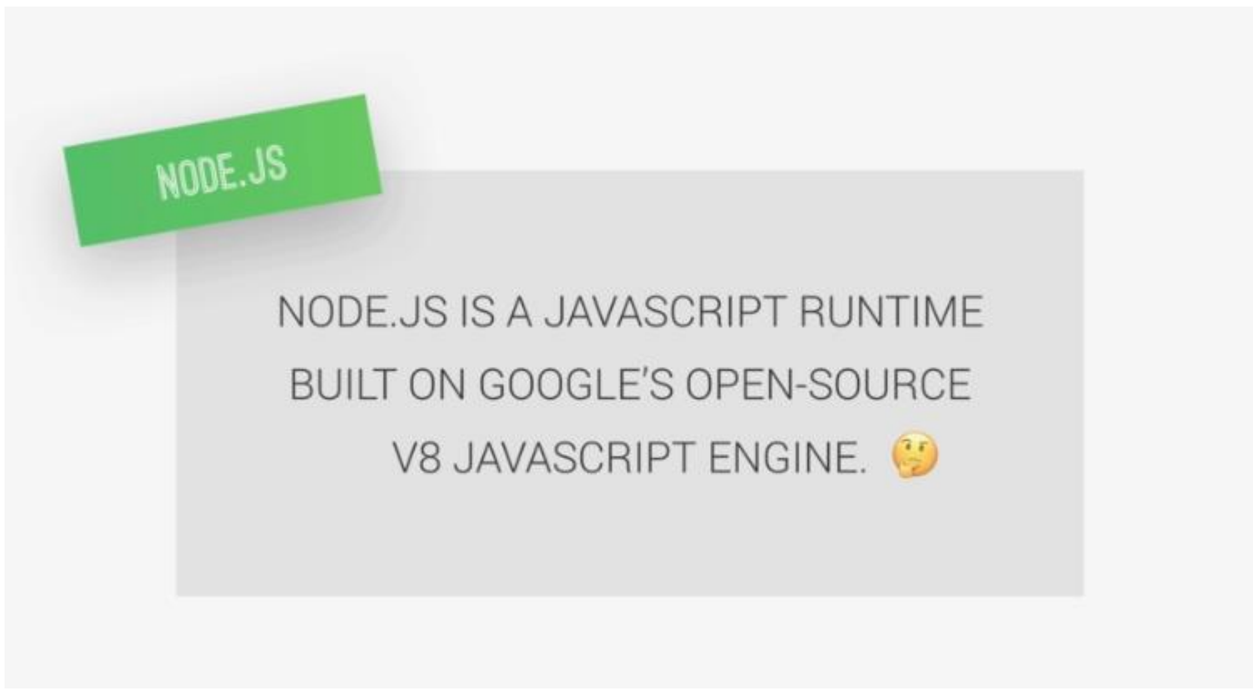
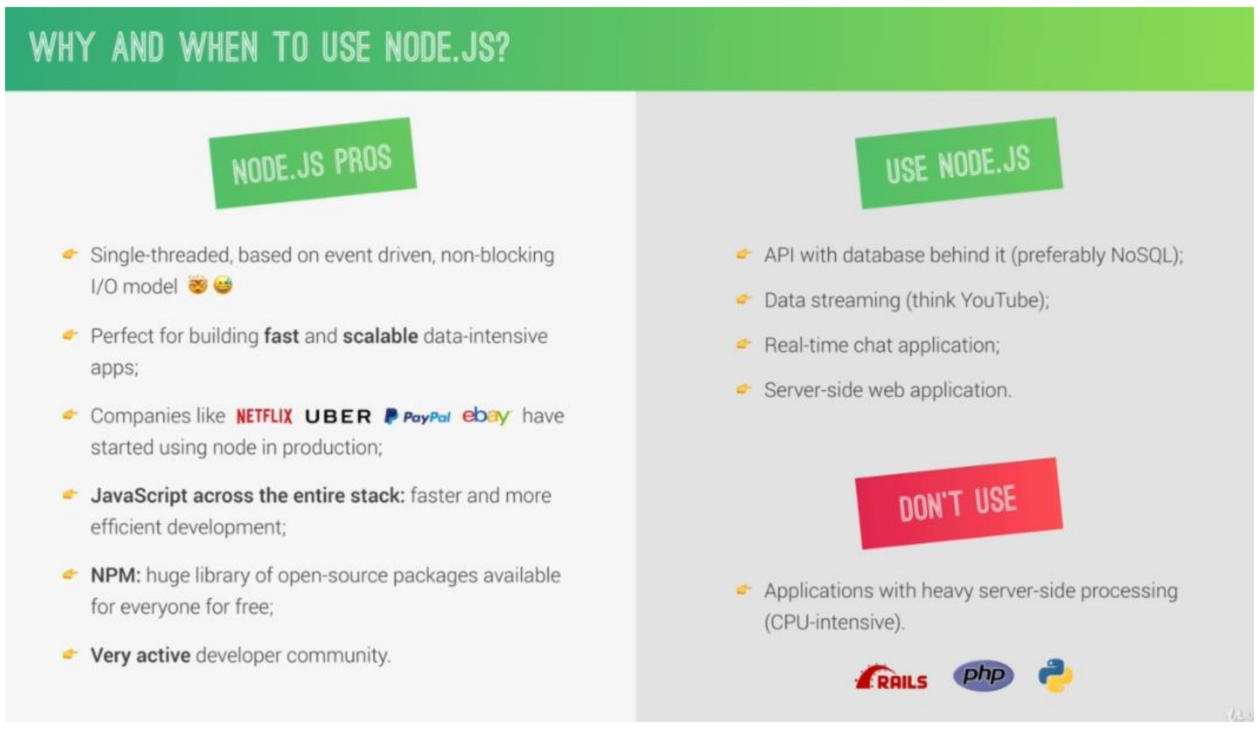


NODE.JS

NODE.JS IS A JAVASCRIPT RUNTIME BUILT ON GOOGLE'S OPEN-SOURCE V8 JAVASCRIPT ENGINE. 🤔



WHY AND WHEN TO USE NODE.JS?

NODE.JS PROS

- Single-threaded, based on event driven, non-blocking I/O model 🤯 😅
- Perfect for building **fast** and **scalable** data-intensive apps;
- Companies like NETFLIX UBER PayPal ebay have started using node in production;
- **JavaScript across the entire stack:** faster and more efficient development;
- **NPM:** huge library of open-source packages available for everyone for free;
- **Very active** developer community.

USE NODE.JS

- API with database behind it (preferably NoSQL);
- Data streaming (think YouTube);
- Real-time chat application;
- Server-side web application.

DON'T USE

- Applications with heavy server-side processing (CPU-intensive).

A scalable web application is **a website that is able to handle an increase in users and load.**

NodeJs fast karon single-threaded, non- blocking i/o model etc.

Write code in Terminal:

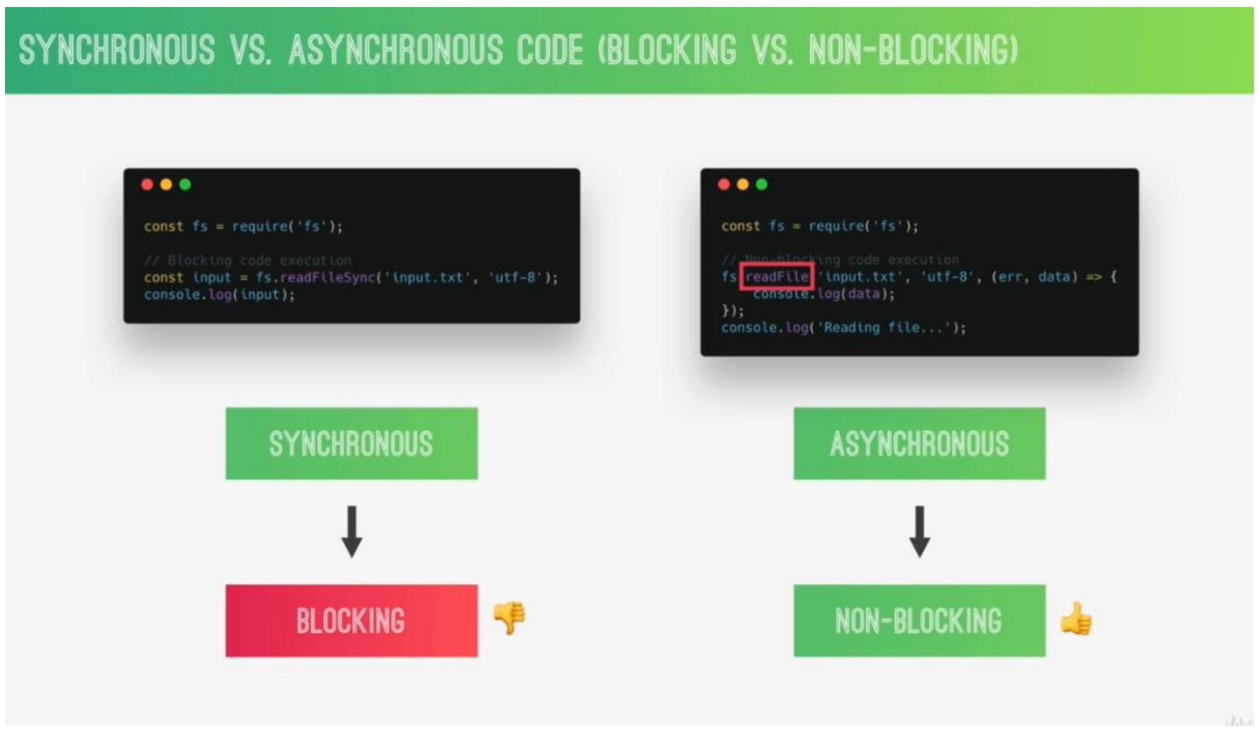
```
node
> const name = 'jonas'
undefined
> name
'jonas'
```

For see all stuff in global :

In Terminal type:

```
Node
Tab tab
```

```
>3*4
12
> _ + 4
16
```



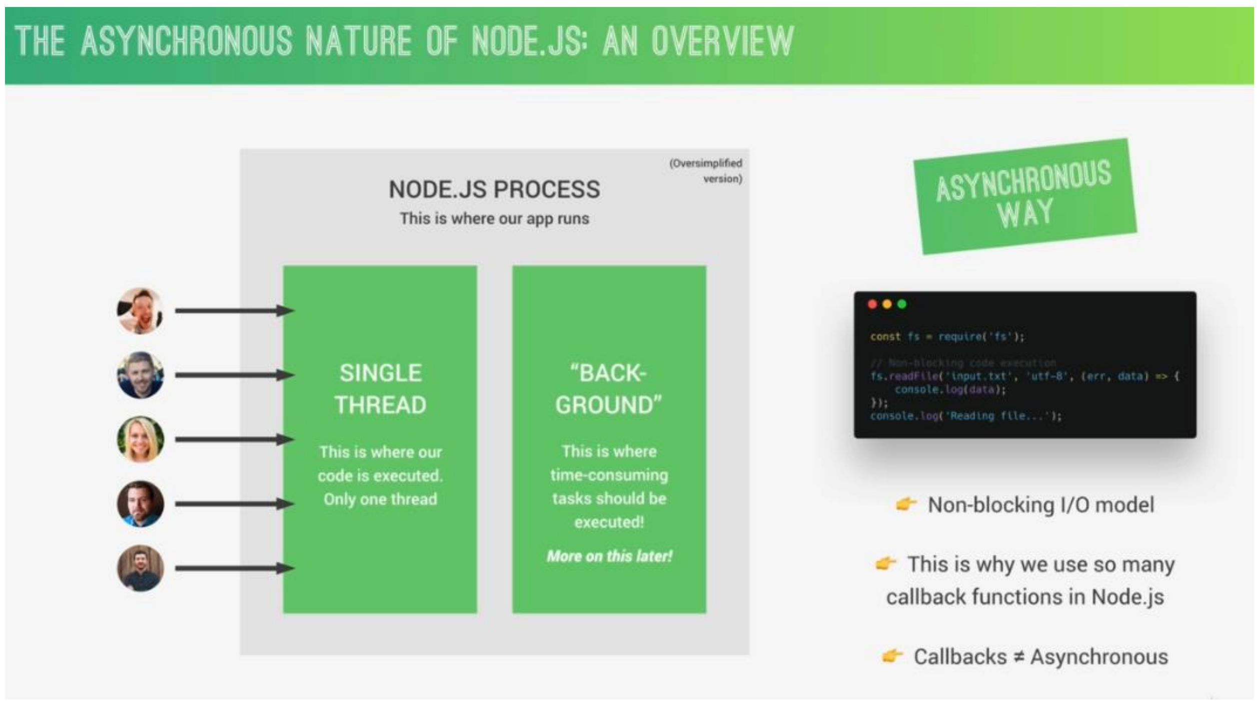
SYNCHRONOUS VS. ASYNCHRONOUS CODE (BLOCKING VS. NON-BLOCKING)

