIFO queue (Last In, First Out)

A simple event loop explanation: https://nodejs.dev/en/learn/the-nodejs-event-loop

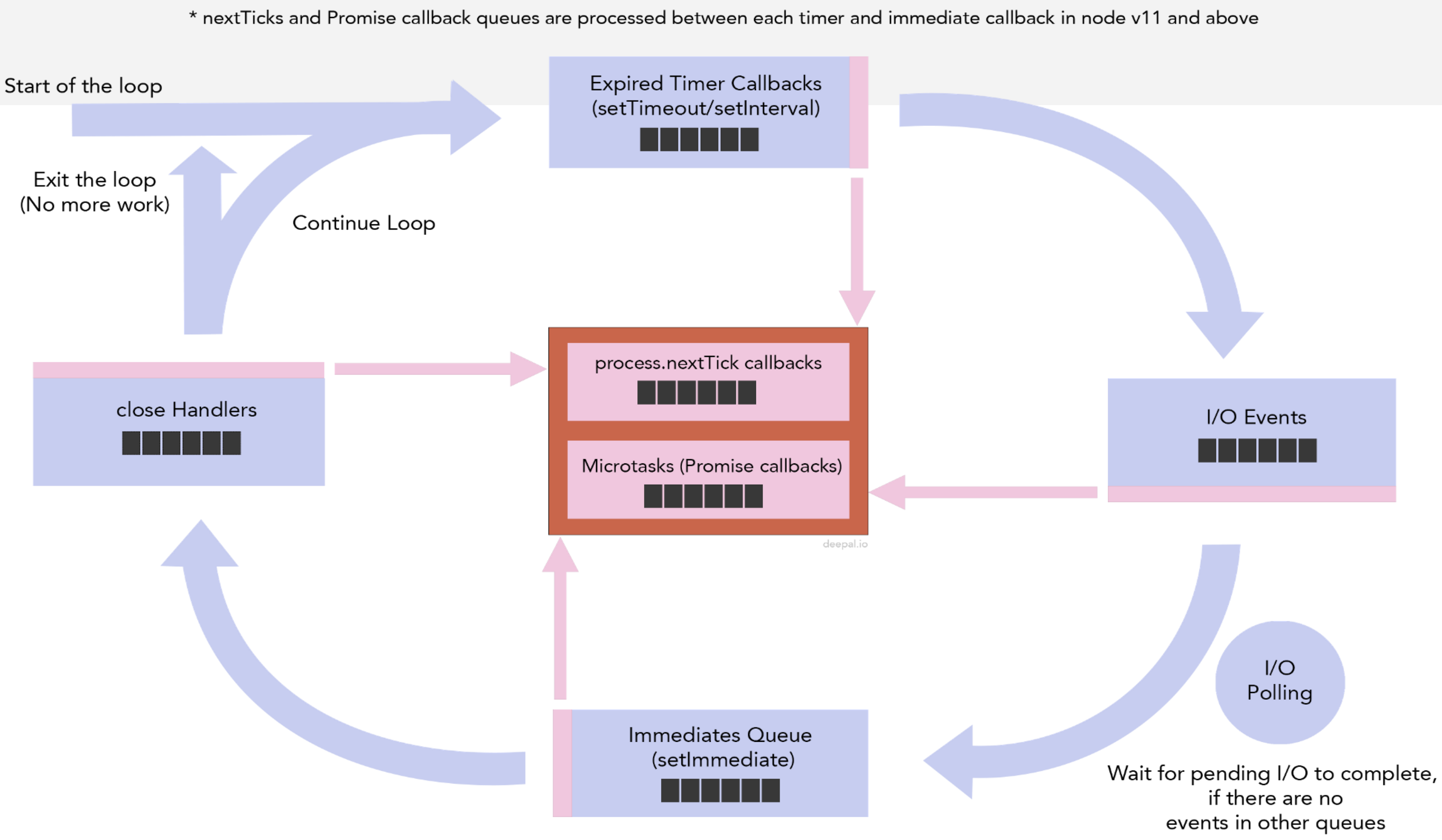
Event loop

There are two types of threads in Nodejs:

- One Event Loop (aka the main loop, main thread, event thread, etc.)
  - Responsible for callbacks and non-blocking I/O (i.e., network I/O)
  - *All incoming requests and outgoing responses pass through the Event Loop*

