JONAS.IO
SCHMEDTMANN

Subscribe here

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

Follow me here

@JONASSCHMEDTMAN

SLIDES FOR THEORY LECTURES

(DON'T SKIP THEM, THEY ARE SUPER IMPORTANT 🤓)

📖 TABLE OF CONTENTS: THEORY LECTURES (CLICK THE TITLES)

SECTION 2 — INTRODUCTION TO NODE.JS

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

INTRODUCTION TO NODE.JS

LECTURE

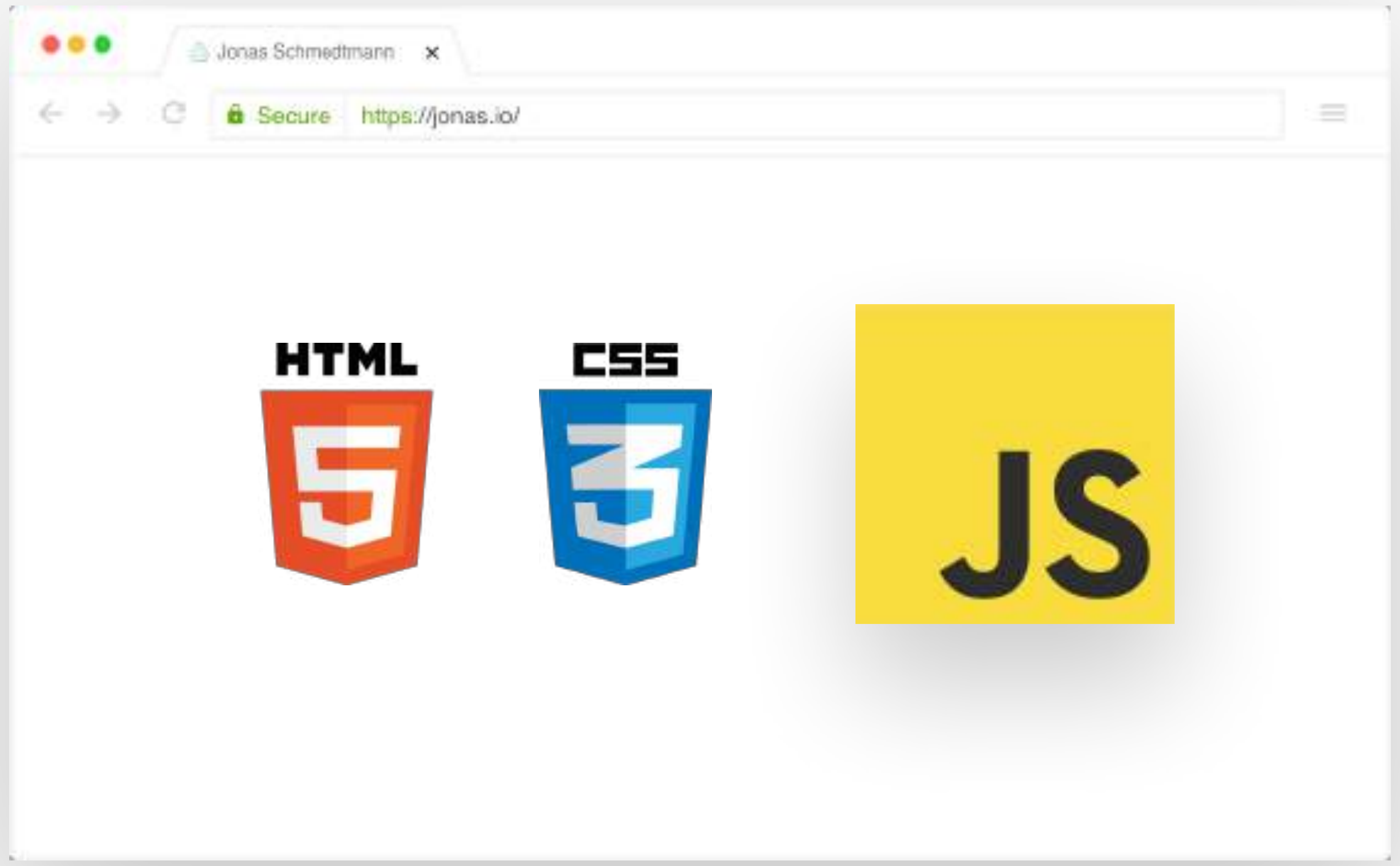
WHAT IS NODE.JS AND WHY USE IT?

WHAT IS NODE.JS?

NODE.JS

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

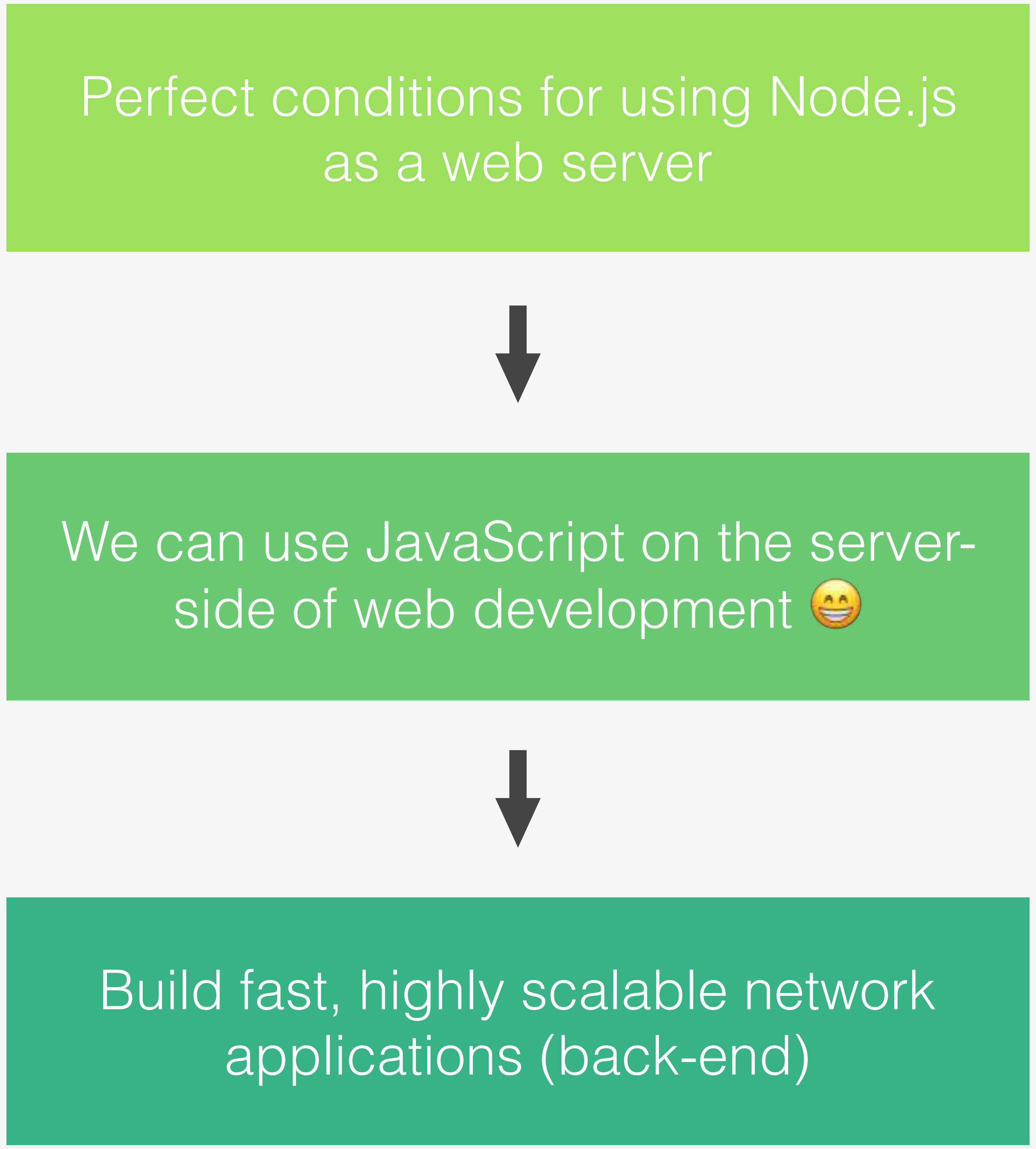
NODE.JS: JAVASCRIPT OUTSIDE OF THE BROWSER



BROWSER

NODE.JS

JAVASCRIPT ON THE SERVER!



Perfect conditions for using Node.js as a web server

We can use JavaScript on the server-side of web development 😁

Build fast, highly scalable network applications (back-end)

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).