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

// Blocking code execution
const input = fs.readFileSync('input.txt', 'utf-8');
console.log(input);
```

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

// Non-blocking code execution
fs.readFile('input.txt', 'utf-8', (err, data) => {
    console.log(data);
});
console.log('Reading file...');
```

SYNCHRONOUS → BLOCKING 👎

ASYNCHRONOUS → NON-BLOCKING 👍



THE ASYNCHRONOUS NATURE OF NODE.JS: AN OVERVIEW

(Oversimplified version)
NODE.JS PROCESS
This is where our app runs
SINGLE THREAD
This is where our code is executed. Only one thread
"BACK-GROUND"
This is where time-consuming tasks should be executed!
More on this later!

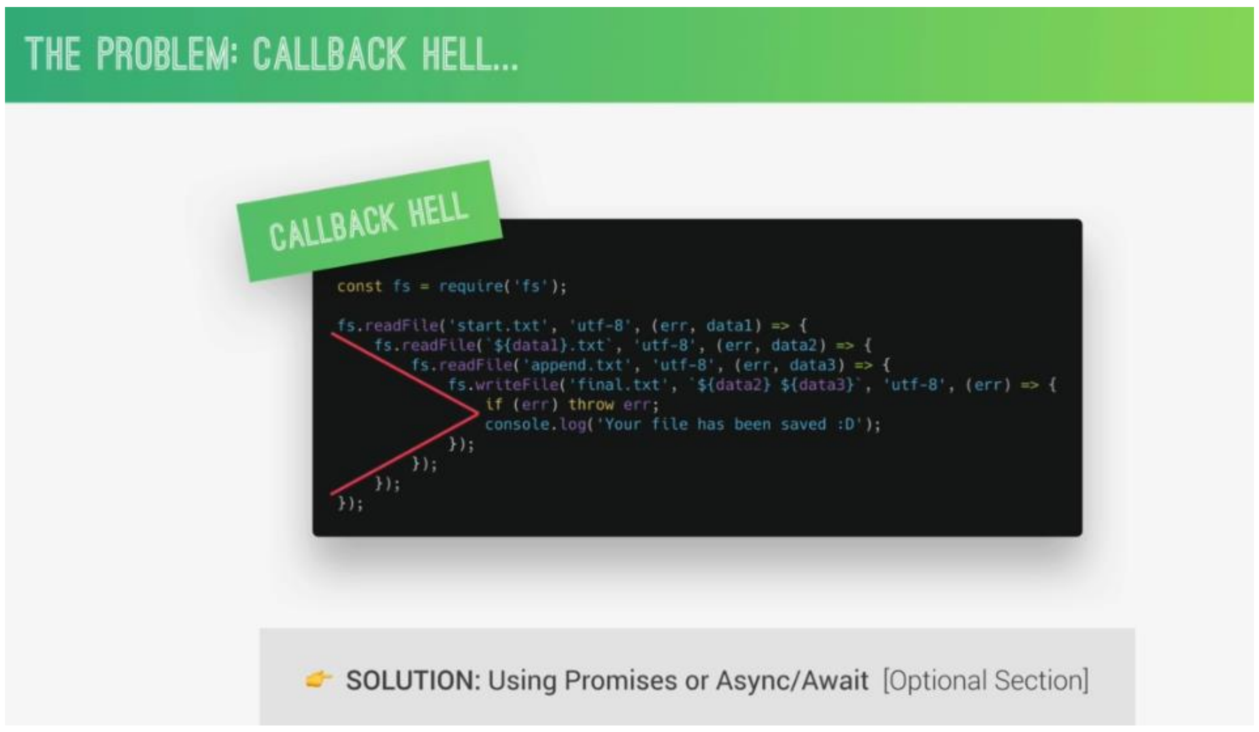
ASYNCHRONOUS WAY

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

// Non-blocking code execution
fs.readFile('input.txt', 'utf-8', (err, data) => {
    console.log(data);
});
console.log('Reading file...');
```

👉 Non-blocking I/O model

👉 This is why we use so many callback functions in Node.js

👉 Callbacks ≠ Asynchronous



THE PROBLEM: CALLBACK HELL...

CALLBACK HELL

```javascript
const fs = require('fs');

fs.readFile('start.txt', 'utf-8', (err, data1) => {
    fs.readFile(`${data1}.txt`, 'utf-8', (err, data2) => {
        fs.readFile('append.txt', 'utf-8', (err, data3) => {
            fs.writeFile('final.txt', `${data2} ${data3}`, 'utf-8', (err) => {
                if (err) throw err;
                console.log('Your file has been saved :D');
            });
        });
    });
});
```

👉 **SOLUTION:** Using Promises or Async/Await [Optional Section]

In terminal:

Ctrl+c

.exit

```javascript
const fs = require('fs');
const textIn = fs.readFileSync('./txt/input.txt', 'utf-8');
console.log(textIn);

const myValues= `Hi`
const myWriteFile = fs.writeFileSync('./txt/out.txt', myValues)
console.log('File Written!');


///////////////////////////////
const http = require('http');
const url = require('url');
//server
const server = http.createServer((req, res) => {
    const pathName = req.url;

    if (pathName === '/' || pathName === '/overview') {
        res.end('This is the OverView');
    }
    else if (pathName === '/product'){
        res.end('This is the Product');
    }
    else {
        res.end('Page not found');
    }
})

server.listen(8000, '127.0.0.1', () => {
    console.log('Listening to requests on port 8000');
});
```

Why do I need to use JSON Stringify?

The JSON. stringify() method is **used to create a JSON string out of it**.
The `JSON.parse()` method parses a JSON string

JSON parsing is the process of converting a JSON object in text format

```
const http = require('http');
const url = require('url');
//server
const server = http.createServer((req, res) => {
    const pathName = req.url;

    if (pathName === '/' || pathName === '/overview') {
        res.end('This is the OverView');
    }
    else if (pathName === '/product'){
        res.end('This is the Product');
    }
    else if (pathName === '/api') {
        // fs.readFile(`${__dirname}/dev-data/data.json`, 'utf-8', (err, data) => {
        fs.readFile(`./dev-data/data.json`, 'utf-8', (err, data) => {
            const productData = JSON.parse(data);
            res.writeHead(200, { 'content-type': 'application/json' });
            // console.log(productData);
             res.end(data);
        })
        // res.end('API');
    }
    else {
        res.end('Page not found');
    }
})

server.listen(8000, '127.0.0.1', () => {
    console.log('Listening to requests on port 8000');
});
```

In Javascript, what is {% include file.js %}?

Likely used by a server side language or templating engine/pre-processor of some kind - it's not "normal" HTML/JS

*placeholder

Developer dependencies hole = --save-dev diya install korbo

Nodemon globally install korbo karon shob project a use korbo.

Version = 1.18.11

= huge chang/ new versione = fetures =error/bug fix old version hole dekhabey = Npm outdated

Old version update command = npm update package name

Package uninstall command = npm uninstall package name

Project share korle package.json & package-lock.json file shoho share korbo.

Vs code extension:

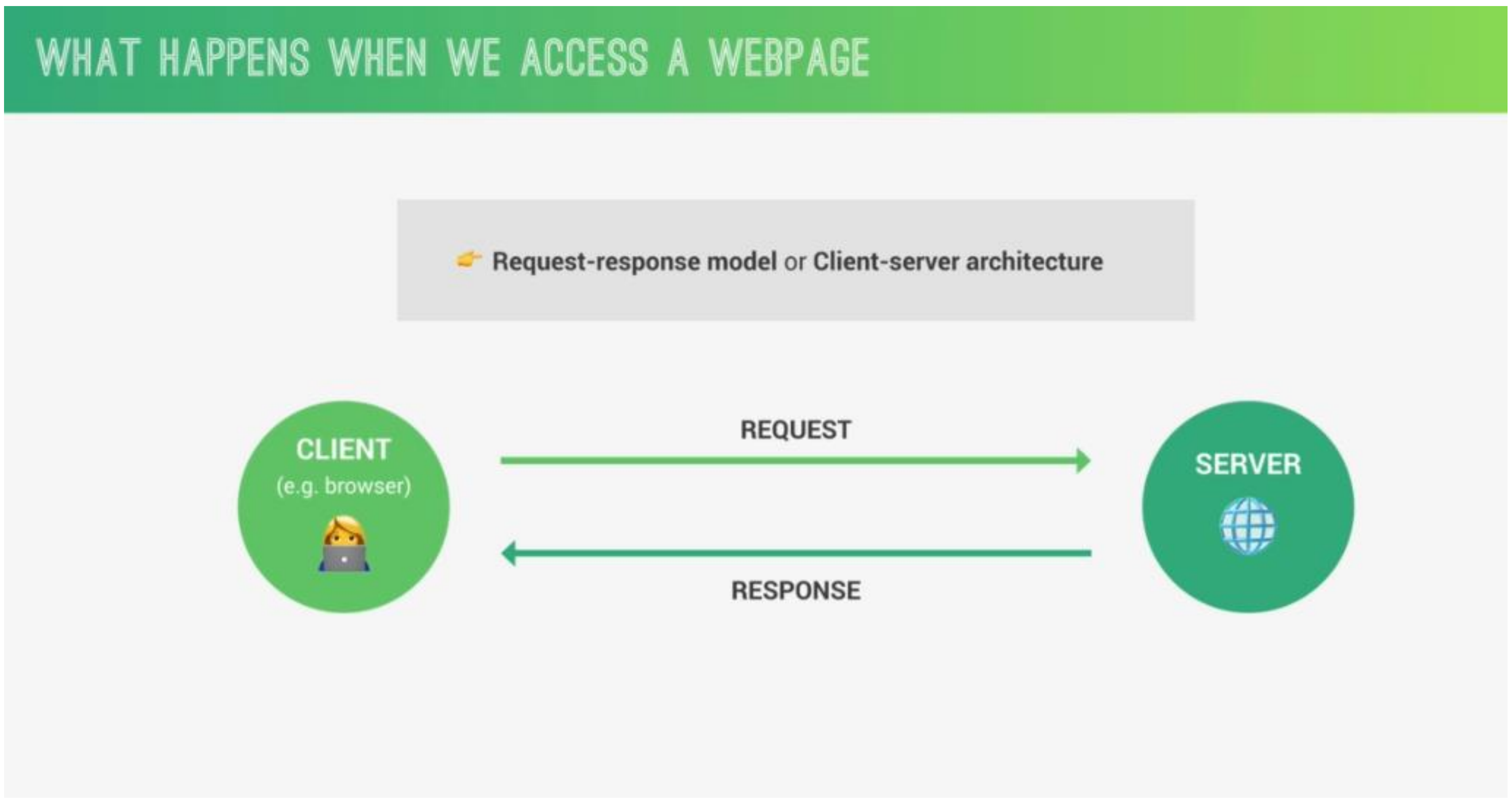
Image preview

Todo Highlight

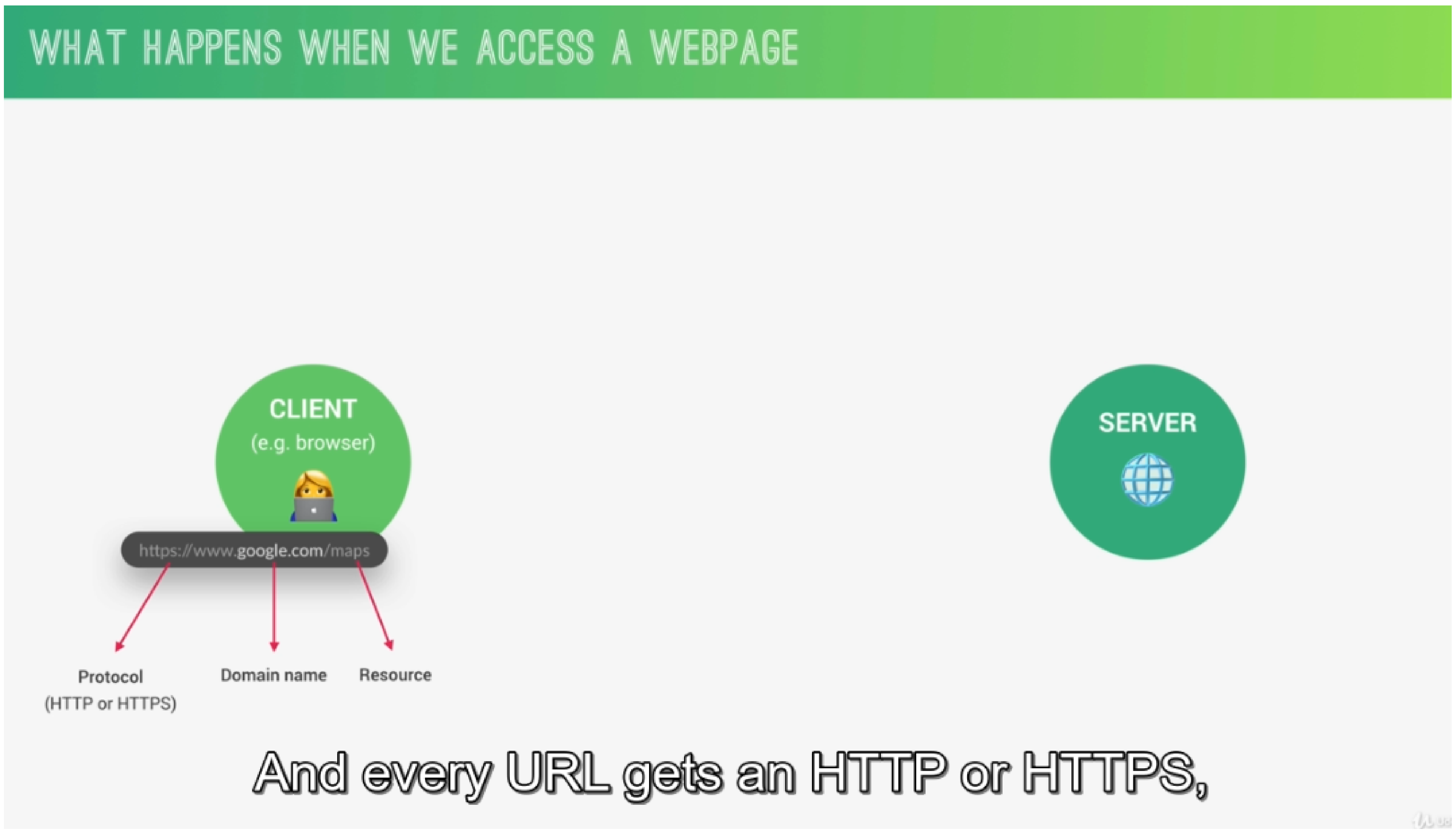
Prettier = search format & select format on save

In project create a file = .prettierrc