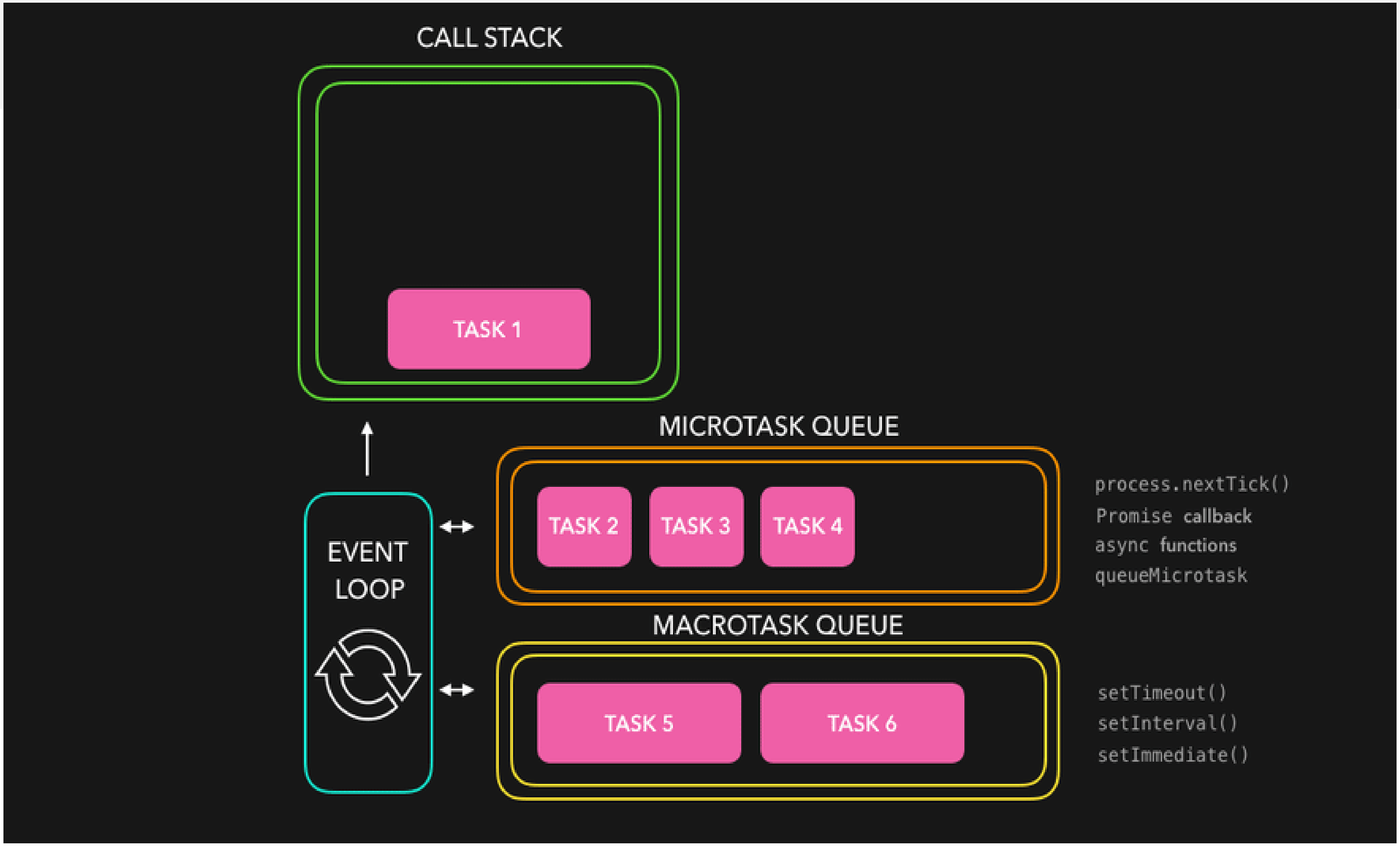
- A pool of k (k=4 by default) Workers in a Worker Pool (aka the threadpool)
  - Handle I/O intensive tasks (dns, file APIs) or CPU-intensive ones (crypto, zlib APIs)

Event loop


* nextTicks and Promise callback queues are processed between each timer and immediate callback in node v11 and above
Start of the loop
Expired Timer Callbacks (setTimeout/setInterval)
Exit the loop (No more work)
Continue Loop
close Handlers
process.nextTick callbacks
Microtasks (Promise callbacks)
I/O Events
deepal.io
I/O Polling
Immediates Queue (setImmediate)
Wait for pending I/O to complete, if there are no events in other queues

https://blog.insiderattack.net/event-loop-and-the-big-picture-nodejs-event-loop-part-1-1cb67a182810