RAILS php

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

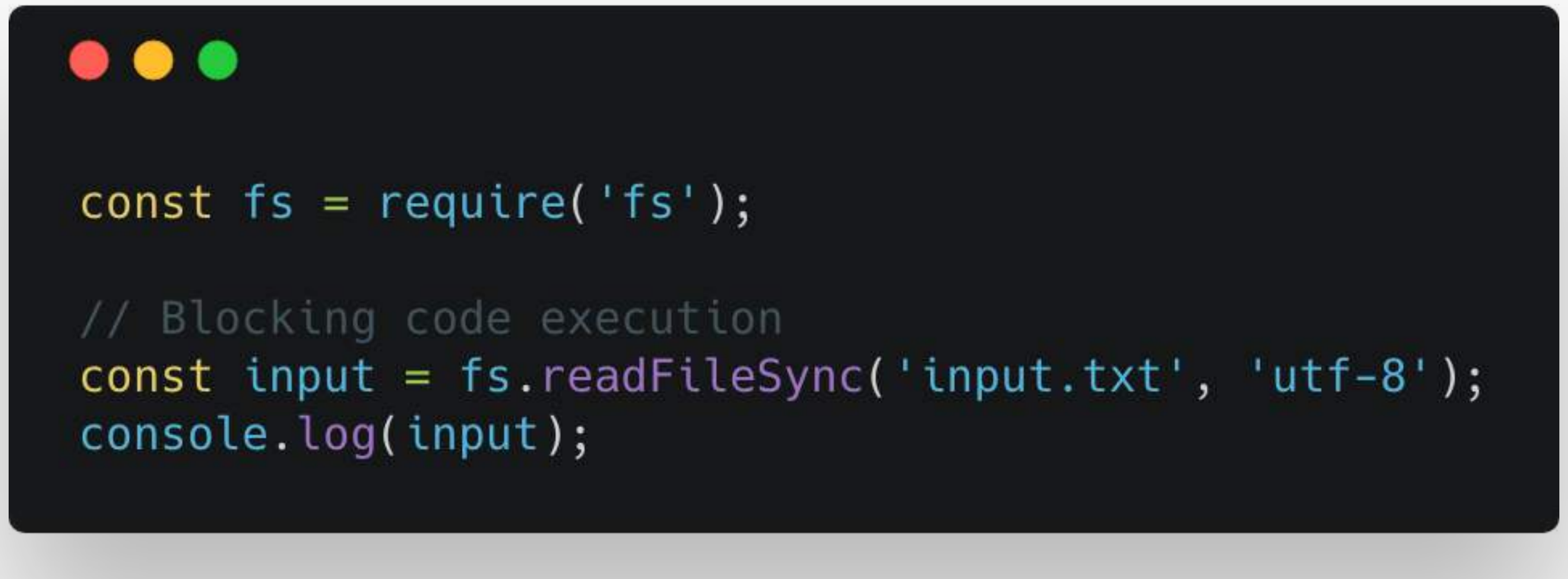
SECTION

INTRODUCTION TO NODE.JS

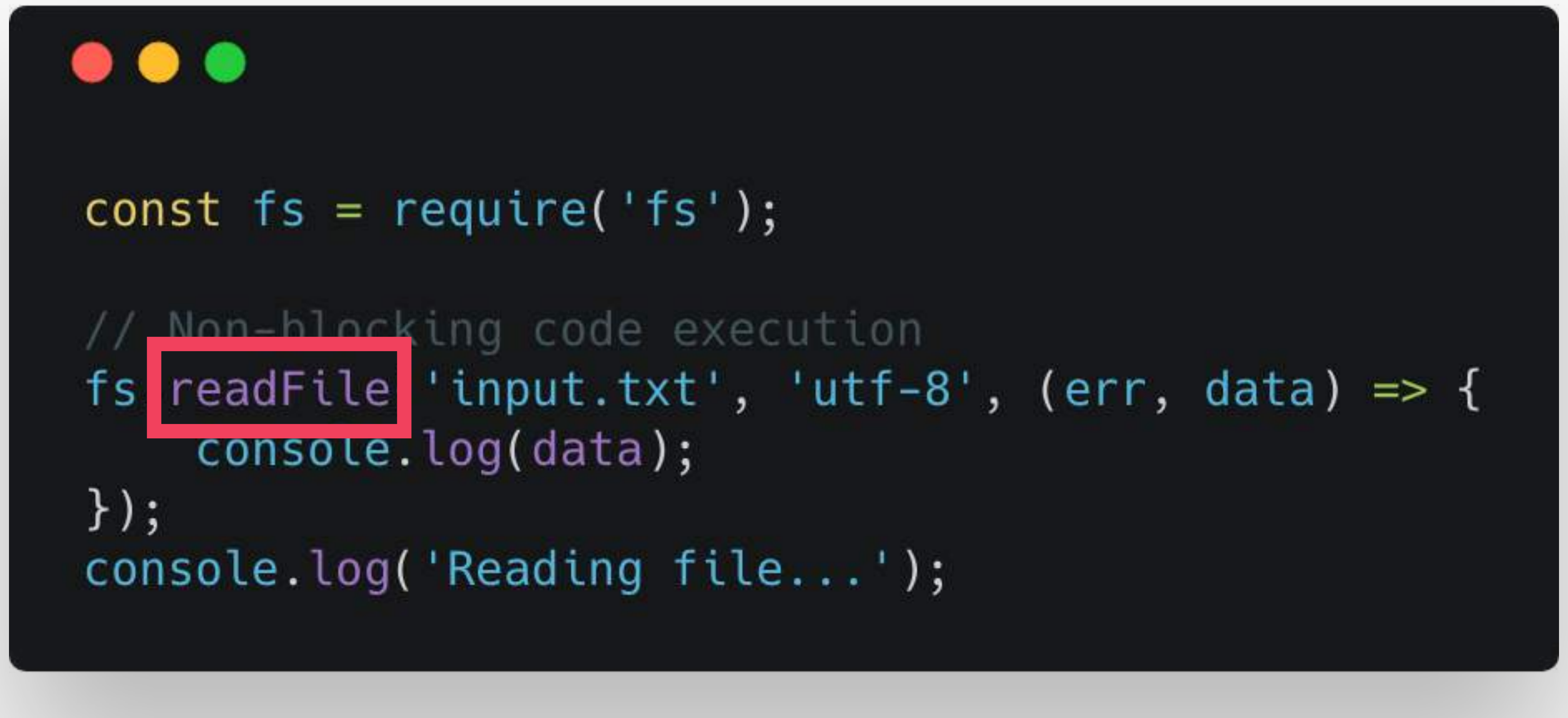
LECTURE

BLOCKING AND NON-BLOCKING: ASYNCHRONOUS NATURE OF NODE.JS

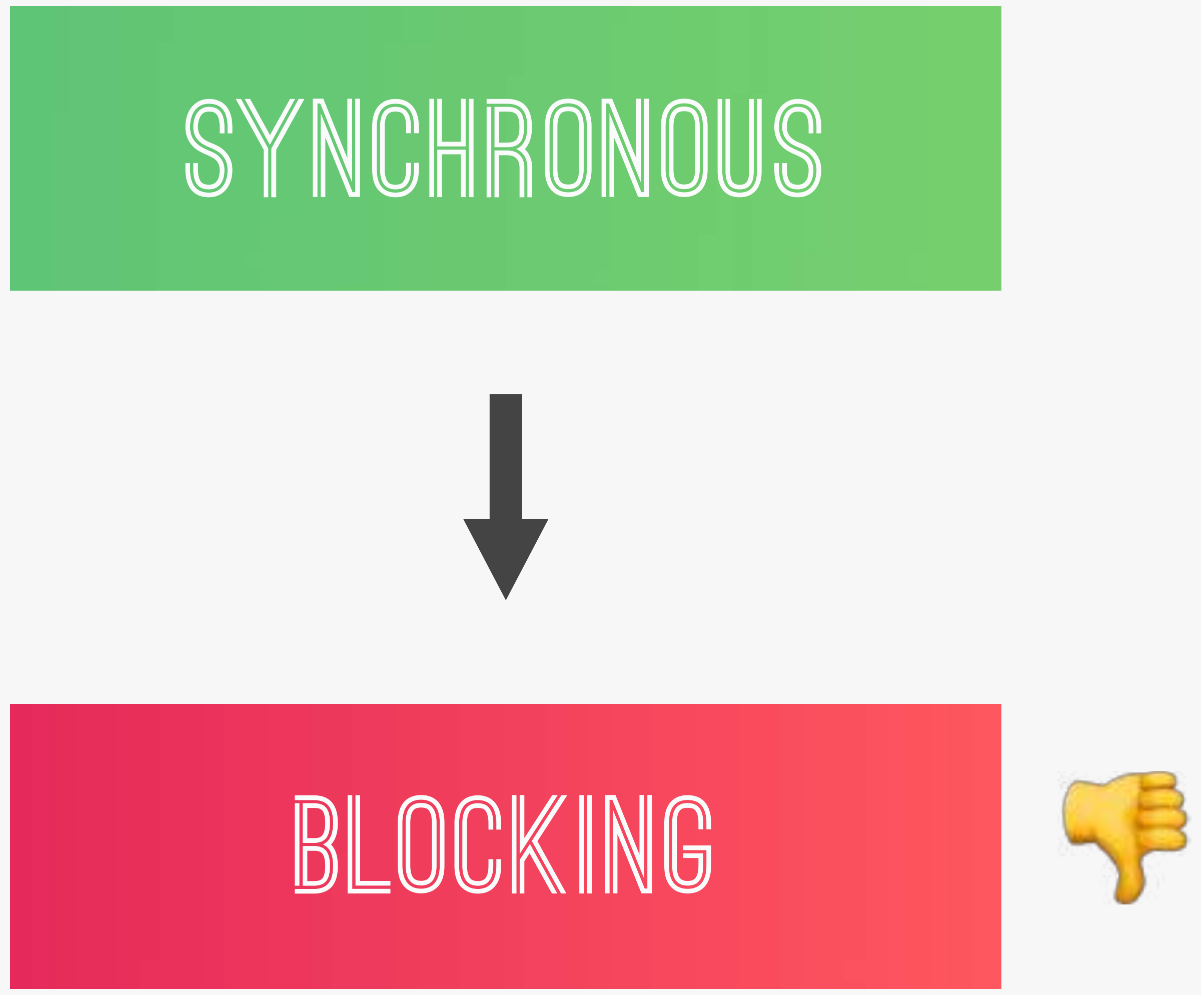
SYNCHRONOUS VS. ASYNCHRONOUS CODE (BLOCKING VS. NON-BLOCKING)



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

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

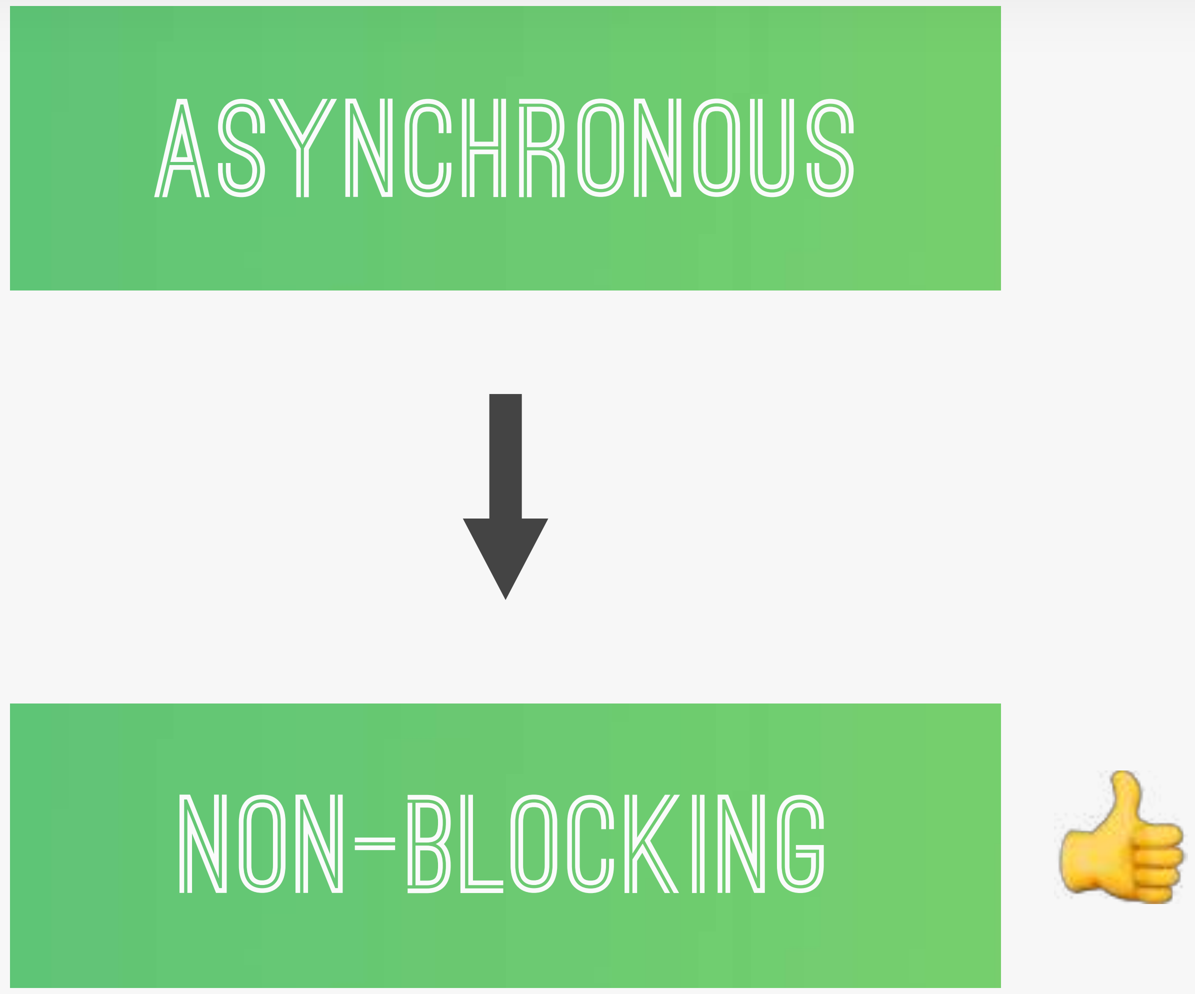


```javascript
const fs = require('fs');

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



SYNCHRONOUS

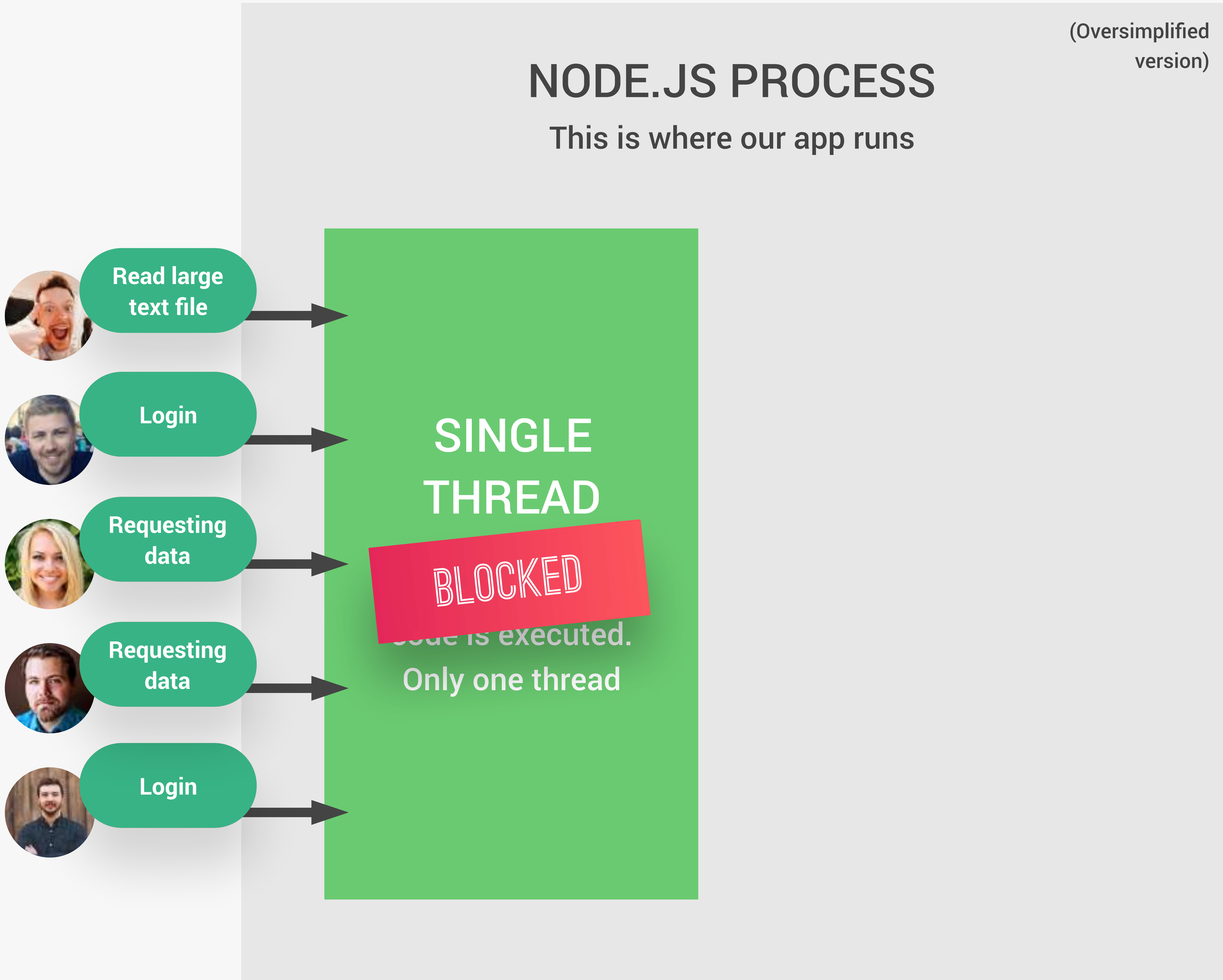
BLOCKING 👎



ASYNCHRONOUS

NON-BLOCKING 👍

THE ASYNCHRONOUS NATURE OF NODE.JS: AN OVERVIEW



(Oversimplified version)

NODE.JS PROCESS

This is where our app runs