```
{
    "singleQuote": true,
    "printWidth": 80  //aita byDefault ase

}
```

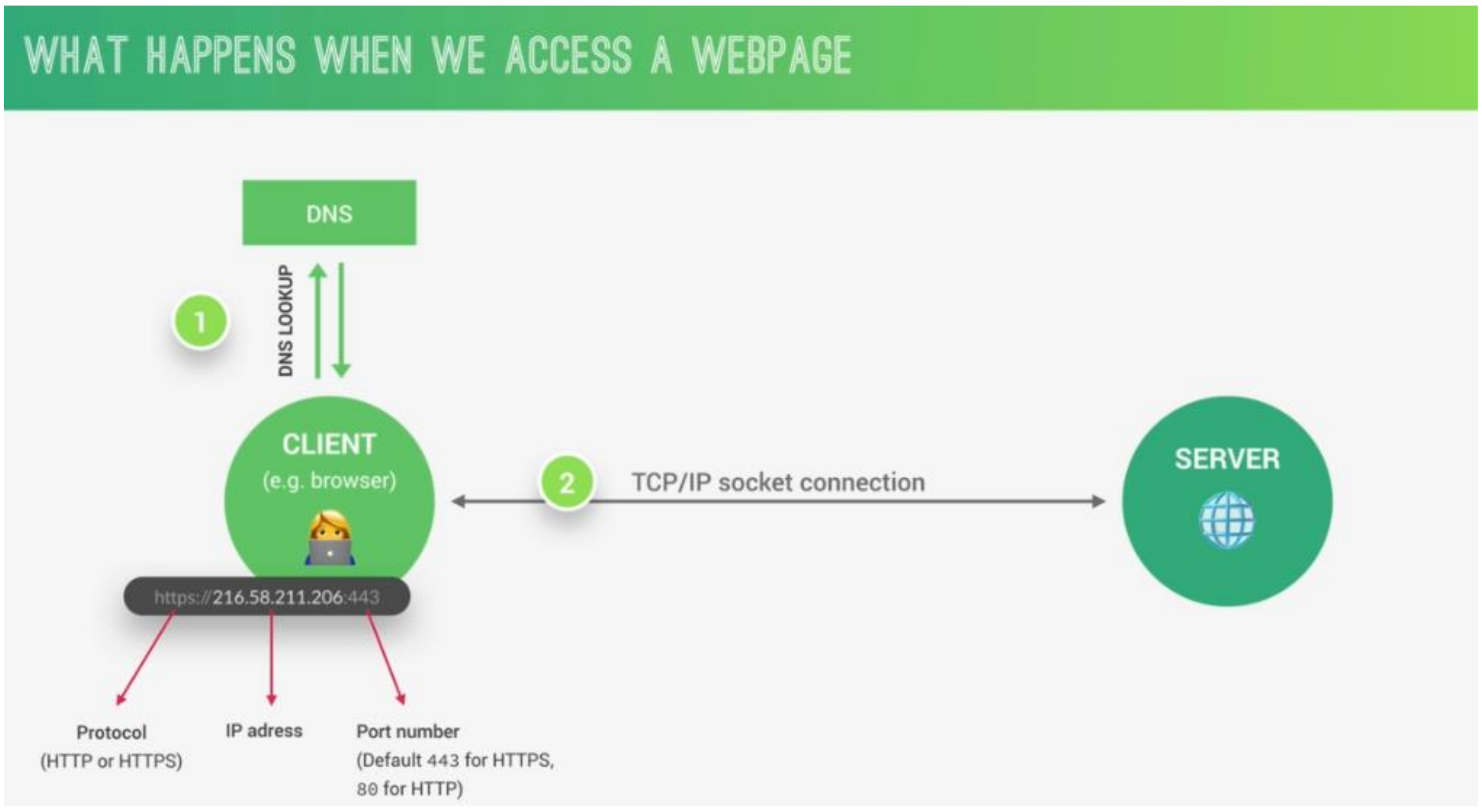

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE
Request-response model or Client-server architecture
CLIENT
(e.g. browser)
REQUEST
RESPONSE
SERVER

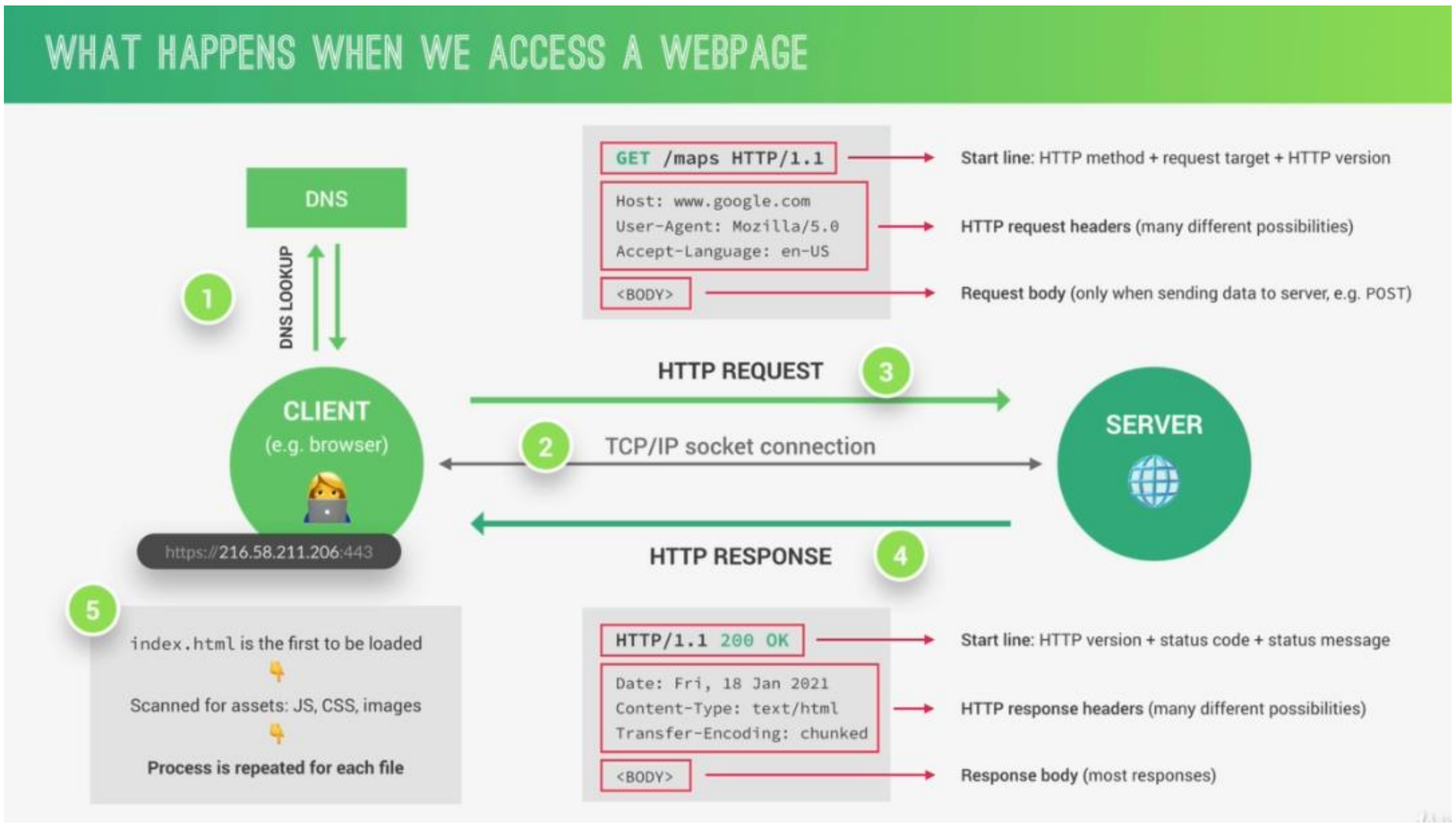
WHAT HAPPENS WHEN WE ACCESS A WEBPAGE



And every URL gets an HTTP or HTTPS,

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE




WHAT HAPPENS WHEN WE ACCESS A WEBPAGE
DNS
DNS LOOKUP
1
CLIENT
(e.g. browser)
https://216.58.211.206:443
2
TCP/IP socket connection
3
HTTP REQUEST
4
HTTP RESPONSE
SERVER
5
index.html is the first to be loaded
Scanned for assets: JS, CSS, images
Process is repeated for each file
GET /maps HTTP/1.1
Host: www.google.com
User-Agent: Mozilla/5.0
Accept-Language: en-US
<BODY>
Start line: HTTP method + request target + HTTP version
HTTP request headers (many different possibilities)
Request body (only when sending data to server, e.g. POST)
HTTP/1.1 200 OK
Date: Fri, 18 Jan 2021
Content-Type: text/html