Event loop



https://techva.me/callback-functions-promises-nodejs/

`setTimeout`

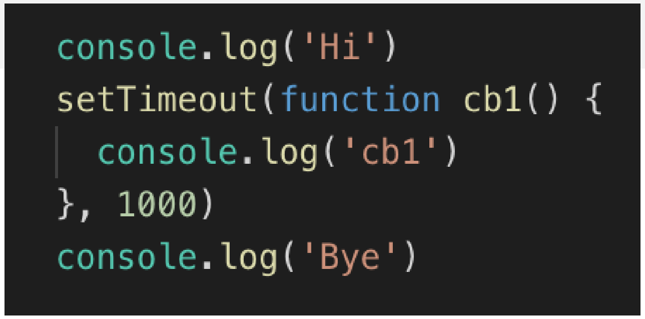
To help us understand the event loop better, let's learn about a new command, `setTimeout`:

`setTimeout(`***function***`, `***delay***`);`

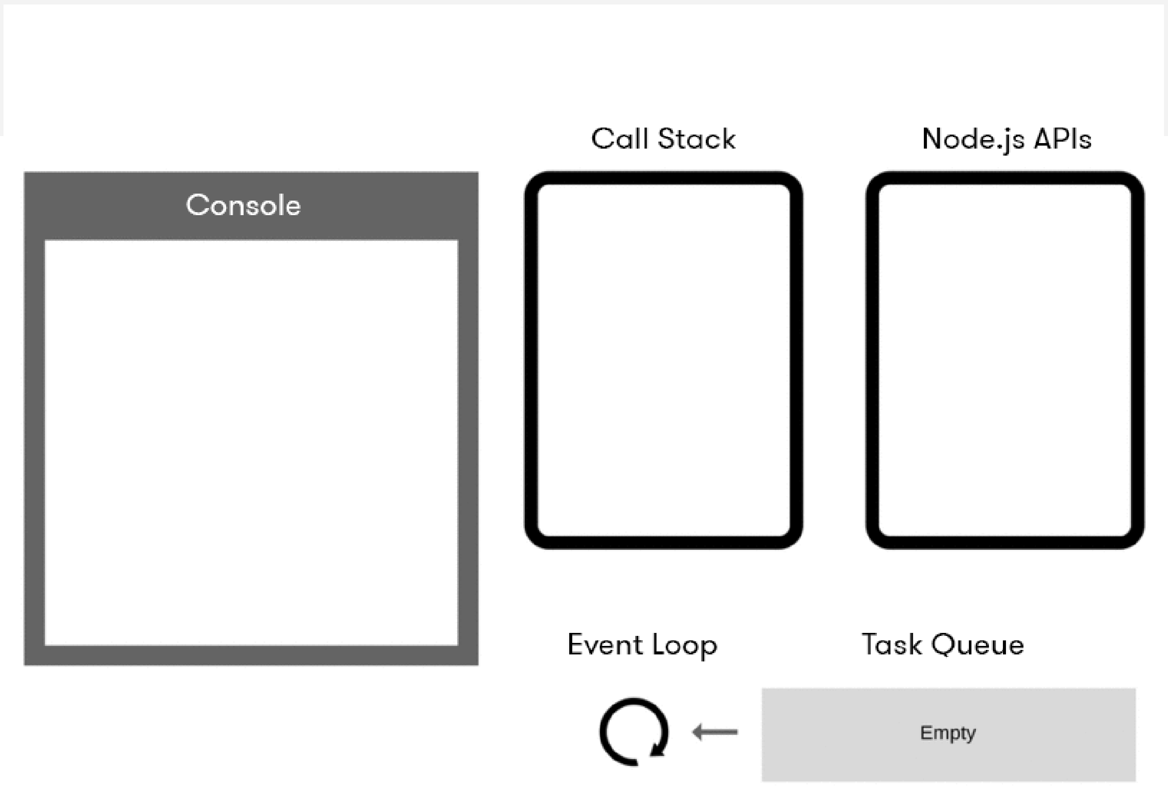
- ***function*** will fire after ***delay*** milliseconds

setImmediate(), nextTick() and setTimeout(fn,0): https://www.voidcanvas.com/setimmediate-vs-nexttick-vs-settimeout/

JS execution

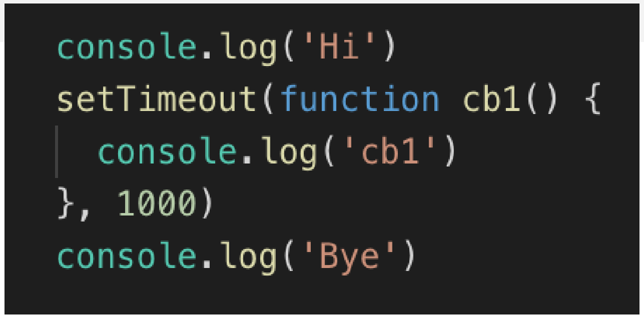


```
console.log('Hi')
setTimeout(function cb1() {
  console.log('cb1')
}, 1000)
console.log('Bye')
```