Read large text file

Login

Requesting data

Requesting data

Login

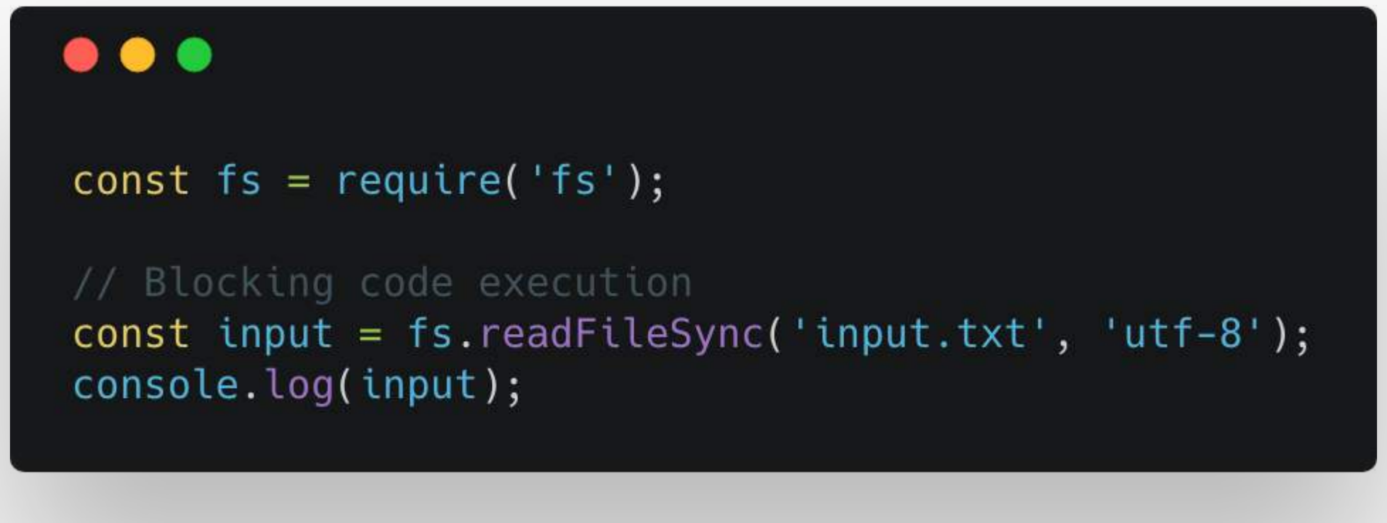
SINGLE THREAD

BLOCKED

code is executed.

Only one thread

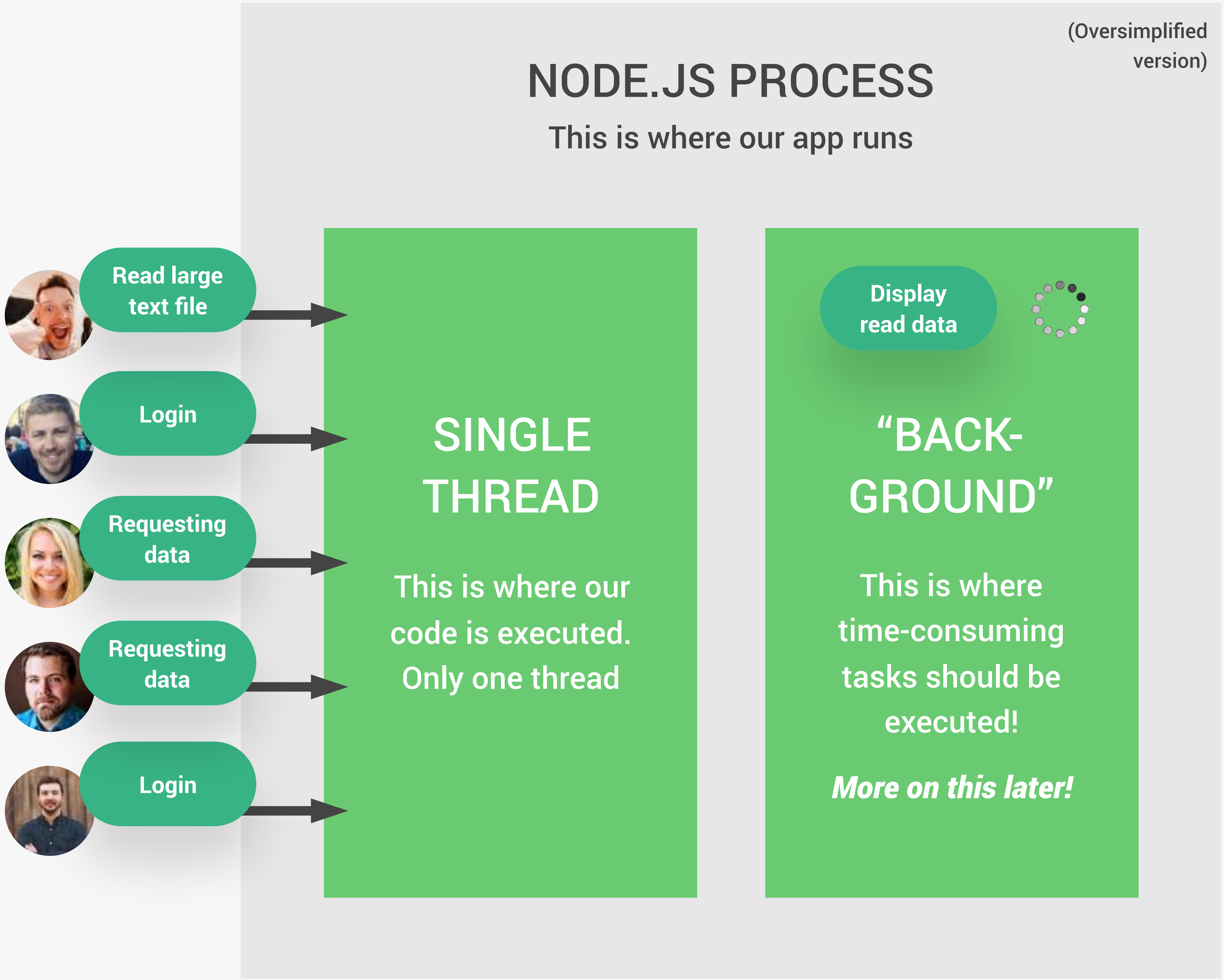
SYNCHRONOUS WAY



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

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

👉 It’s **YOUR** job as a developer to avoid this kind of situation!

THE ASYNCHRONOUS NATURE OF NODE.JS: AN OVERVIEW



(Oversimplified version)

NODE.JS PROCESS

This is where our app runs

SINGLE THREAD

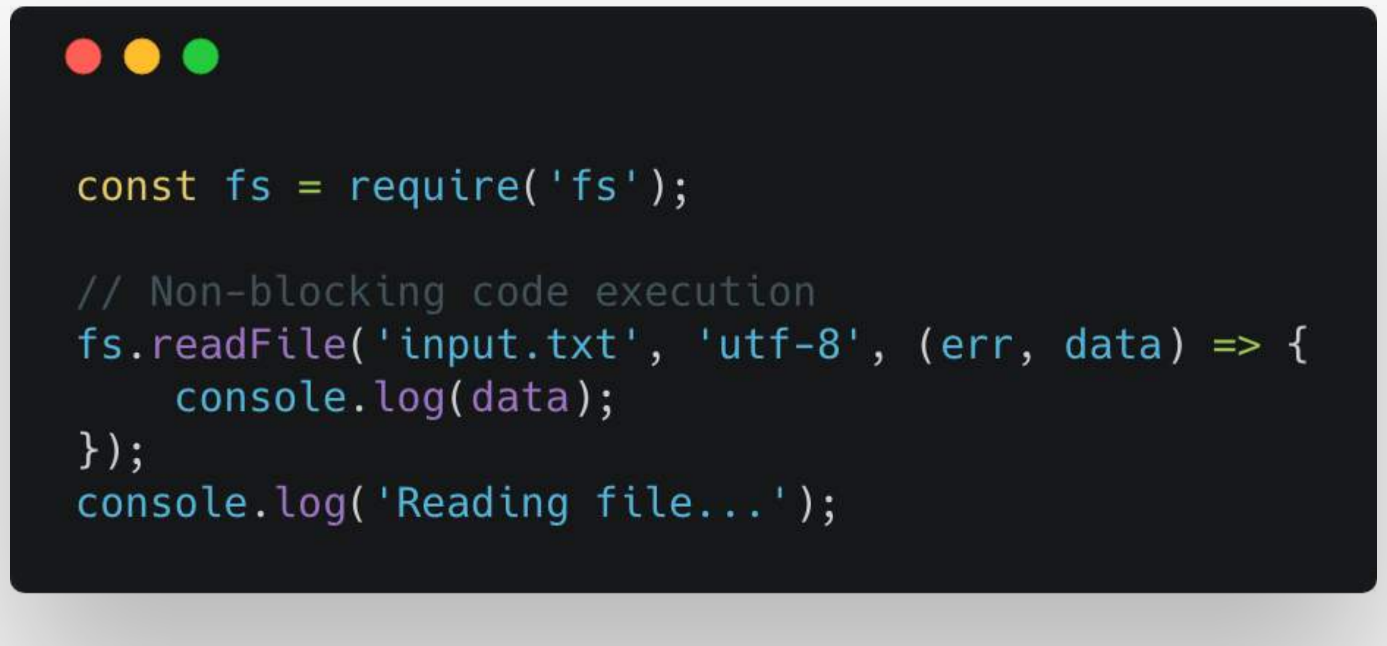
This is where our code is executed. Only one thread

"BACK-GROUND"

This is where time-consuming tasks should be executed!

More on this later!

ASYNCHRONOUS WAY



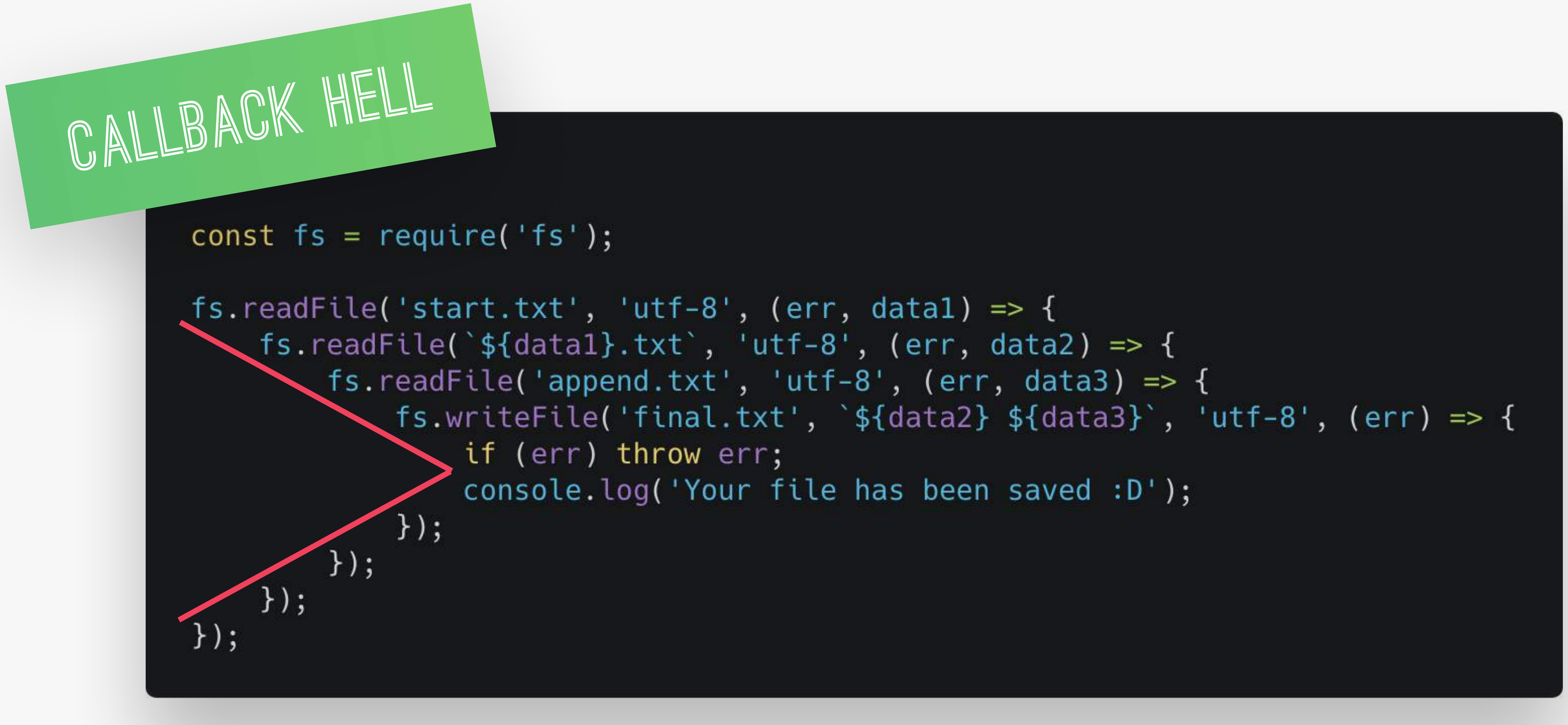
```
const fs = require('fs');