Transfer-Encoding: chunked
<BODY>
Start line: HTTP version + status code + status message
HTTP response headers (many different possibilities)
Response body (most responses)

DNS = Domain name server   (IP er shate domain match kore thik ase ki na)

TCP = Transmission Control Protocol

http = http is a protocol

TCP is to break out the request and responses into thousands of small chunks called packets before they are set. Then once they get to their destination, it will reassemble all the packets into the original request or response. So that the message arrives at the destination as quick as possible, which would not be possible if we sent the website as one big chunk.

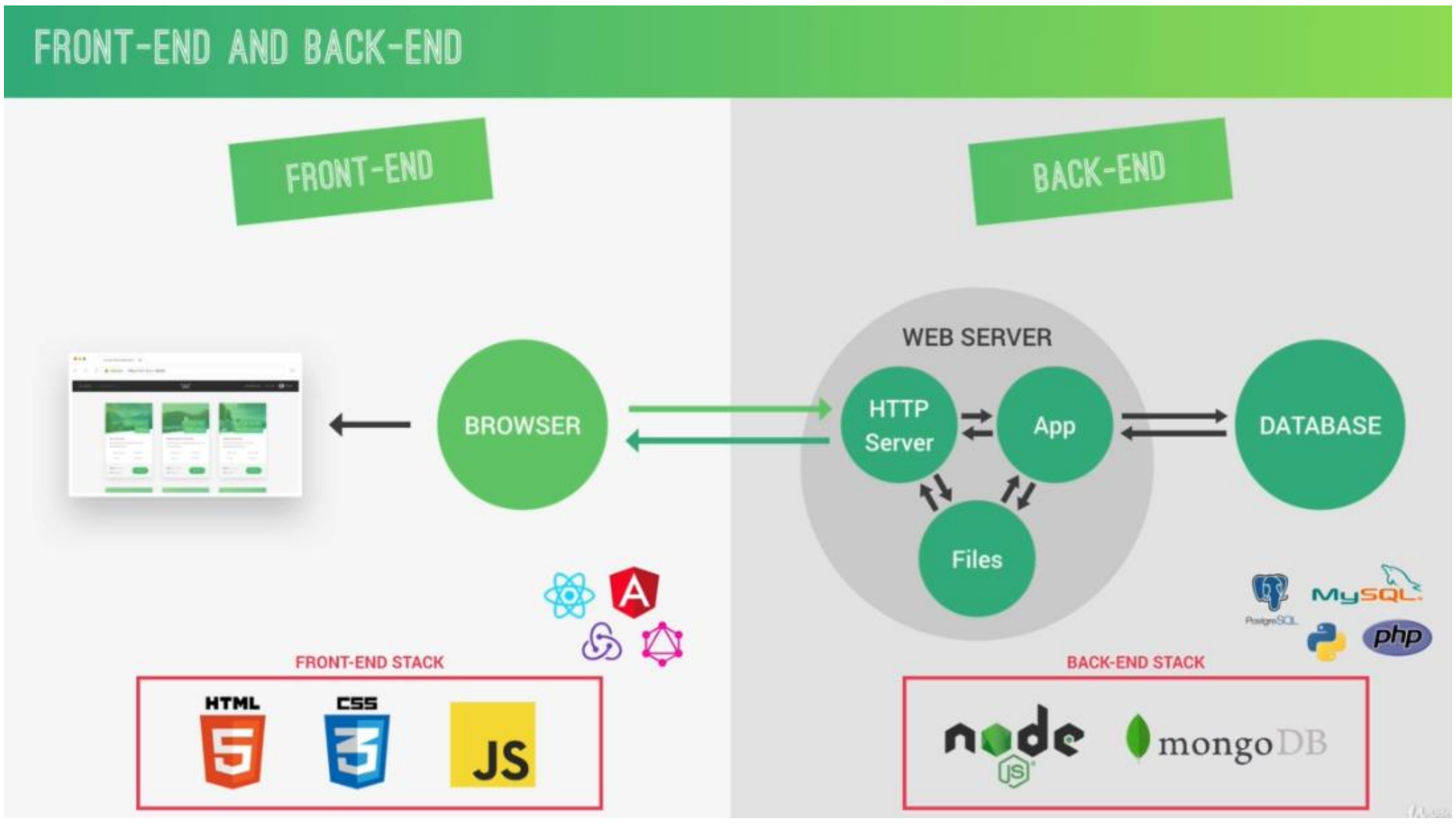
IP protocol = actually send and route all of these packet through the internet.

A communication protocol is a simply a system of rules that allows two or more parties to communicate.

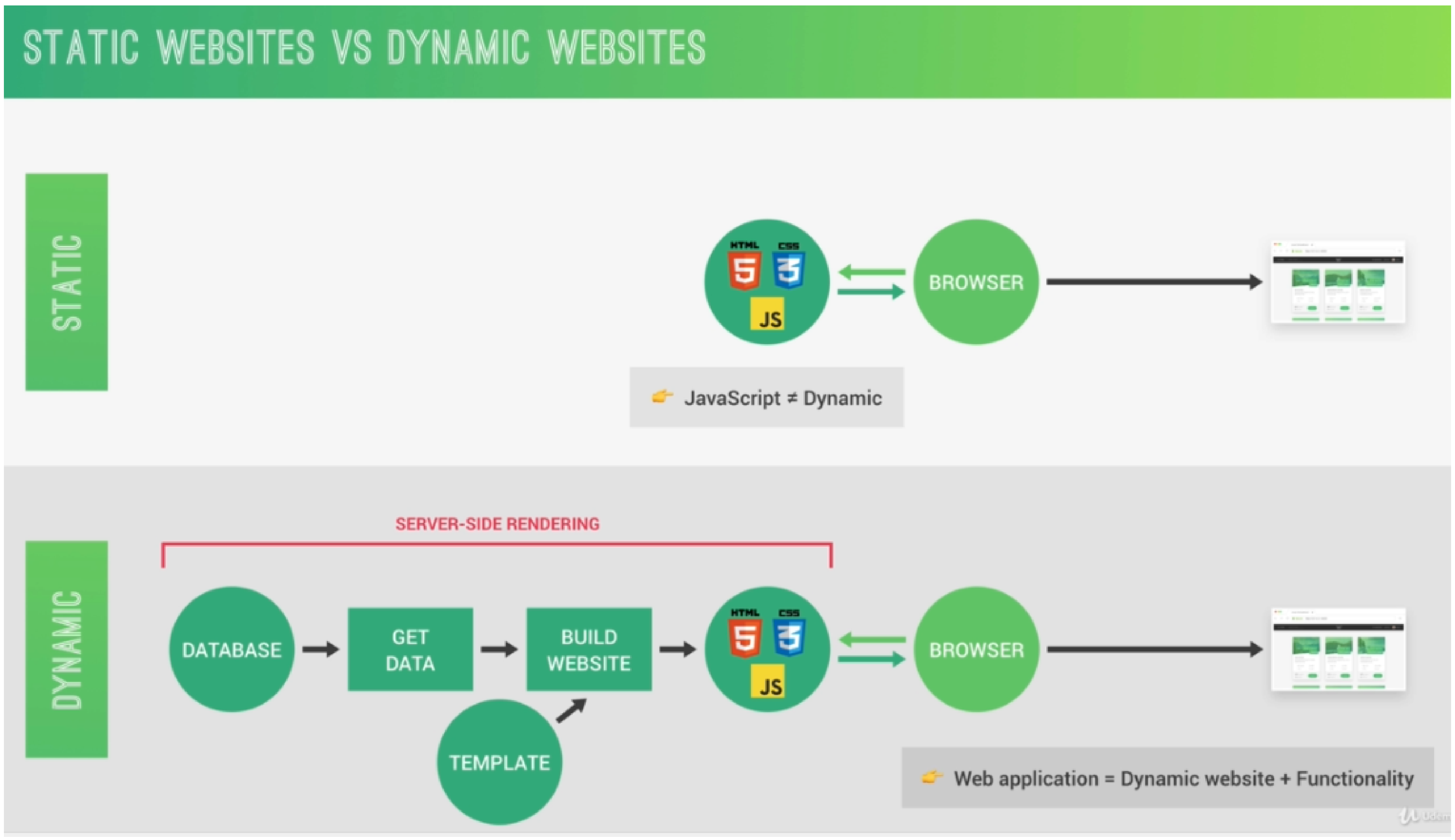
https = https is encrypted using TLS or SSL