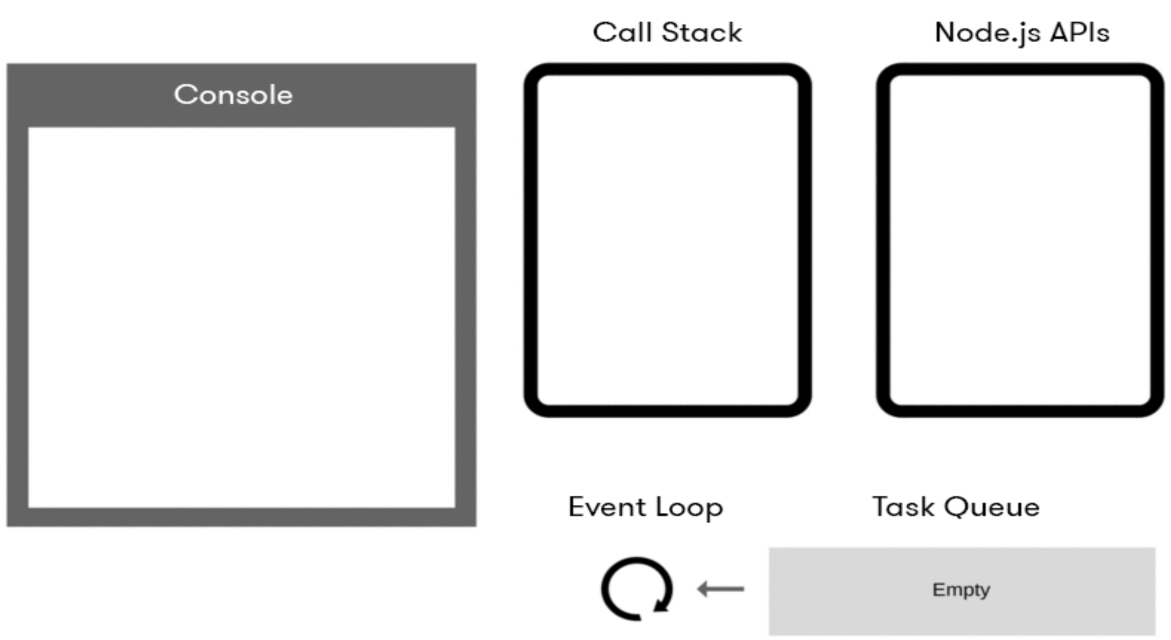


Call stack: JavaScript runtime call stack. Executes the JavaScript commands, functions

JS execution

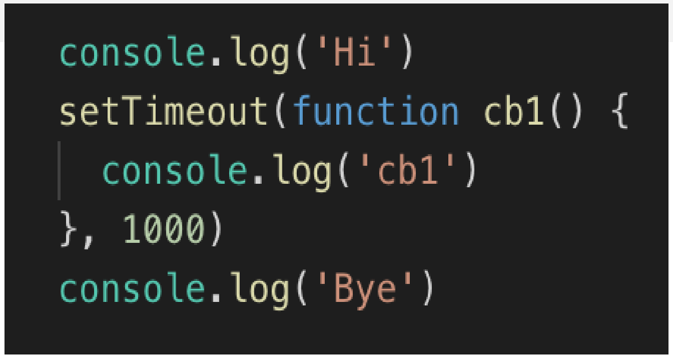


```
console.log('Hi')
setTimeout(function cb1() {
  console.log('cb1')
}, 1000)
console.log('Bye')
```