// 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...



```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]

SECTION 3 — INTRODUCTION TO BACK-END WEB DEVELOPMENT

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

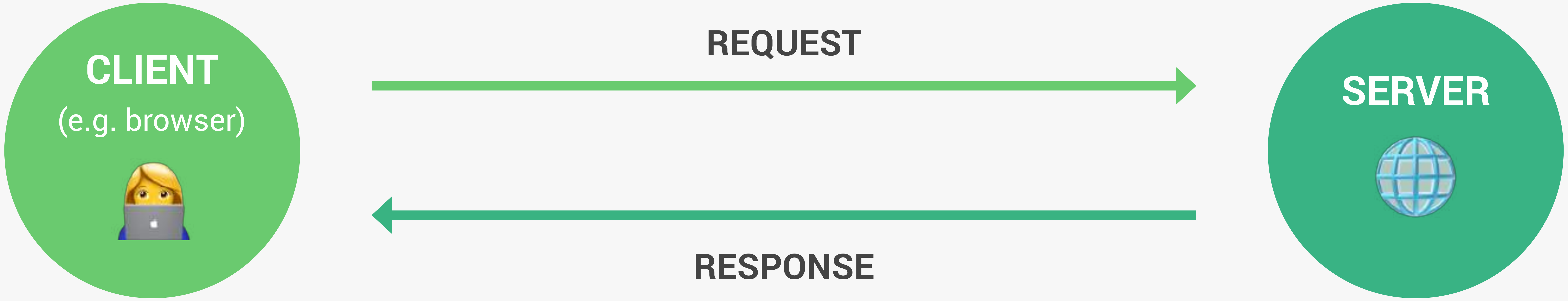
INTRODUCTION TO BACK-END WEB DEVELOPMENT

LECTURE

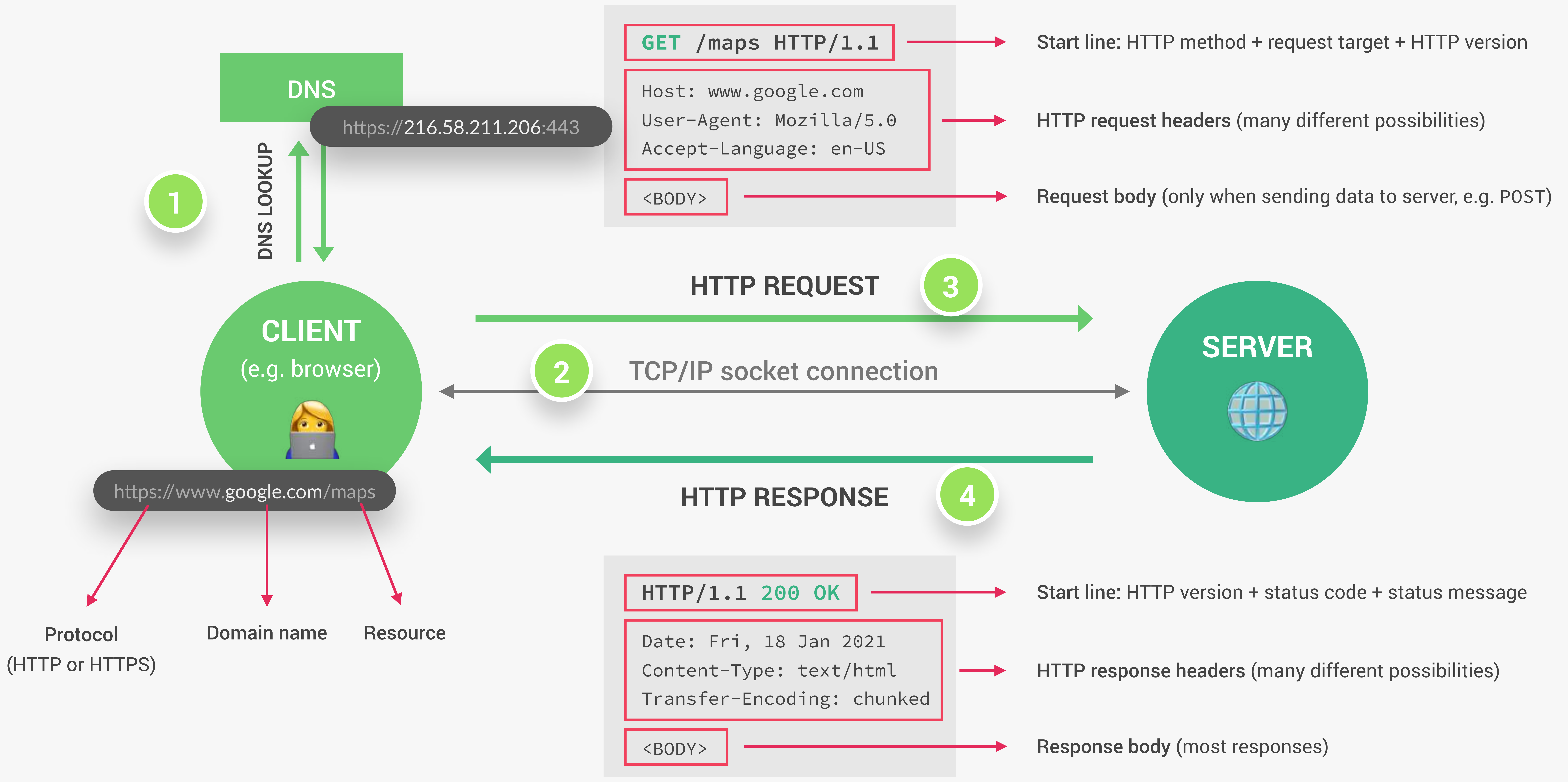
AN OVERVIEW OF HOW THE WEB WORKS

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE

👉 **Request-response model** or **Client-server architecture**



WHAT HAPPENS WHEN WE ACCESS A WEBPAGE



```
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)