Server = Amader computer and internet connect thakle e aita ekta basic server. Which first store a websites files like Html, css and images. And second its run an http server that is capable of understing URLs, request and also delivering response

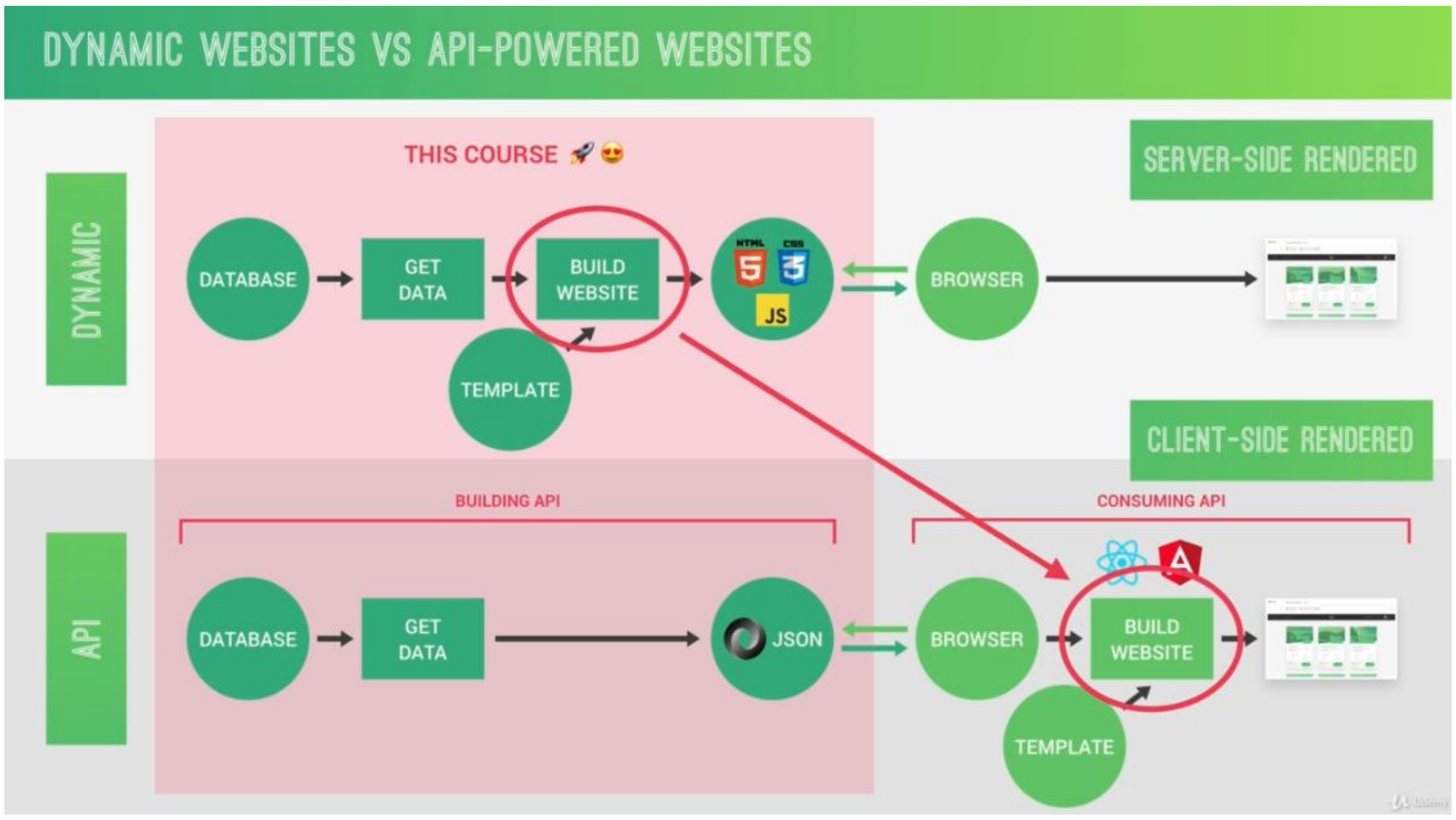
API = Application programming interface(its basically A piece of software that can be used by another piece of softare)



Web Application = Dynamic website same kotha. Traditionaly Static & dynamic But now Recently API base web Application very popular.



Bashir bug shomoi dynamic websites k bola hoi server-side rendered karon aita actually built on server. And API-powered websites k pry shomoi bola hoi client-side rendered



Node Nije ekta program

V8 = C++ diya toiri

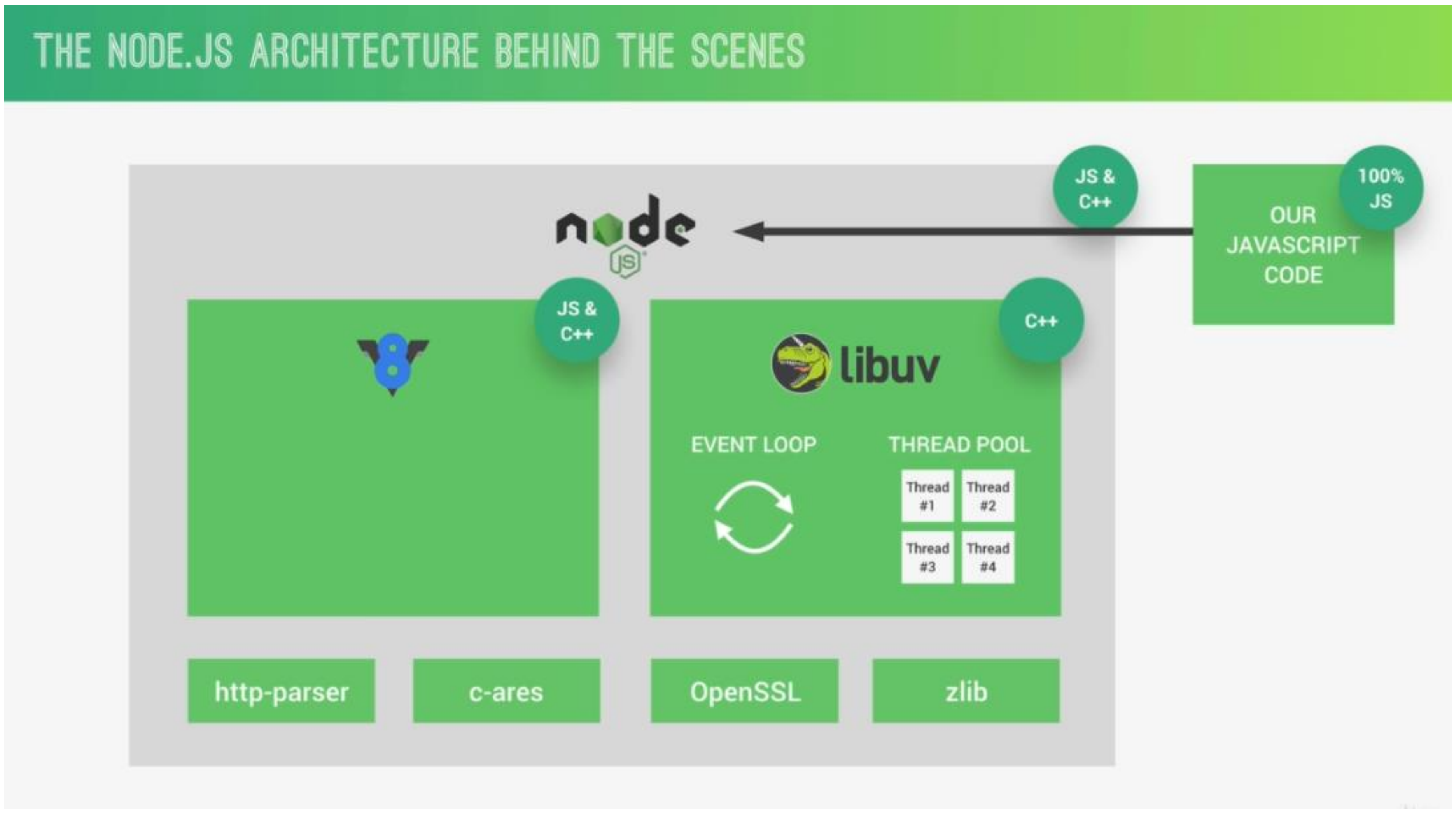
Node er dependiences/ node run time dependiencies = v8 & libuv.

V8 chara Node bujte parbe na Javascript.

V8 is fundamental architecture for Node. But v8 eka enough na node er jonno er shate libuv ase.

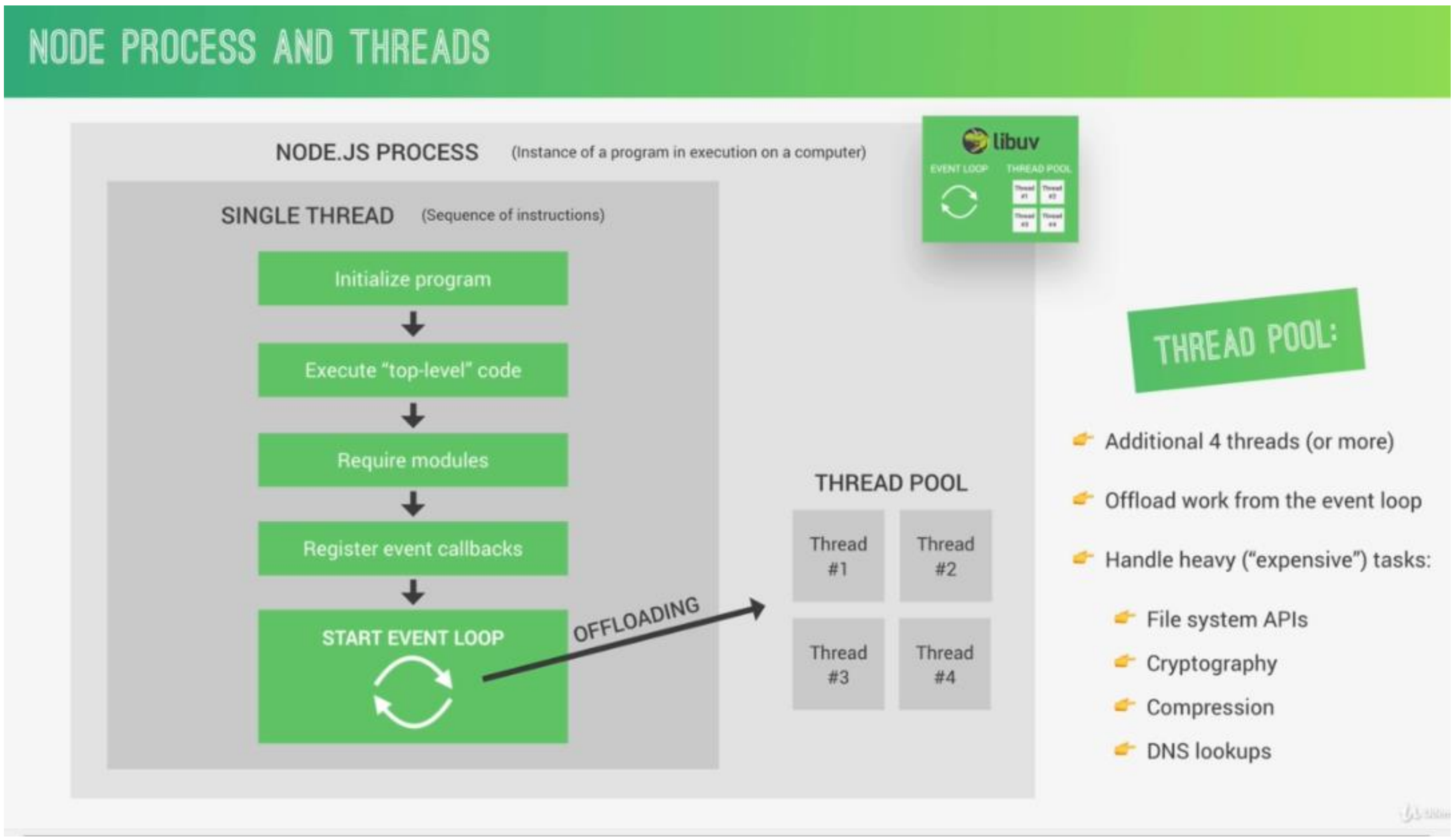