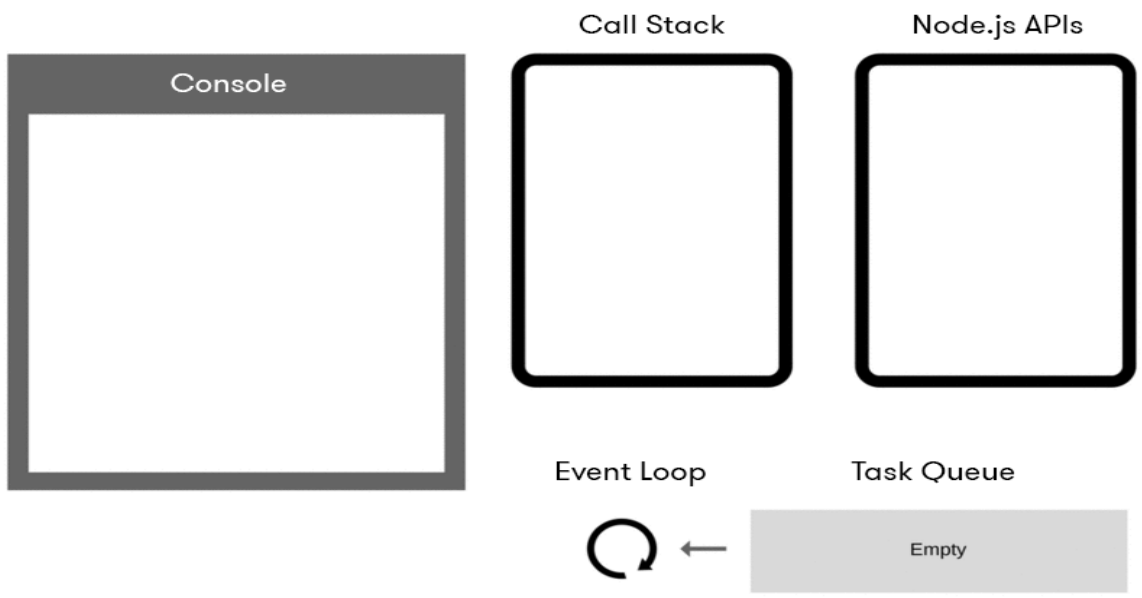


Task Queue: When Node.js APIs notice a callback from something like setTimeout should be fired, it creates a Task and enqueues it in the Task Queue

JS execution



```
console.log('Hi')
setTimeout(function cb1() {
  console.log('cb1')
}, 1000)
console.log('Bye')
```

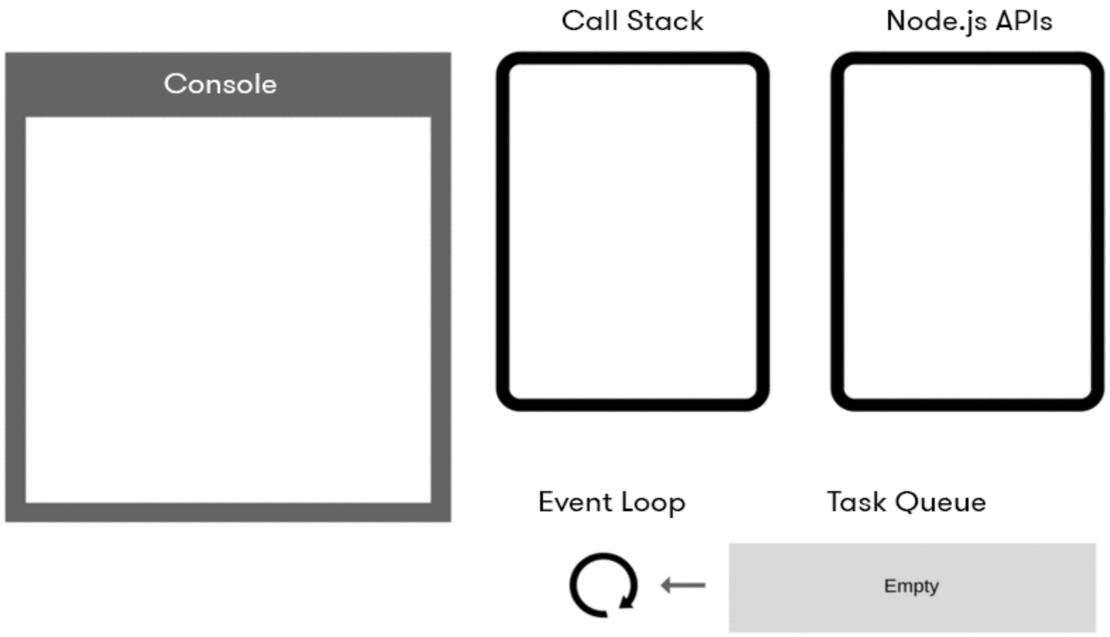


Event loop: Processes the task queues

- When the call stack is empty, the event loop pulls the next task from the task queues and puts it on the call stack

JS execution

```
console.log('Hi')
setTimeout(function cb1() {
  console.log('cb1')
}, 1000)
console.log('Bye')
```



https://symphony.is/blog/secret-life-event-loop-meetup-overview