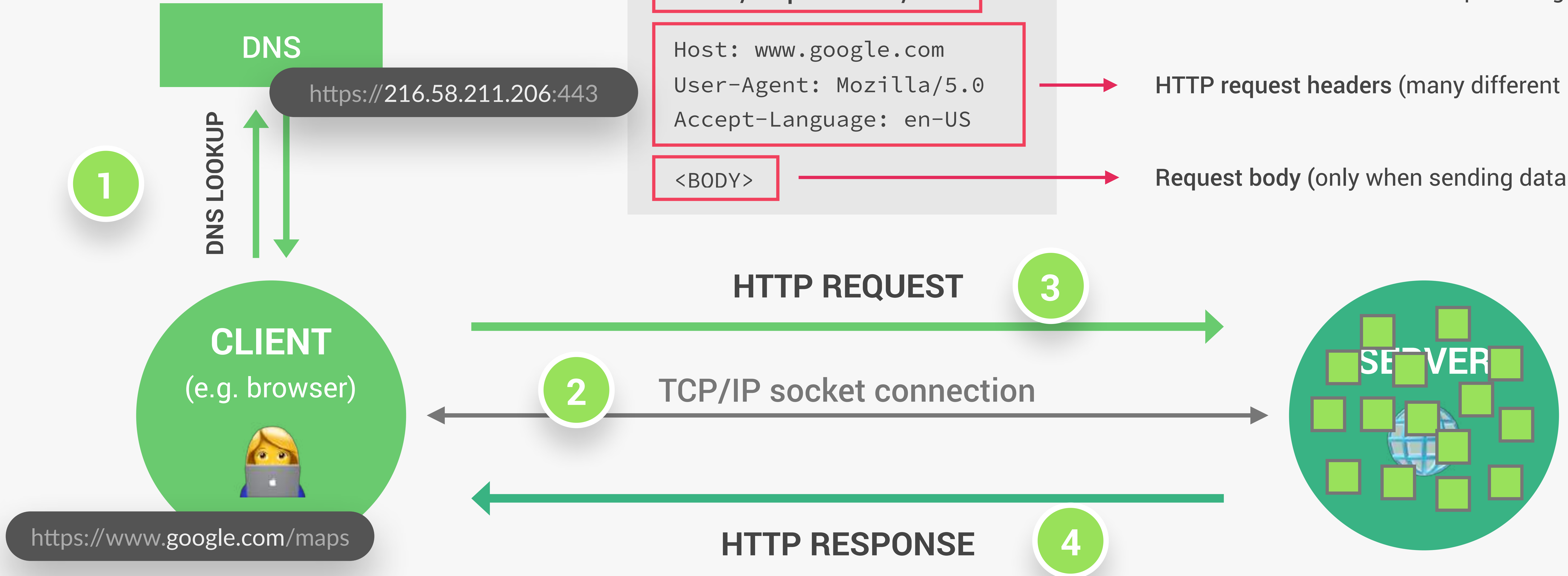
```
HTTP/1.1 200 OK

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)

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE



DNS

https://216.58.211.206:443

DNS LOOKUP

1

CLIENT

(e.g. browser)

https://www.google.com/maps

HTTP REQUEST

3

2

TCP/IP socket connection

SERVER

HTTP RESPONSE

4

```
GET /maps HTTP/1.1
```

→ **Start line:** HTTP method + request target + HTTP version

```
Host: www.google.com
User-Agent: Mozilla/5.0
Accept-Language: en-US
```

→ **HTTP request headers** (many different possibilities)

```
<BODY>
```

→ **Request body** (only when sending data to server, e.g. POST)

```
HTTP/1.1 200 OK
```

→ **Start line:** HTTP version + status code + status message

```
Date: Fri, 18 Jan 2021
Content-Type: text/html
Transfer-Encoding: chunked
```

→ **HTTP response headers** (many different possibilities)

```
<BODY>
```

→ **Response body** (most responses)

5

`index.html` is the first to be loaded

👇

Scanned for assets: JS, CSS, images

👇

Process is repeated for each file

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

INTRODUCTION TO BACK-END WEB DEVELOPMENT

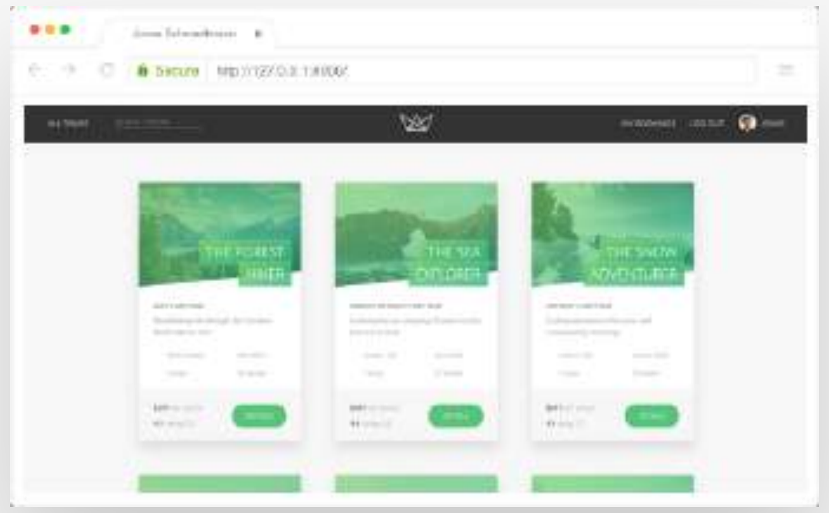
LECTURE

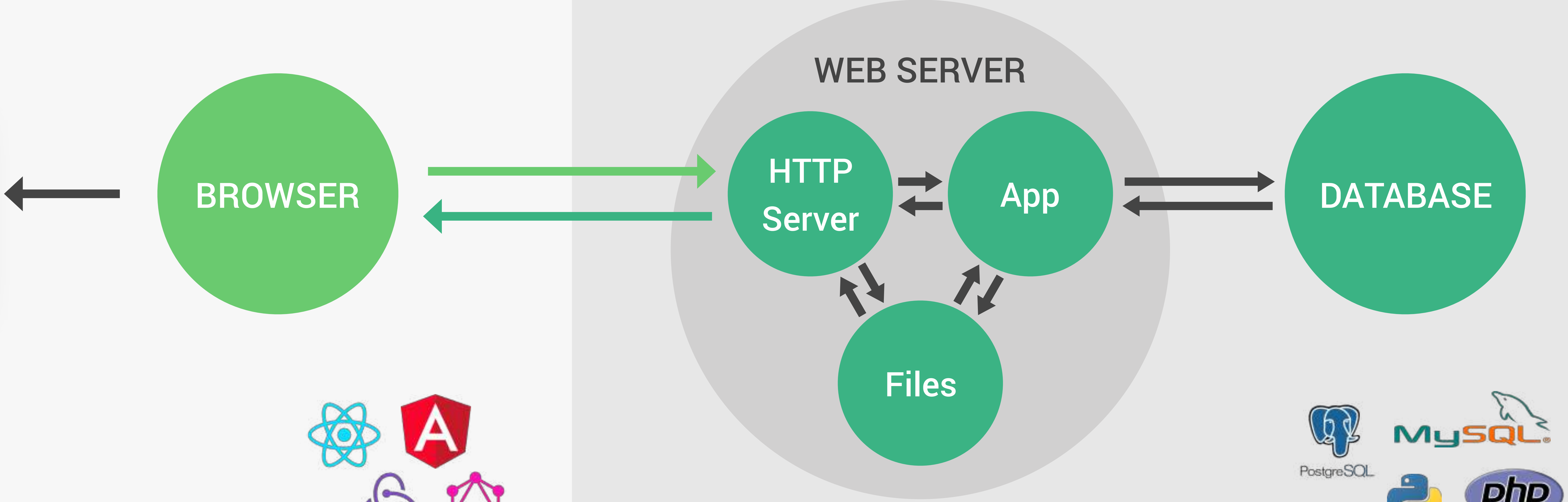
FRONT-END VS. BACK-END WEB DEVELOPMENT

FRONT-END AND BACK-END

FRONT-END

BACK-END





FRONT-END STACK

HTML CSS JS

BACK-END STACK

node mongoDB

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

INTRODUCTION TO BACK-END WEB DEVELOPMENT

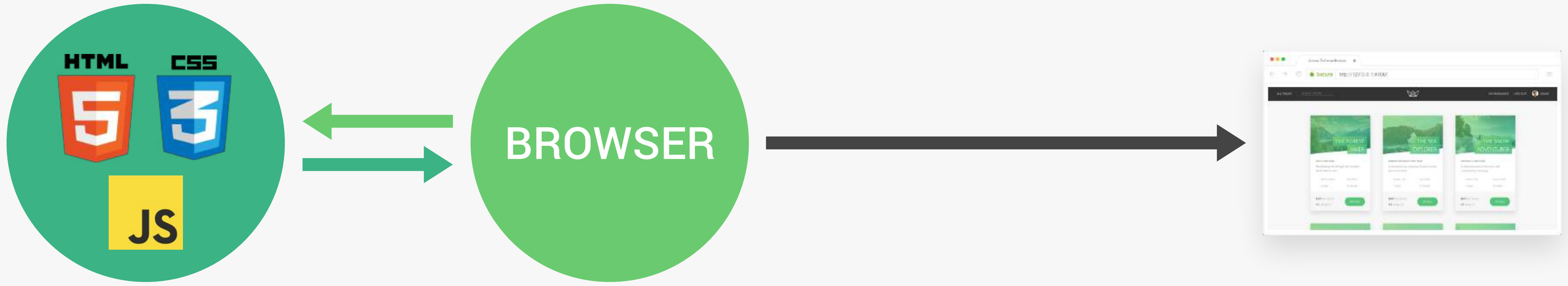
LECTURE

STATIC VS DYNAMIC VS API

@JONASSCHMEDTMAN

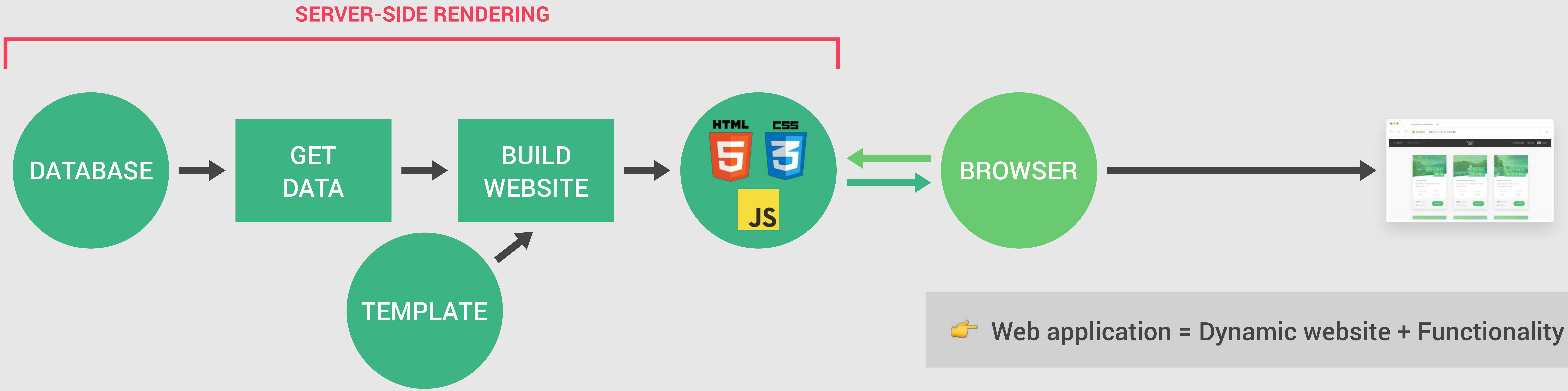
STATIC WEBSITES VS DYNAMIC WEBSITES

STATIC



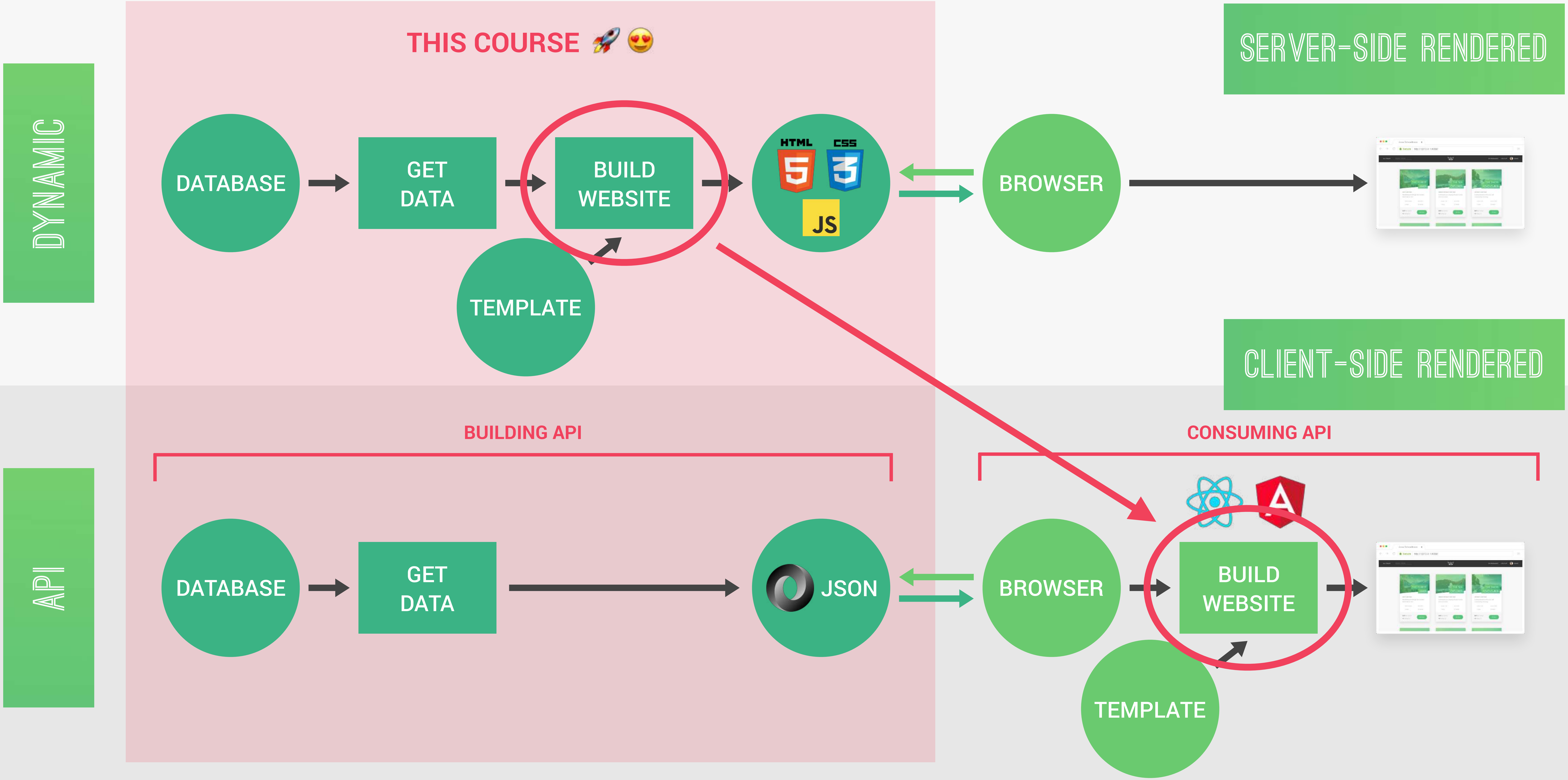
👉 JavaScript ≠ Dynamic

DYNAMIC



👉 Web application = Dynamic website + Functionality

DYNAMIC WEBSITES VS API-POWERED WEBSITES



THIS COURSE 🚀 😍

SERVER-SIDE RENDERED

DYNAMIC

DATABASE → GET DATA → BUILD WEBSITE → HTML CSS JS ⇄ BROWSER →

TEMPLATE

CLIENT-SIDE RENDERED

API

BUILDING API

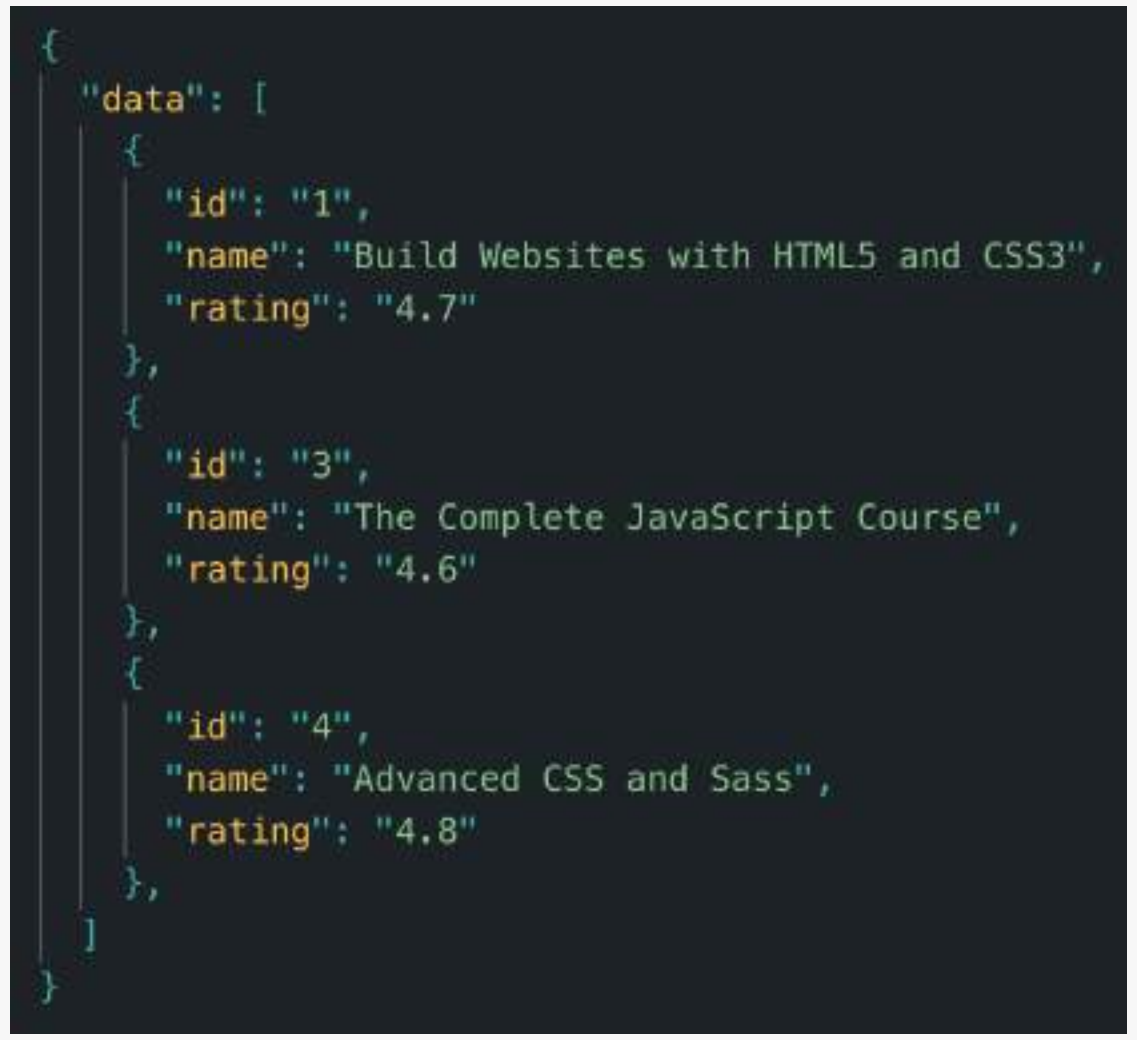
CONSUMING API

DATABASE → GET DATA → JSON ⇄ BROWSER → BUILD WEBSITE →

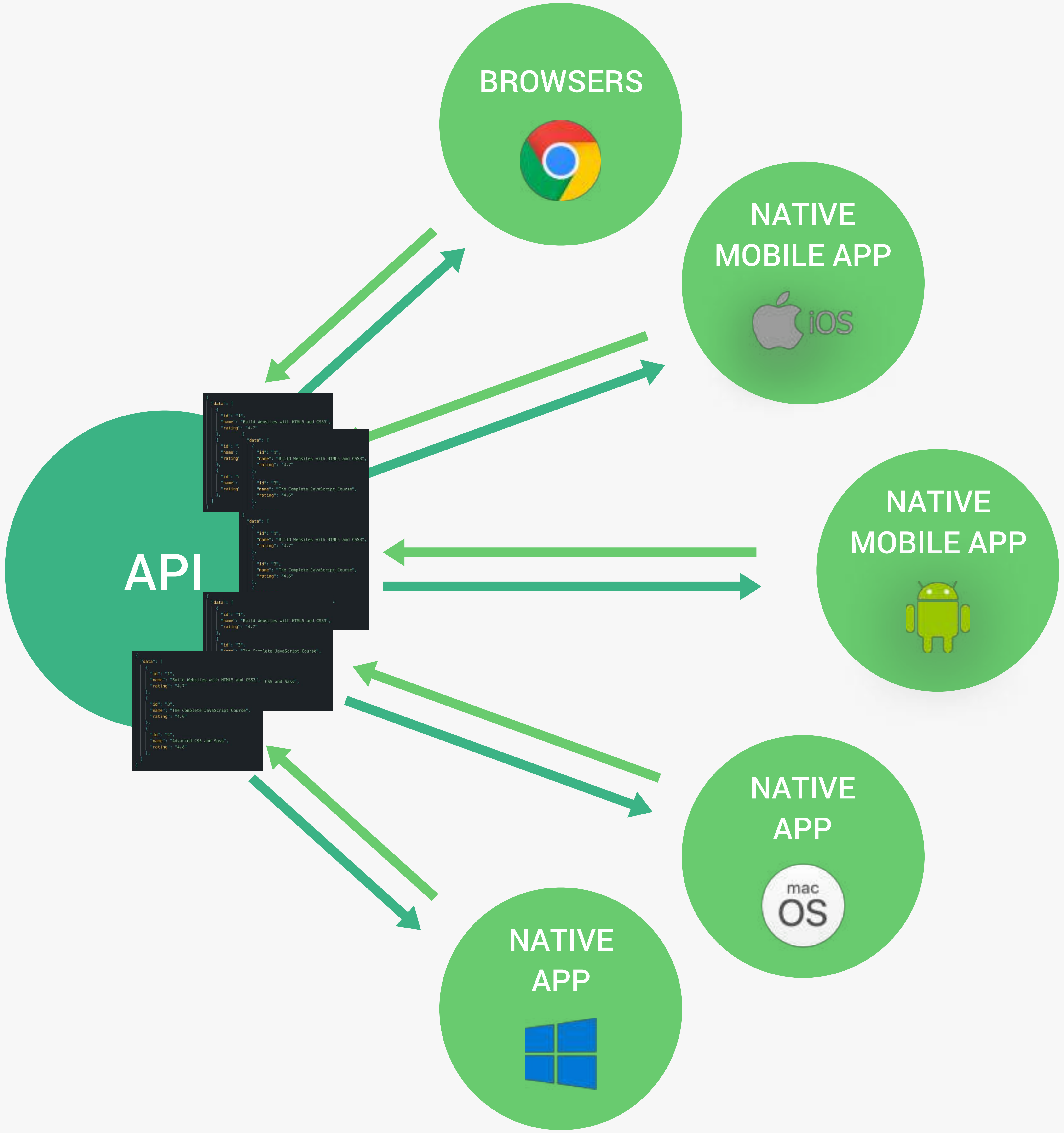
TEMPLATE

ONE API, MANY CONSUMERS

https://www.jonas.io/api/myCourseData