Libuv(c++ diya toiri) = open source library jetar strongly focus asynchronous IO. OT= input, output , libuv er maddome node access pai computer operating system, file sysem, networking and more. And eta implement kore event loop, thread pool.

Event loop = execute kore call back, network IO

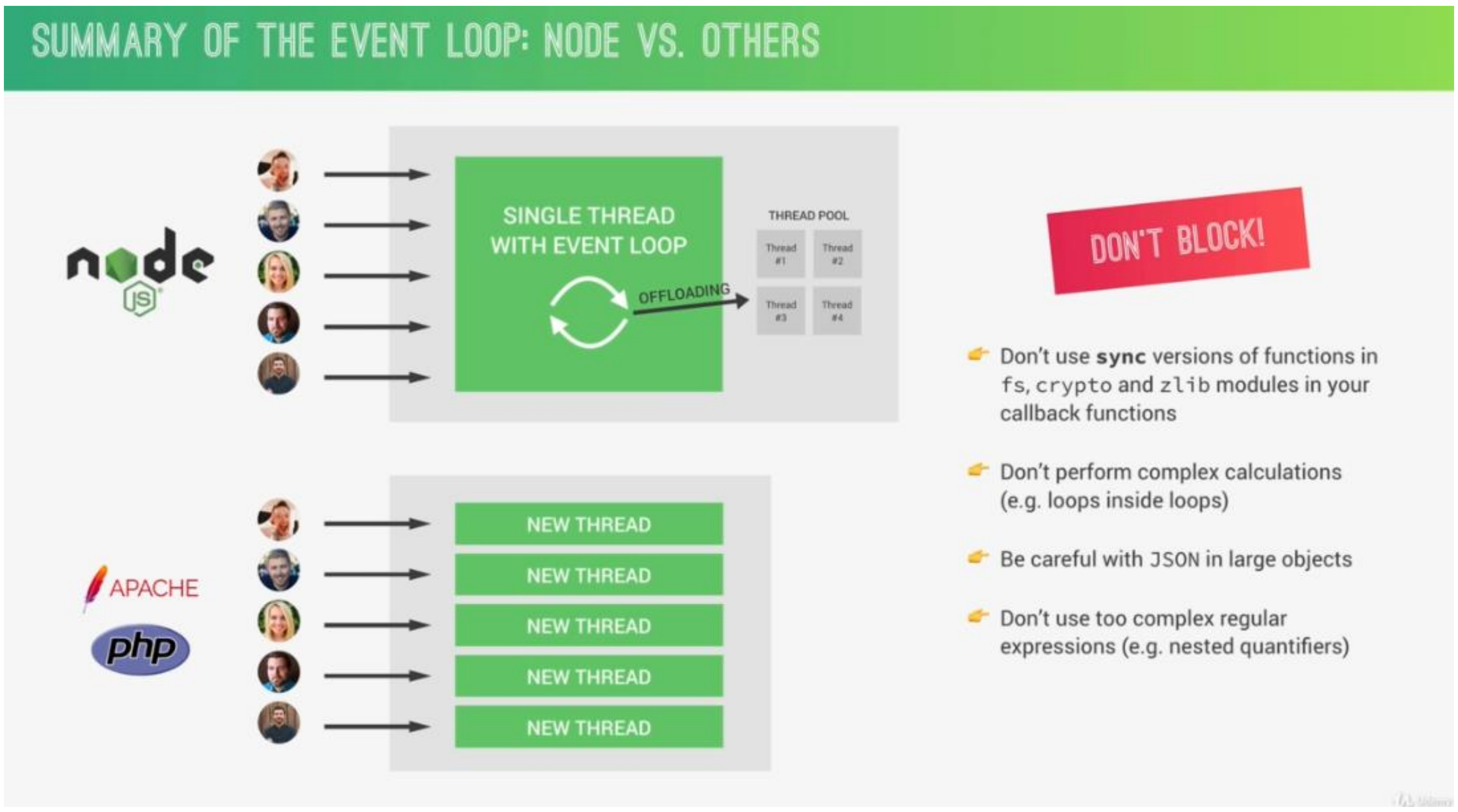


Event loop = amder application bashir bug kaj e event loop kore thake

Heavy kaj gula event loop korte pare na ar jonno libvu thear pool er madome ai kaj gulu kore



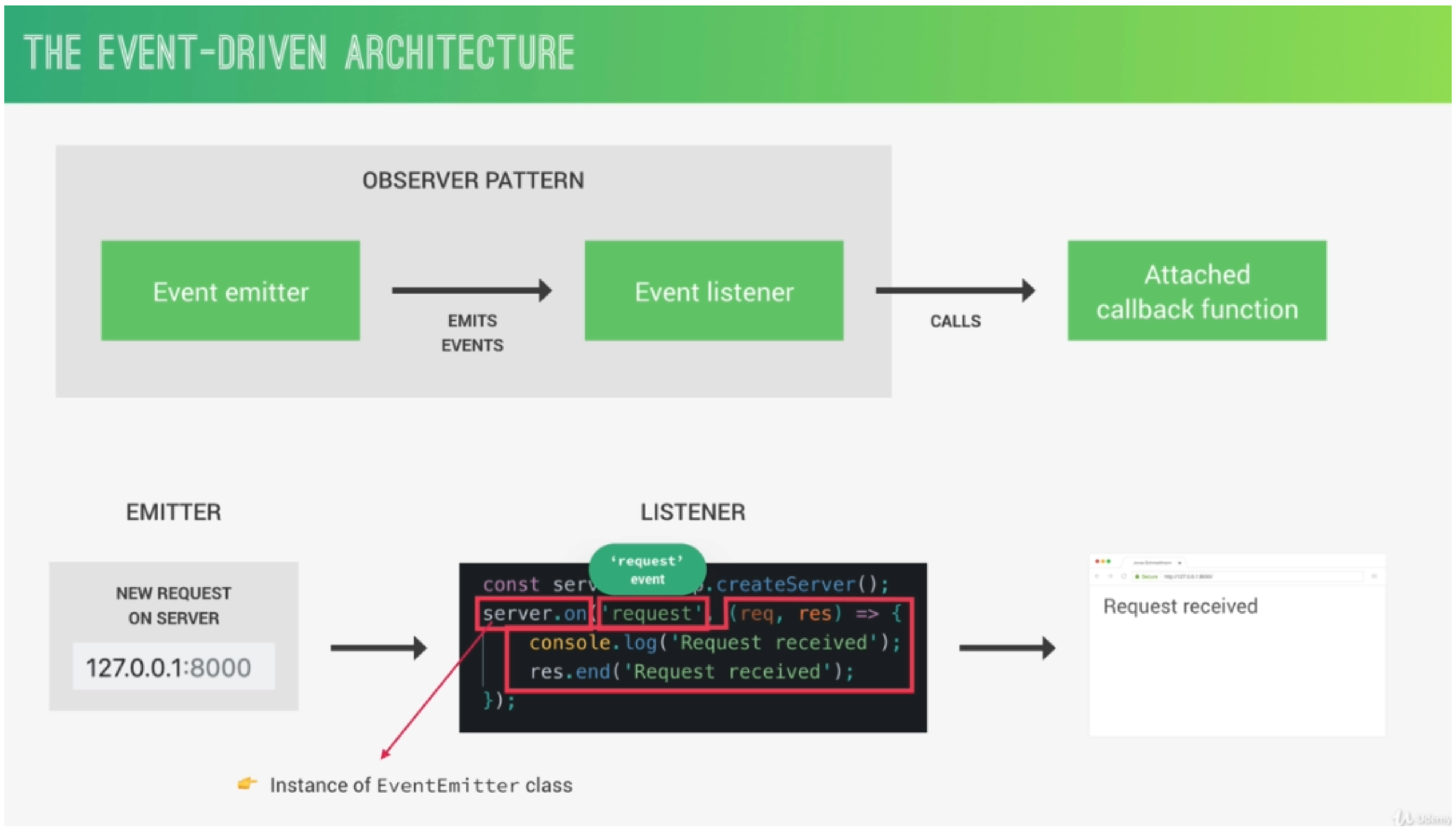
- Additional 4 threads (or more)
- Offload work from the event loop
- Handle heavy ("expensive") tasks:
  - File system APIs
  - Cryptography
  - Compression
  - DNS lookups



SUMMARY OF THE EVENT LOOP: NODE VS. OTHERS

DON'T BLOCK!

- Don't use **sync** versions of functions in `fs`, `crypto` and `zlib` modules in your callback functions
- Don't perform complex calculations (e.g. loops inside loops)
- Be careful with `JSON` in large objects
- Don't use too complex regular expressions (e.g. nested quantifiers)

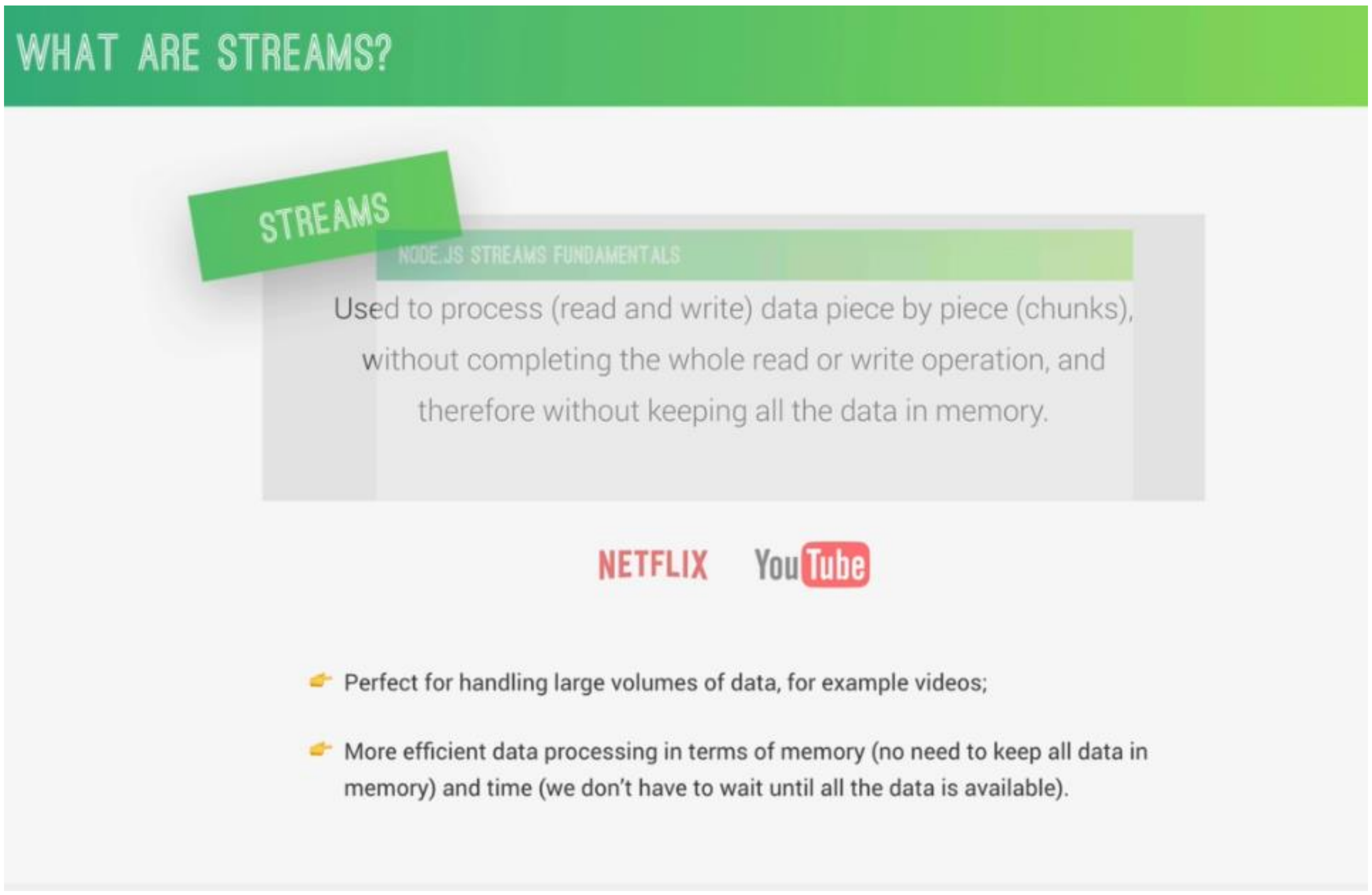


THE EVENT-DRIVEN ARCHITECTURE