```
{
  "data": [
    {
      "id": "1",
      "name": "Build Websites with HTML5 and CSS3",
      "rating": "4.7"
    },
    {
      "id": "3",
      "name": "The Complete JavaScript Course",
      "rating": "4.6"
    },
    {
      "id": "4",
      "name": "Advanced CSS and Sass",
      "rating": "4.8"
    },
  ]
}
```



SECTION 4 — HOW NODE.JS WORKS: A LOOK BEHIND THE SCENES

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

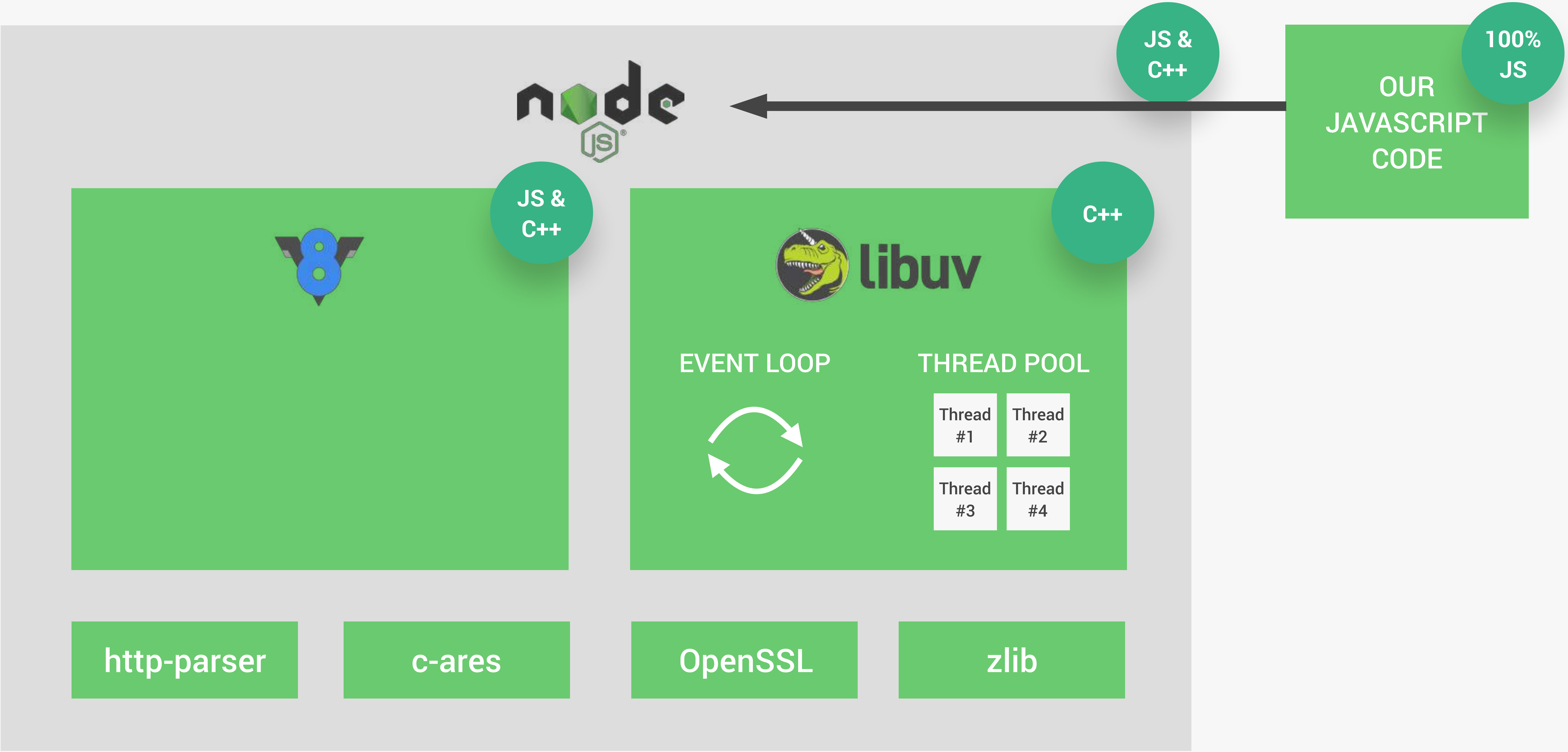
SECTION

HOW NODE.JS WORKS: A LOOK BEHIND THE SCENES

LECTURE

NODE, V8, LIBUV AND C++

THE NODE.JS ARCHITECTURE BEHIND THE SCENES



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

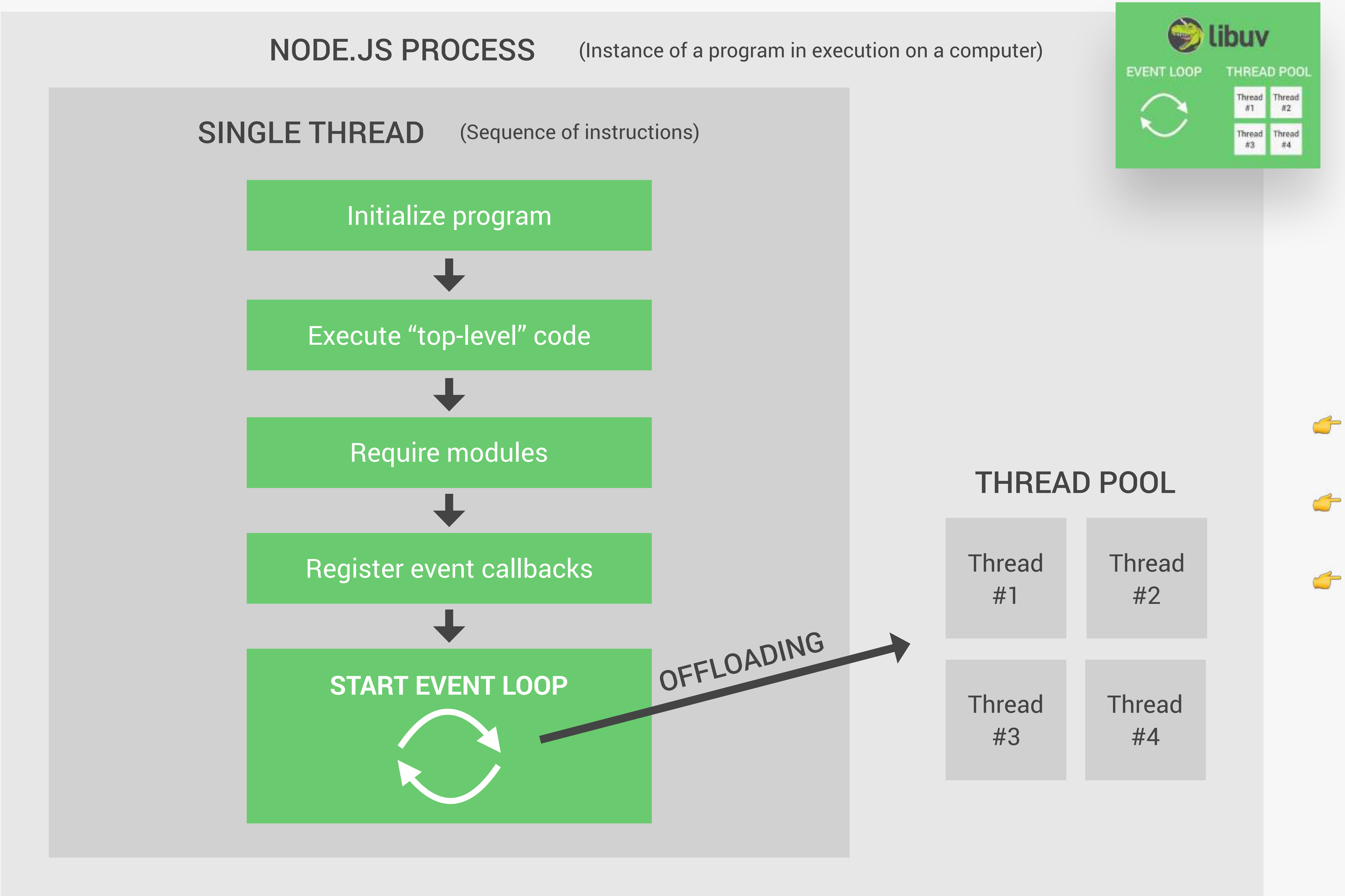
SECTION

HOW NODE.JS WORKS: A LOOK BEHIND THE SCENES

LECTURE

PROCESSES, THREADS AND THE THREAD POOL

NODE PROCESS AND THREADS



NODE.JS PROCESS (Instance of a program in execution on a computer)

SINGLE THREAD (Sequence of instructions)

Initialize program

Execute “top-level” code

Require modules

Register event callbacks

START EVENT LOOP

OFFLOADING

THREAD POOL

Thread #1

Thread #2

Thread #3

Thread #4

THREAD POOL:

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

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

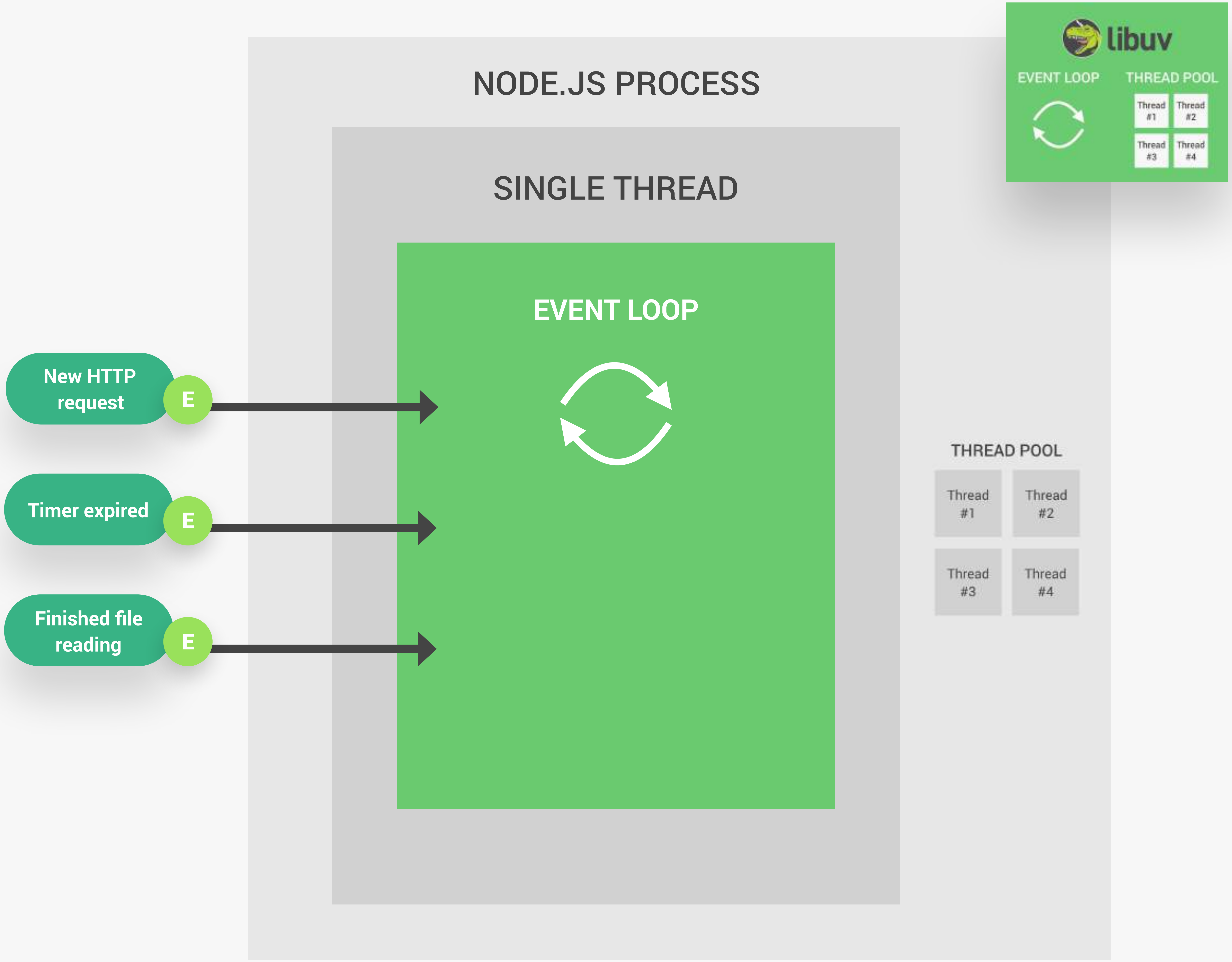
SECTION

HOW NODE.JS WORKS: A LOOK BEHIND THE SCENES

LECTURE

THE NODE.JS EVENT LOOP

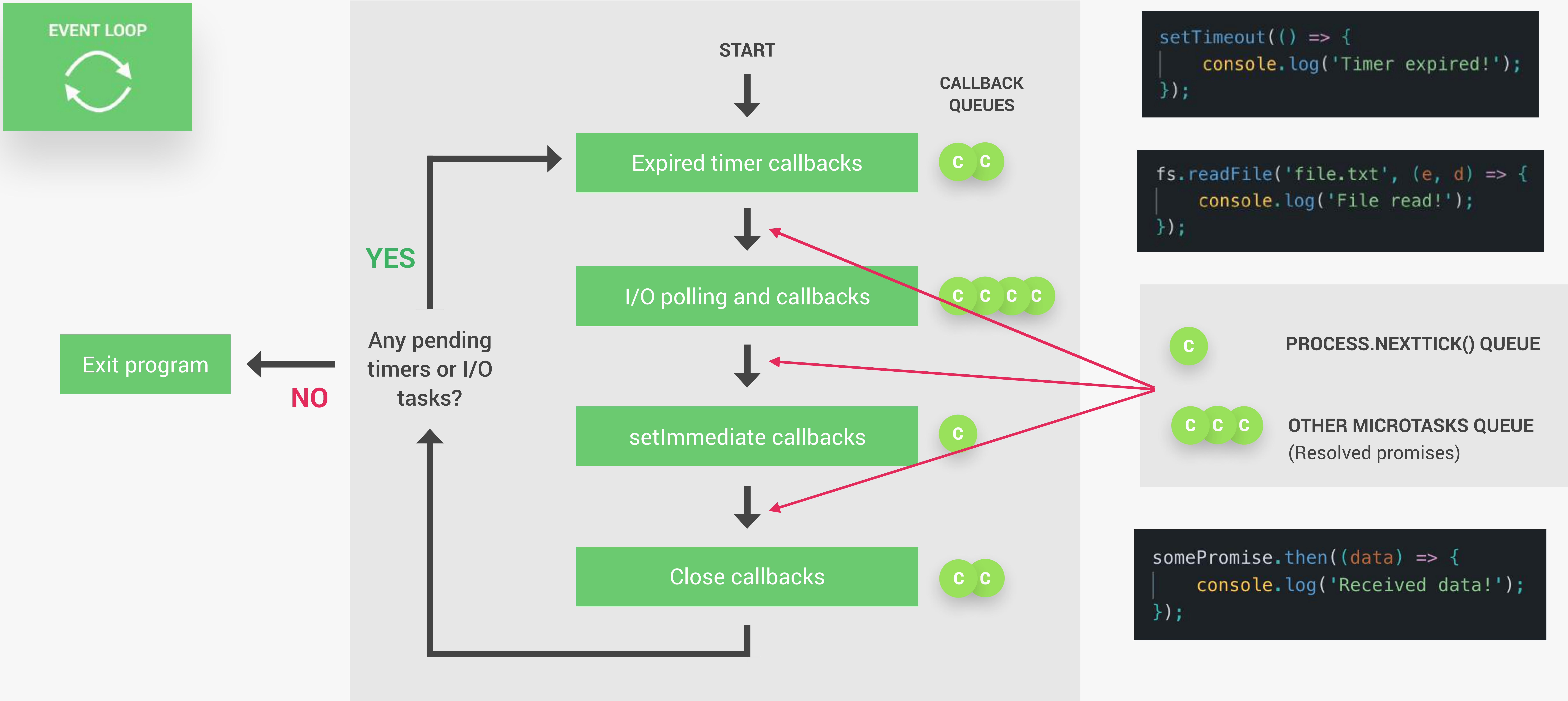
THE HEART OF NODE.JS: THE EVENT LOOP



EVENT LOOP:

- All the application code that is inside **callback functions** (non-top-top-level code)
- Node.js is build around callback functions
- Event-driven architecture:
  - Events are emitted
  - Event loops picks them up
  - Callbacks are called
- Event loop does **orchestration**

THE EVENT LOOP IN DETAIL

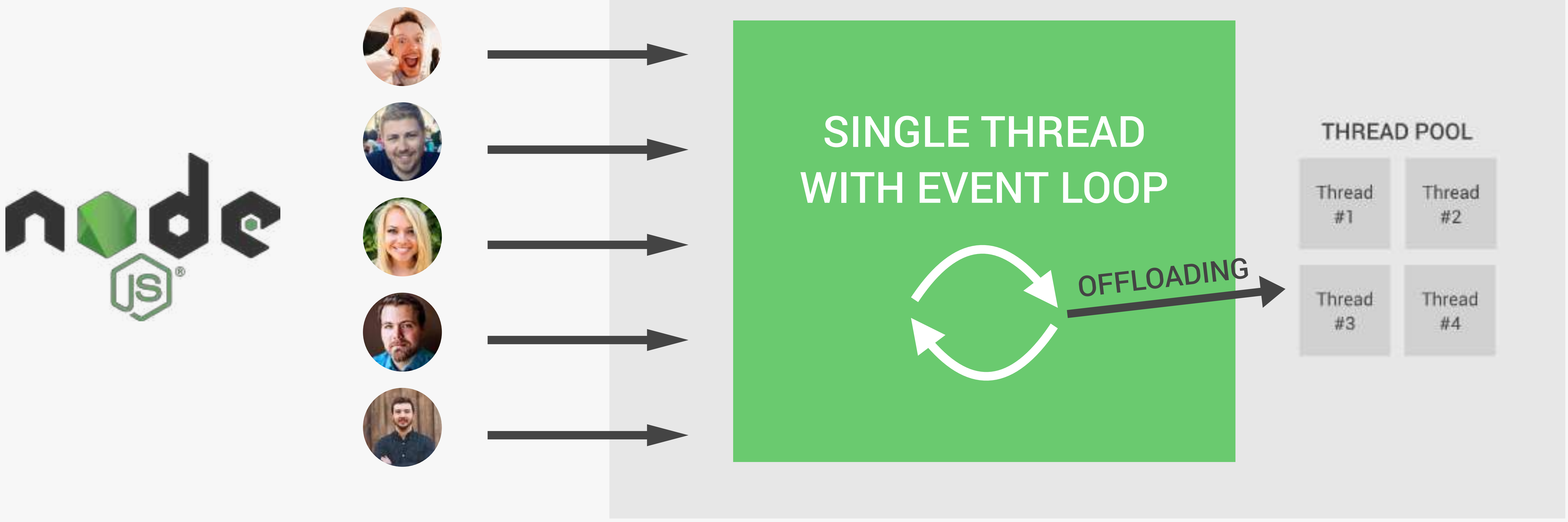


```
setTimeout(() => {
    console.log('Timer expired!');
});
```

```
fs.readFile('file.txt', (e, d) => {
    console.log('File read!');
});
```

```
somePromise.then((data) => {
    console.log('Received data!');
});
```

SUMMARY OF THE EVENT LOOP: NODE VS. OTHERS



SINGLE THREAD WITH EVENT LOOP

OFFLOADING

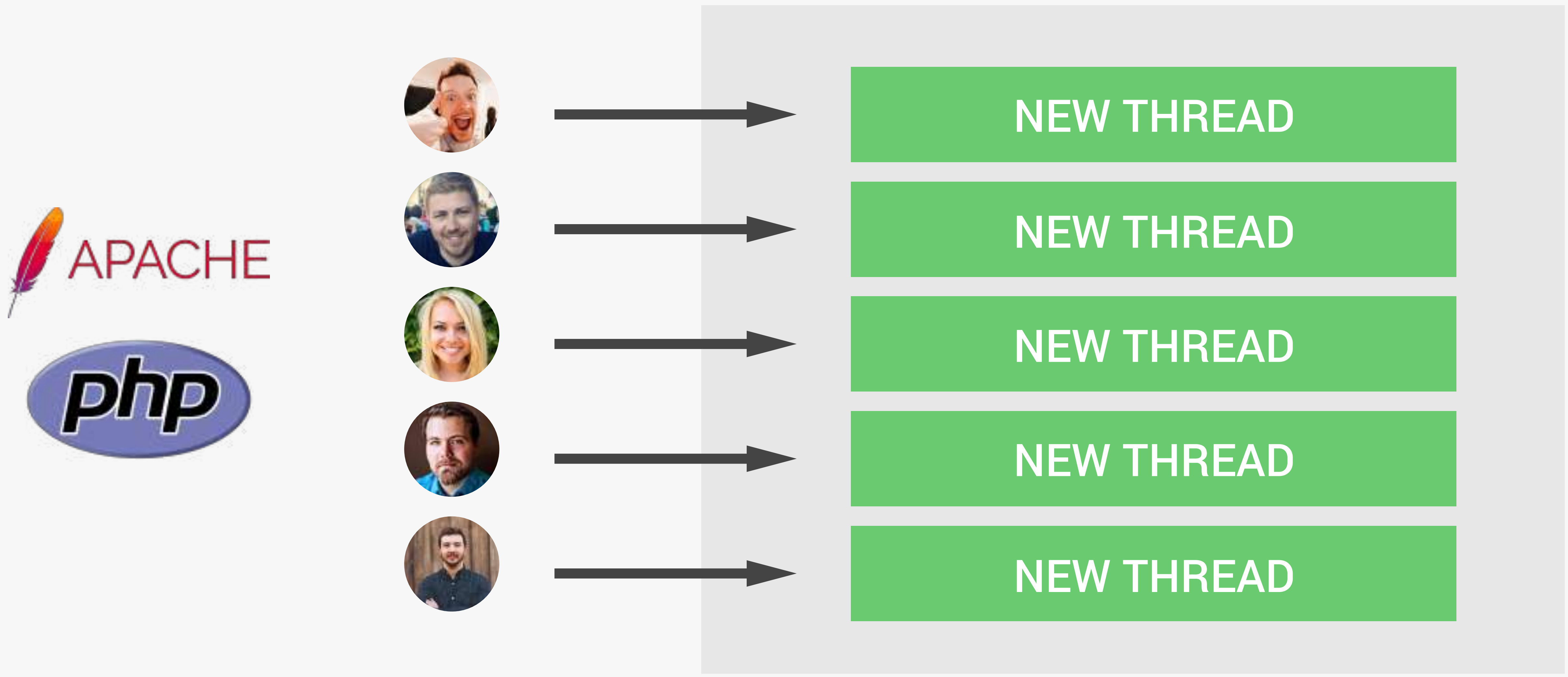
THREAD POOL

Thread #1

Thread #2

Thread #3

Thread #4



NEW THREAD

NEW THREAD

NEW THREAD

NEW THREAD

NEW THREAD

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)

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

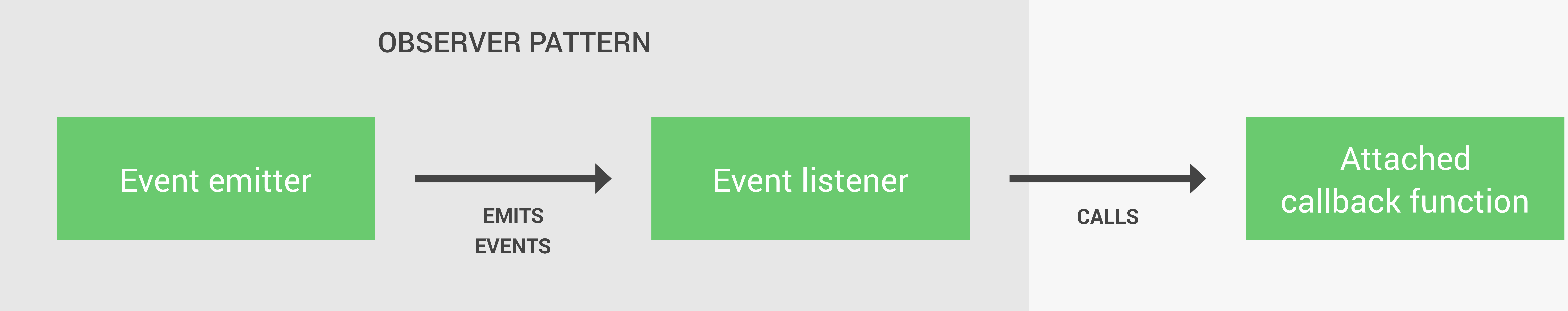
SECTION

HOW NODE.JS WORKS: A LOOK BEHIND THE SCENES

LECTURE

EVENTS AND EVENT-DRIVEN ARCHITECTURE

THE EVENT-DRIVEN ARCHITECTURE



OBSERVER PATTERN

Event emitter

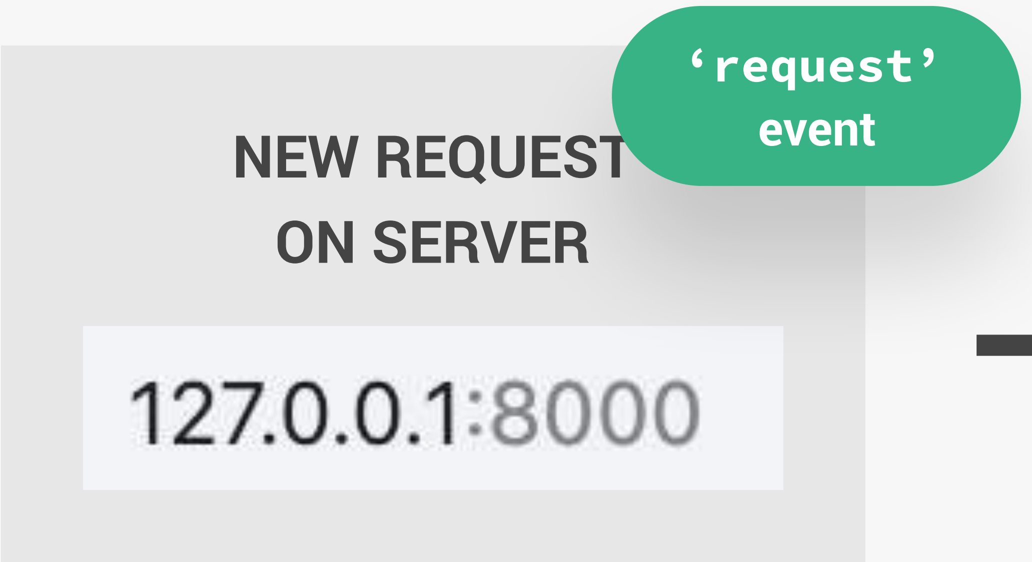
EMITS EVENTS

Event listener

CALLS

Attached callback function

EMITTER

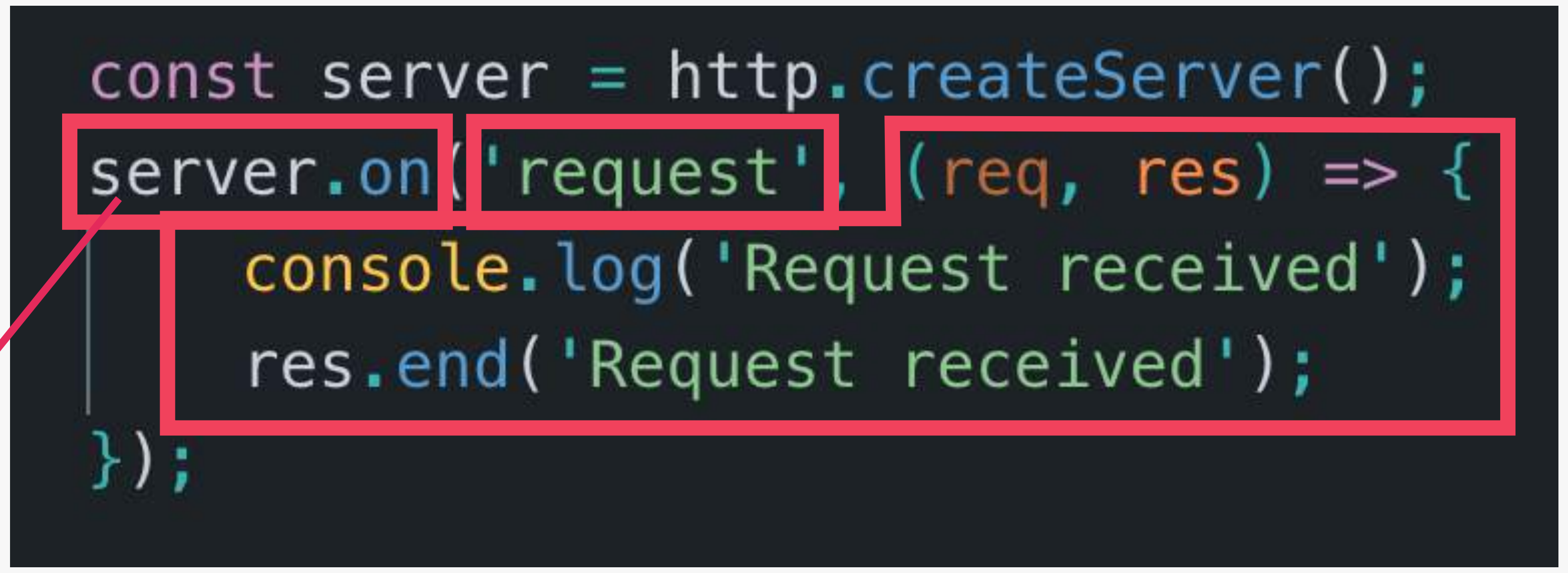


‘request’ event

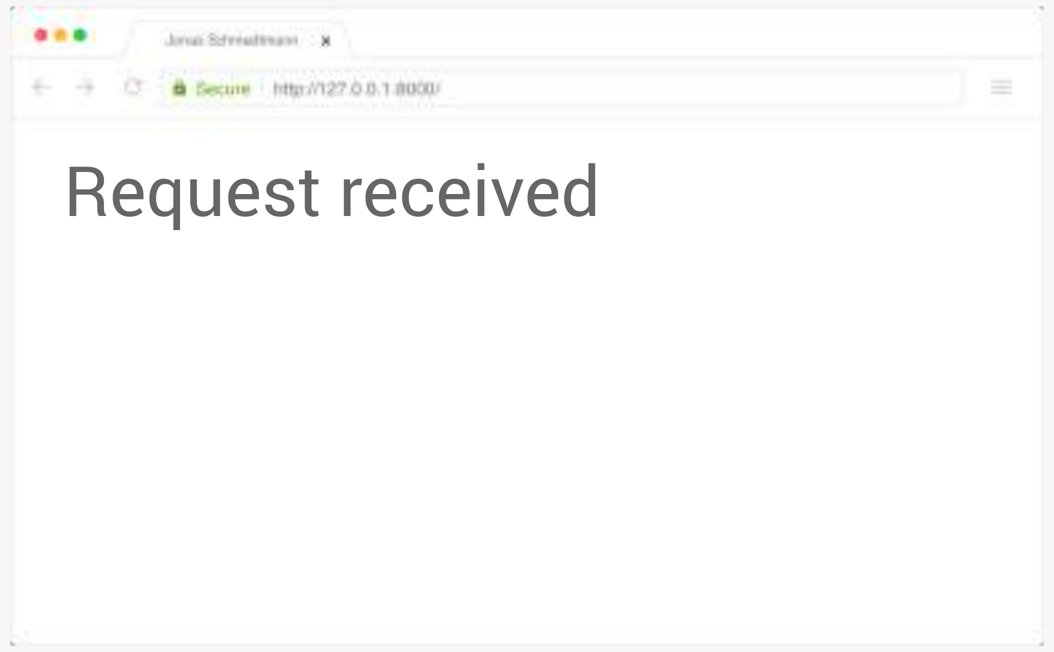
NEW REQUEST ON SERVER

127.0.0.1:8000

LISTENER



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



Request received

👉 Instance of `EventEmitter` class

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

HOW NODE.JS WORKS: A LOOK BEHIND THE SCENES

LECTURE

INTRODUCTION TO STREAMS

WHAT ARE STREAMS?

STREAMS

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	CONSUME STREAMS	
TRANSFORM STREAMS	Duplex streams that transform data as it is written or read	👉 `zlib` Gzip creation		

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

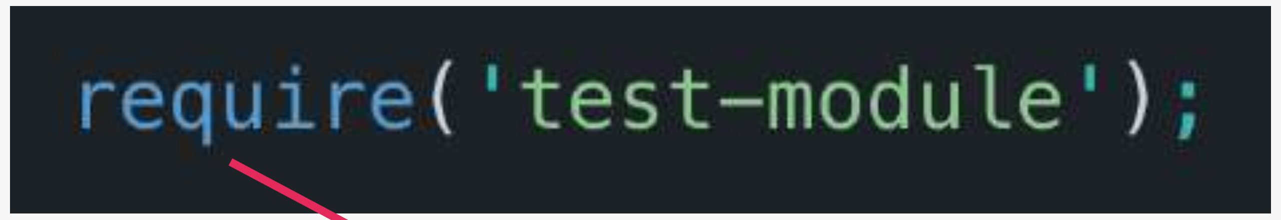
HOW NODE.JS WORKS: A LOOK BEHIND THE SCENES

LECTURE

HOW REQUIRING MODULES REALLY WORKS

THE COMMONJS MODULE SYSTEM

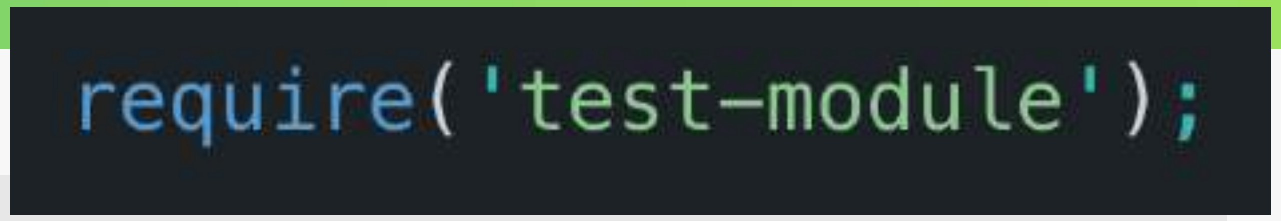
- Each JavaScript file is treated as a separate module;
- Node.js uses the **CommonJS module system**: `require()`, `exports` or `module.exports`;
- **ES module system** is used in browsers: `import/export`;
- There have been attempts to bring ES modules to node.js (`.mjs`).



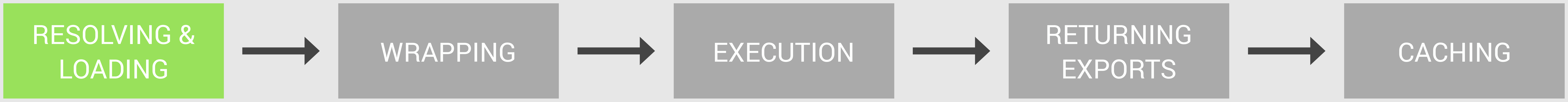
```
require('test-module');
```

Where does it come from?

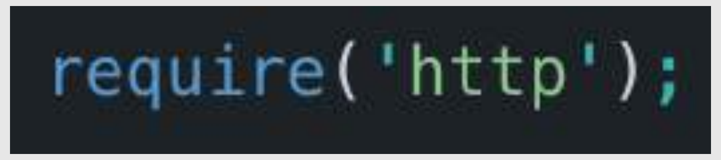
WHAT HAPPENS WHEN WE REQUIRE() A MODULE



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

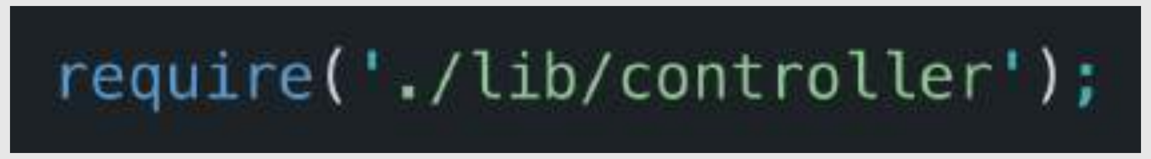


👉 **Core modules**



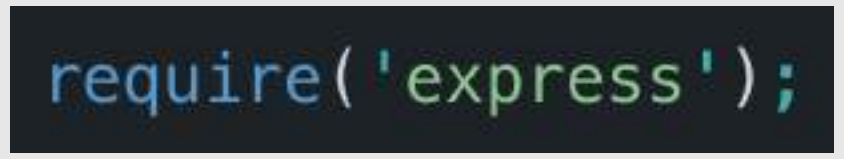
```
require('http');
```

👉 **Developer modules**