```
const ser          .createServer();
server.on('request', (req, res) => {
  console.log('Request received');
  res.end('Request received');
});
```

- Instance of `EventEmitter` class

Nodes core module like http, file system, timer are built around an event-driven architecture.



WHAT ARE STREAMS?

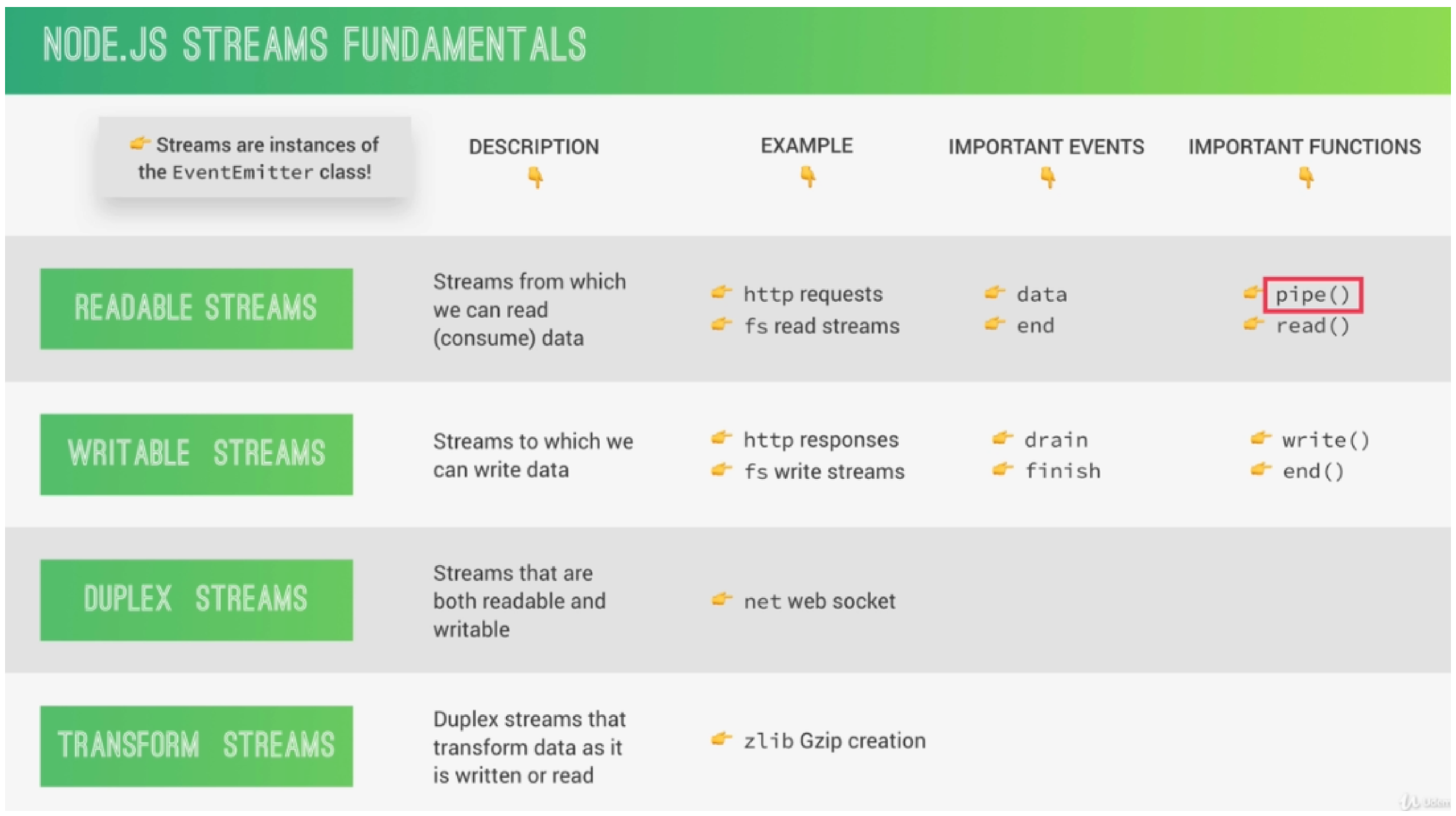
STREAMS

NODE.JS STREAMS FUNDAMENTALS

Used to process (read and write) data piece by piece (chunks), without completing the whole read or write operation, and therefore without keeping all the data in memory.

NETFLIX YouTube

- Perfect for handling large volumes of data, for example videos;
- More efficient data processing in terms of memory (no need to keep all data in memory) and time (we don't have to wait until all the data is available).



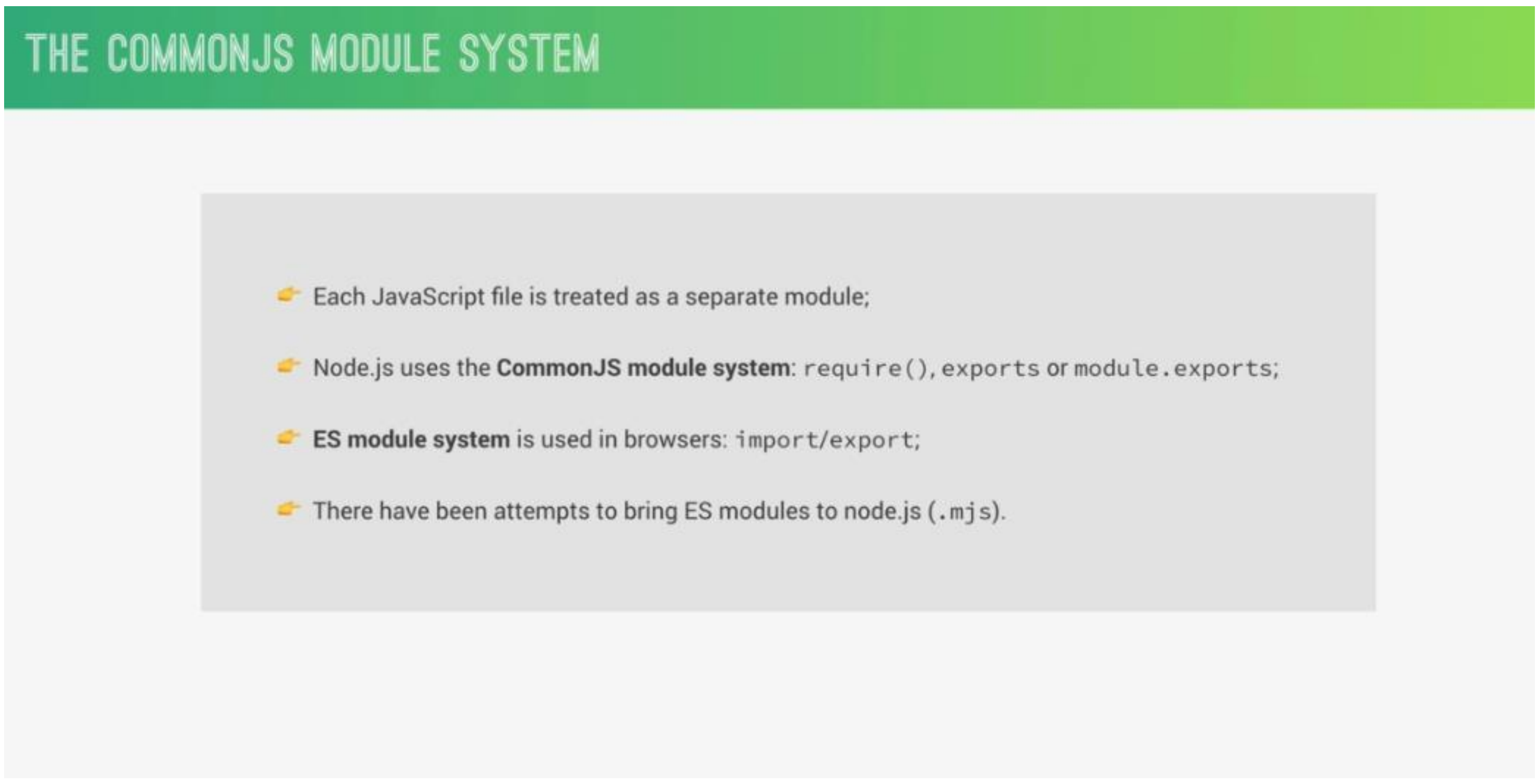
NODE.JS STREAMS FUNDAMENTALS

Streams are instances of the `EventEmitter` class!	DESCRIPTION	EXAMPLE	IMPORTANT EVENTS	IMPORTANT FUNCTIONS

READABLE STREAMS	Streams from which we can read (consume) data	`http` requests
		`fs` read streams
			`data`
			`end`
				`pipe()`
				`read()`

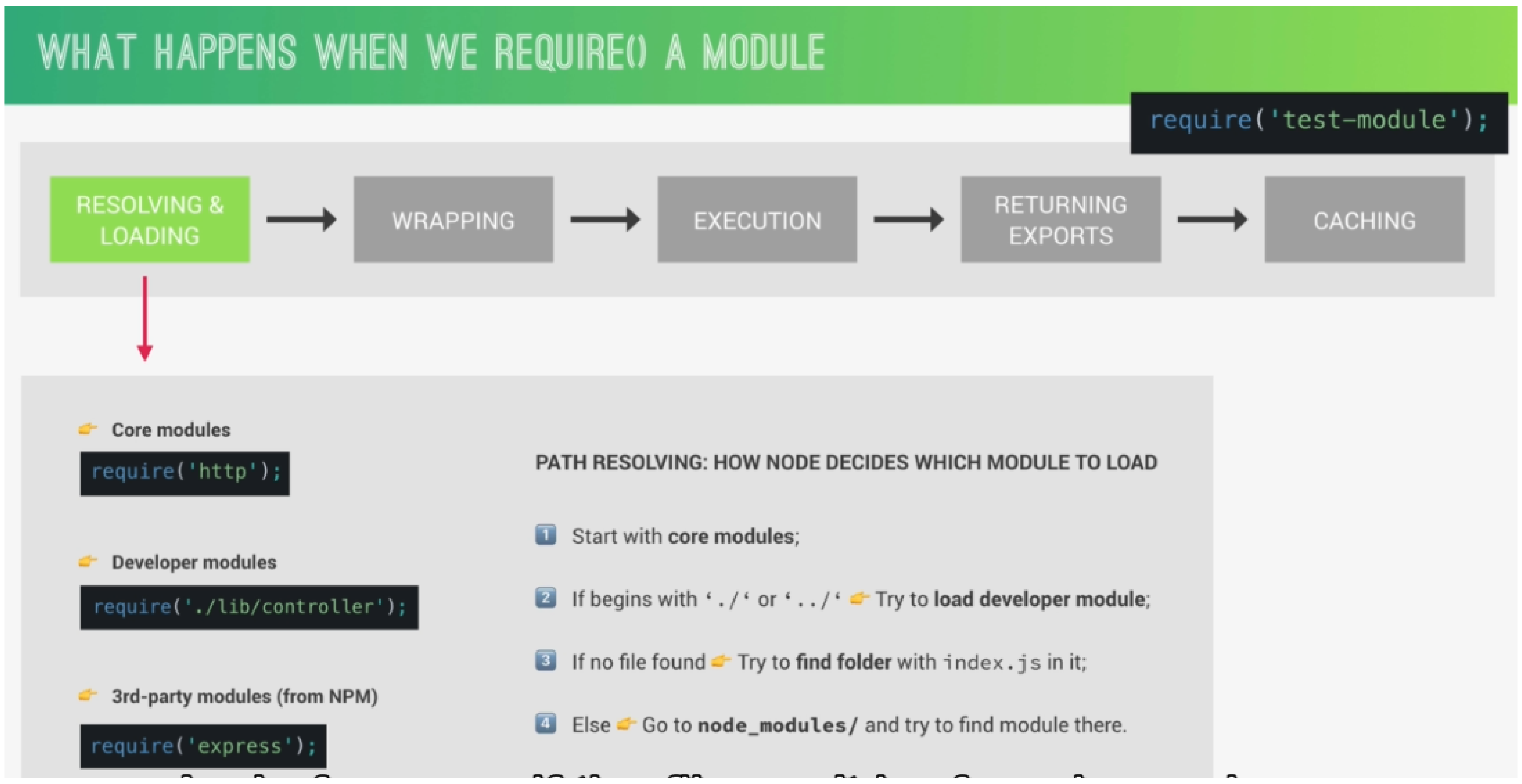
WRITABLE STREAMS	Streams to which we can write data	`http` responses
		`fs` write streams
			`drain`
			`finish`
				`write()`
				`end()`

DUPLEX STREAMS	Streams that are both readable and writable	`net` web socket		
TRANSFORM STREAMS	Duplex streams that transform data as it is written or read	`zlib` Gzip creation		



THE COMMONJS MODULE SYSTEM

- Each JavaScript file is treated as a separate module;
- Node.js uses the **CommonJS module system**: `require()`, `exports` or `module.exports`;
- **ES module system** is used in browsers: `import/export`;
- There have been attempts to bring ES modules to node.js (`.mjs`).



WHAT HAPPENS WHEN WE REQUIRE() A MODULE

```
require('test-module');
```

- **Core modules**

```
require('http');
```

- **Developer modules**

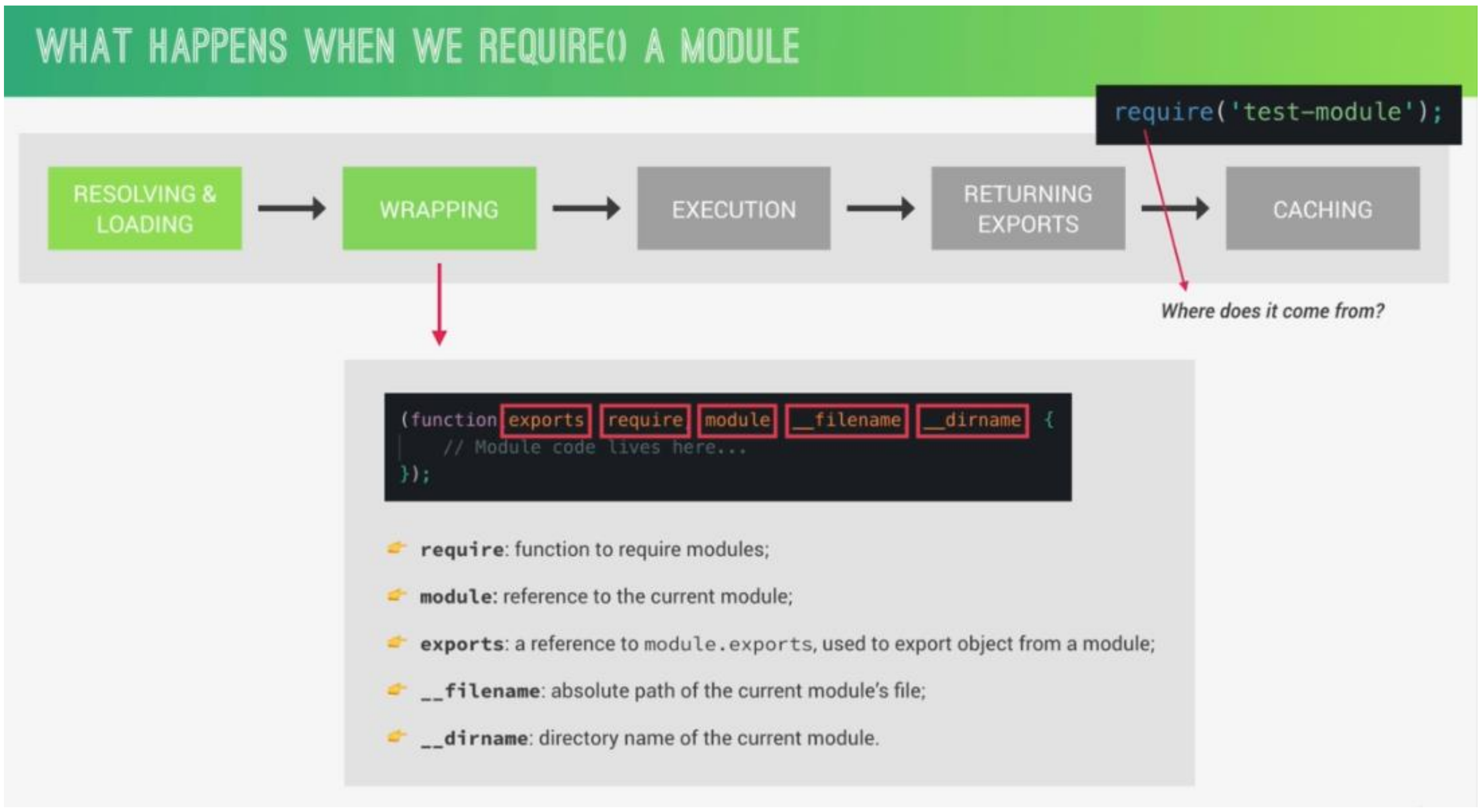
```
require('./lib/controller');
```

- **3rd-party modules (from NPM)**

```
require('express');
```

PATH RESOLVING: HOW NODE DECIDES WHICH MODULE TO LOAD

1. Start with **core modules**;
2. If begins with `'./'` or `'../'` Try to **load developer module**;
3. If no file found Try to **find folder** with `index.js` in it;
4. Else Go to **`node_modules/`** and try to find module there.



WHAT HAPPENS WHEN WE REQUIRE() A MODULE

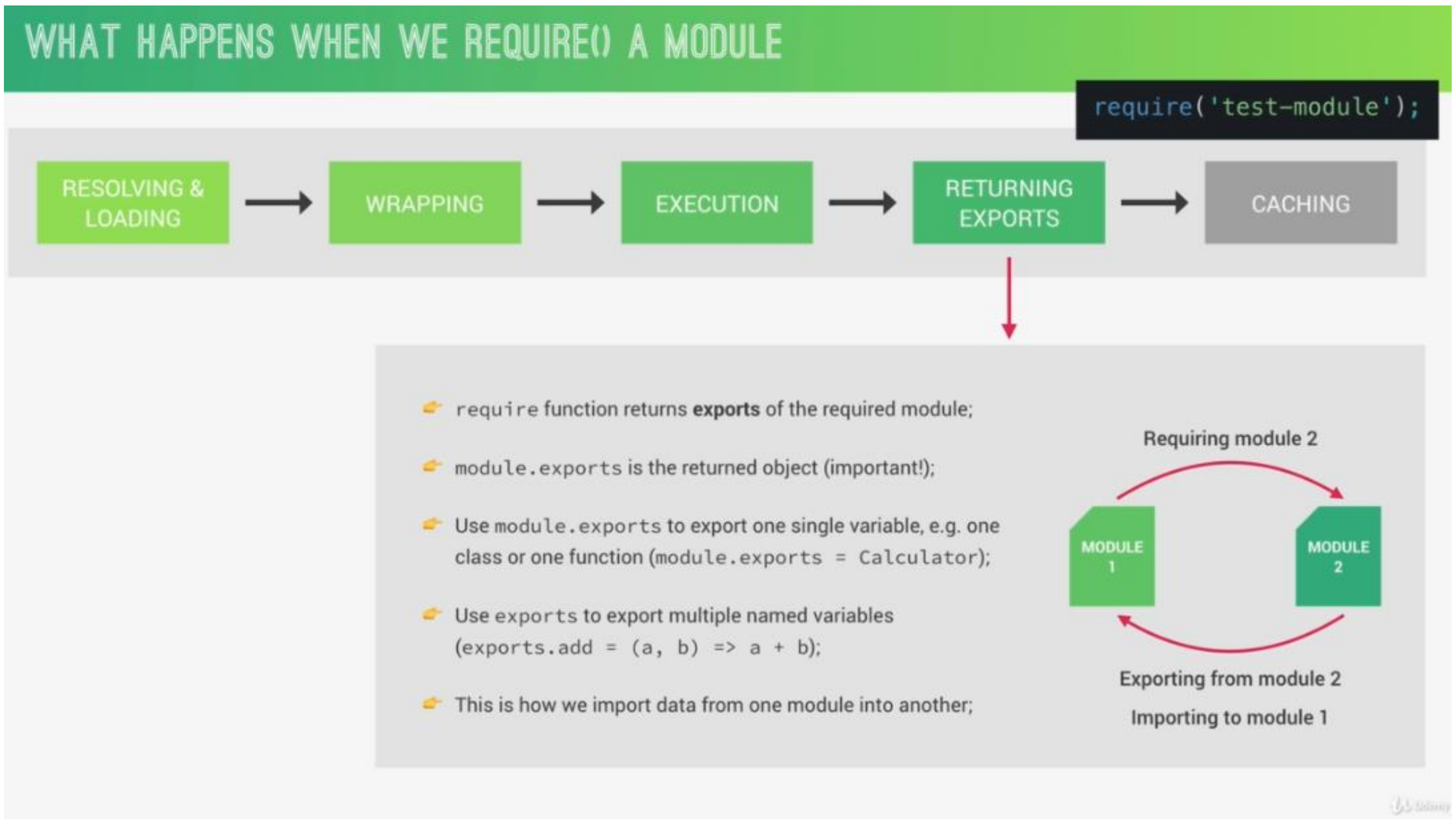
```
require('test-module');
```

RESOLVING & LOADING → WRAPPING → EXECUTION → RETURNING EXPORTS → CACHING

Where does it come from?

```
(function(exports, require, module, __filename, __dirname) {
    // Module code lives here...
});
```

- **`require`**: function to require modules;
- **`module`**: reference to the current module;
- **`exports`**: a reference to `module.exports`, used to export object from a module;
- **`__filename`**: absolute path of the current module's file;
- **`__dirname`**: directory name of the current module.



WHAT HAPPENS WHEN WE REQUIRE() A MODULE

```
require('test-module');
```

RESOLVING & LOADING → WRAPPING → EXECUTION → RETURNING EXPORTS → CACHING

- `require` function returns **exports** of the required module;
- `module.exports` is the returned object (important!);
- Use `module.exports` to export one single variable, e.g. one class or one function (`module.exports = Calculator`);
- Use `exports` to export multiple named variables (`exports.add = (a, b) => a + b`);
- This is how we import data from one module into another;

Console.log(arguments); // arguments is an array in JS. Ai array contain all values that passed into a function.

//event-loop, event, stream code

```javascript
//event-loop
const fs = require("fs");
const crypto = require("crypto");

const start = Date.now();
process.env.UV_THREADPOOL_SIZE = 4;

setTimeout(() => console.log("Timer 1 finished"), 0);
setImmediate(() => console.log("Immediate 1 finished"));

fs.readFile("test-file.txt", () => {
  console.log("I/O finished");
  console.log("----------------");

  setTimeout(() => console.log("Timer 2 finished"), 0);
  setTimeout(() => console.log("Timer 3 finished"), 3000);
  setImmediate(() => console.log("Immediate 2 finished"));

  process.nextTick(() => console.log("Process.nextTick"));

  crypto.pbkdf2Sync("password", "salt", 100000, 1024, "sha512");
  console.log(Date.now() - start, "Password encrypted");

  crypto.pbkdf2Sync("password", "salt", 100000, 1024, "sha512");
  console.log(Date.now() - start, "Password encrypted");

  crypto.pbkdf2Sync("password", "salt", 100000, 1024, "sha512");
  console.log(Date.now() - start, "Password encrypted");

  crypto.pbkdf2Sync("password", "salt", 100000, 1024, "sha512");
  console.log(Date.now() - start, "Password encrypted");
});

console.log("Hello from the top-level code");



//events
const EventEmitter = require("events");
const http = require("http");

class Sales extends EventEmitter {
  constructor() {
    super();
  }
}

const myEmitter = new Sales();

myEmitter.on("newSale", () => {
  console.log("There was a new sale!");
});

myEmitter.on("newSale", () => {
  console.log("Costumer name: Jonas");
});

myEmitter.on("newSale", (stock) => {
  console.log(`There are now ${stock} items left in stock.`);
});

myEmitter.emit("newSale", 9);

//////////////////

const server = http.createServer();

server.on("request", (req, res) => {
  console.log("Request received!");
  console.log(req.url);
  res.end("Request received");
});

server.on("request", (req, res) => {
  console.log("Another request 😀");
});

server.on("close", () => {
  console.log("Server closed");
});

server.listen(8000, "127.0.0.1", () => {
  console.log("Waiting for requests...");
});



//streams
const fs = require("fs");
const server = require("http").createServer();

server.on("request", (req, res) => {
  // Solution 1
  // fs.readFile("test-file.txt", (err, data) => {
  //   if (err) console.log(err);
  //   res.end(data);
  // });

  // Solution 2: Streams
  // const readable = fs.createReadStream("test-file.txt");
  // readable.on("data", chunk => {
```

```
  //   res.write(chunk);
  // });
  // readable.on("end", () => {
  //   res.end();
  // });
  // readable.on("error", err => {
  //   console.log(err);
  //   res.statusCode = 500;
  //   res.end("File not found!");
  // });

  // Solution 3
  const readable = fs.createReadStream("test-file.txt");
  readable.pipe(res);
  // readableSource.pipe(writeableDest)
});

server.listen(8000, "127.0.0.1", () => {
  console.log("Listening...");
});
```