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

👉 **3rd-party modules (from NPM)**



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

PATH RESOLVING: HOW NODE DECIDES WHICH MODULE TO LOAD

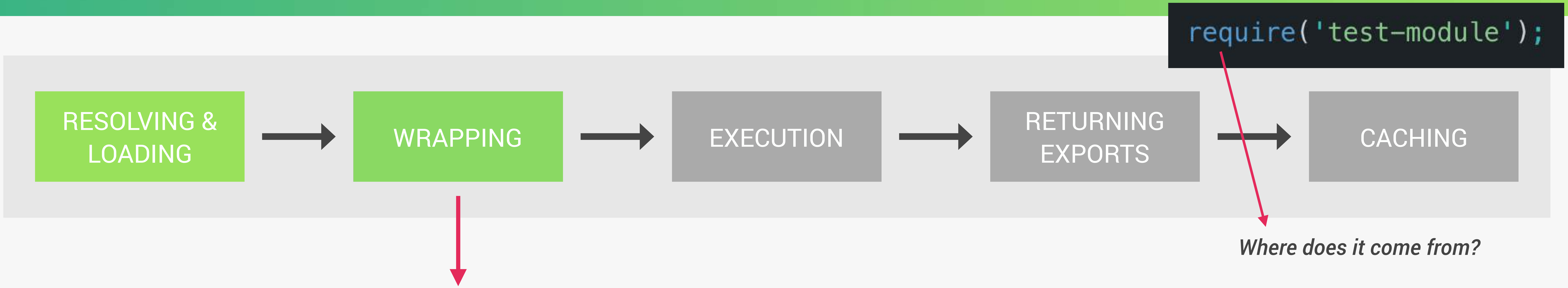
1️⃣ Start with **core modules**;

2️⃣ If begins with `‘./‘` or `‘../‘` 👉 Try to **load developer module**;

3️⃣ If no file found 👉 Try to **find folder** with `index.js` in it;

4️⃣ Else 👉 Go to **`node_modules/`** and try to find module there.

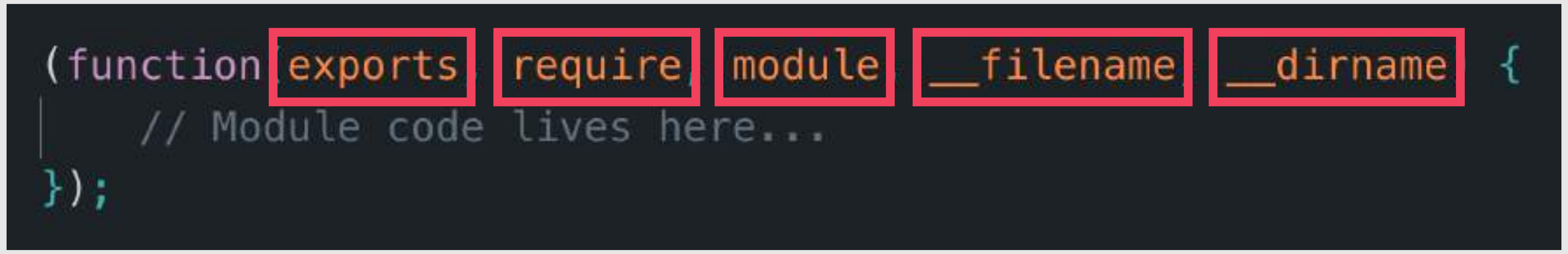
WHAT HAPPENS WHEN WE REQUIRE() A MODULE



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

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

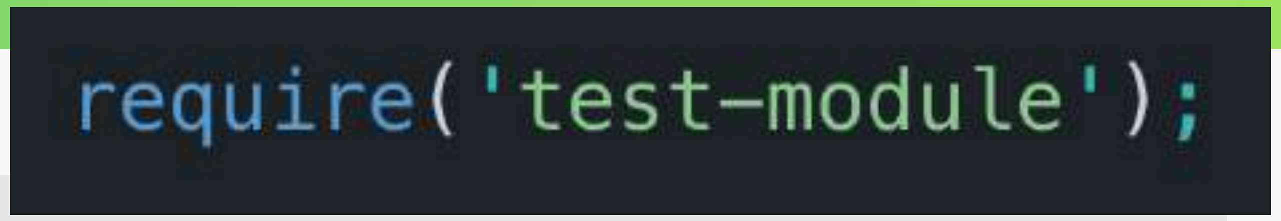
Where does it come from?



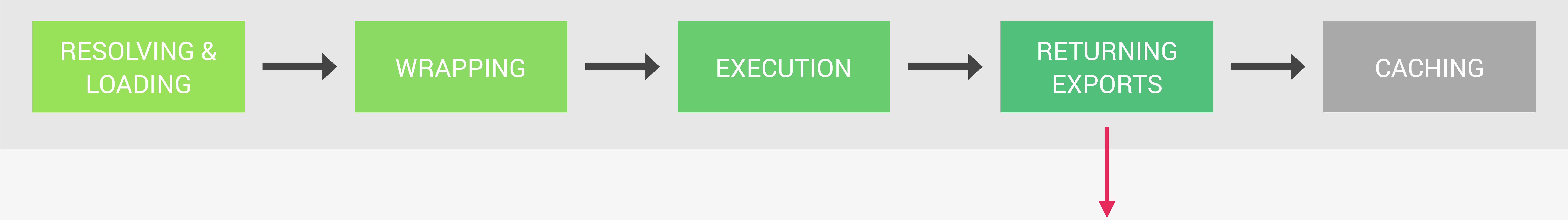
```
(function(exports, require, module, __filename, __dirname) {
    // Module code lives here...
});
```

- **`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.

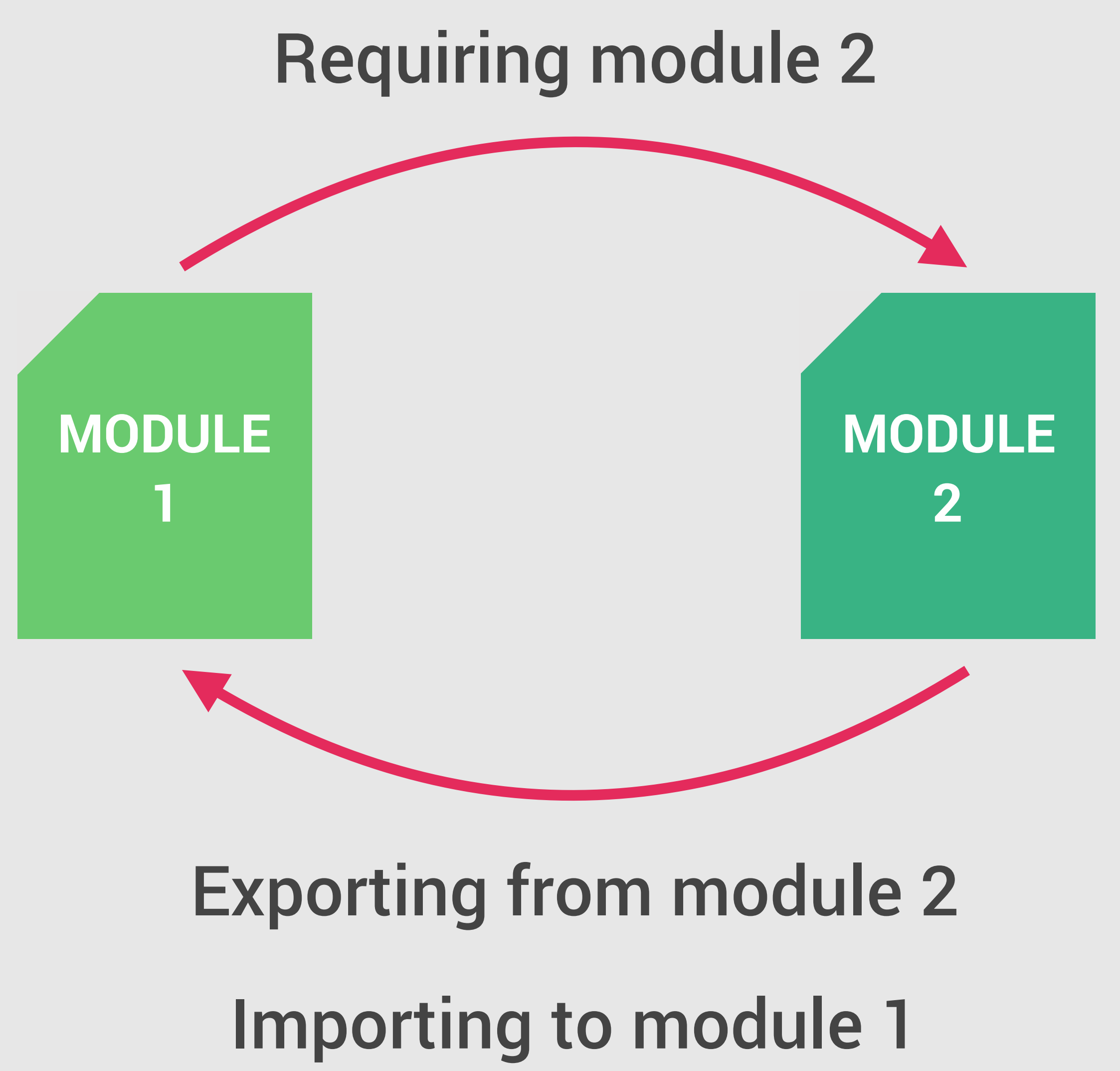
WHAT HAPPENS WHEN WE REQUIRE() A MODULE



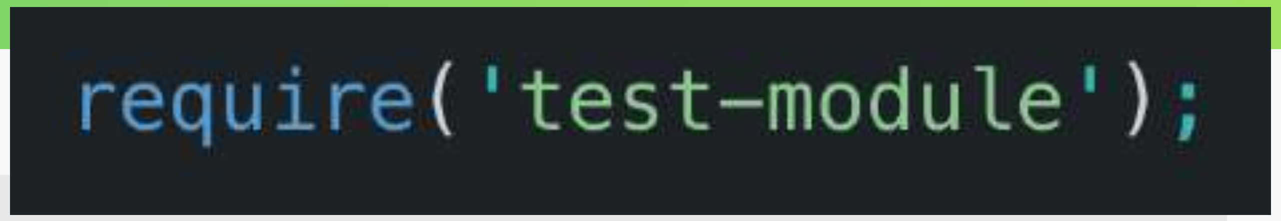
```
require('test-module');
```



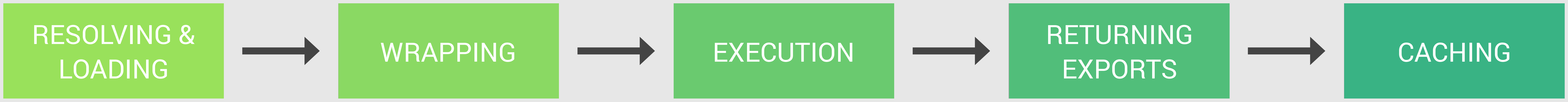
- `require` function returns **exports** of the required module;
- `module.exports` is the returned object (important!);
- Use `module.exports` to export one single variable, e.g. one class or one function (`module.exports = Calculator`);
- Use `exports` to export multiple named variables (`exports.add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



WHAT HAPPENS WHEN WE REQUIRE() A MODULE



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



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

SECTION 6 — EXPRESS: LET'S START BUILDING THE NATOURS API!

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

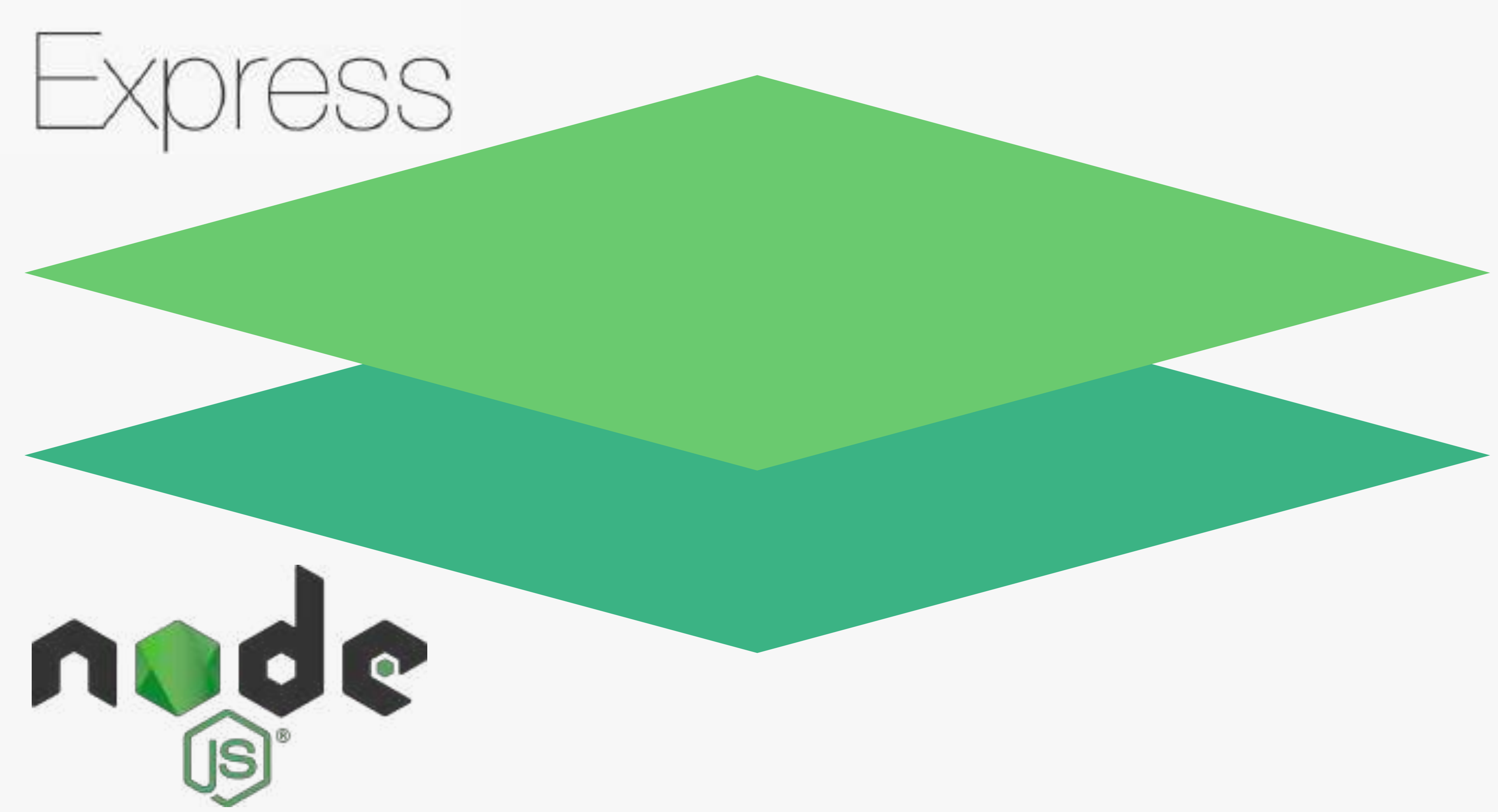
SECTION

EXPRESS: LET'S START BUILDING THE NATOURS API!

LECTURE

WHAT IS EXPRESS?

WHAT IS EXPRESS, AND WHY USE IT?



- Express is a minimal node.js framework, a higher level of abstraction;
- Express contains a very robust set of features: **complex routing, easier handling of requests and responses, middleware, server-side rendering**, etc.;
- Express allows for rapid development of node.js applications: *we don’t have to re-invent the wheel*;
- Express makes it easier to organize our application into the MVC architecture.

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

EXPRESS: LET'S START BUILDING THE NATOURS API!

LECTURE

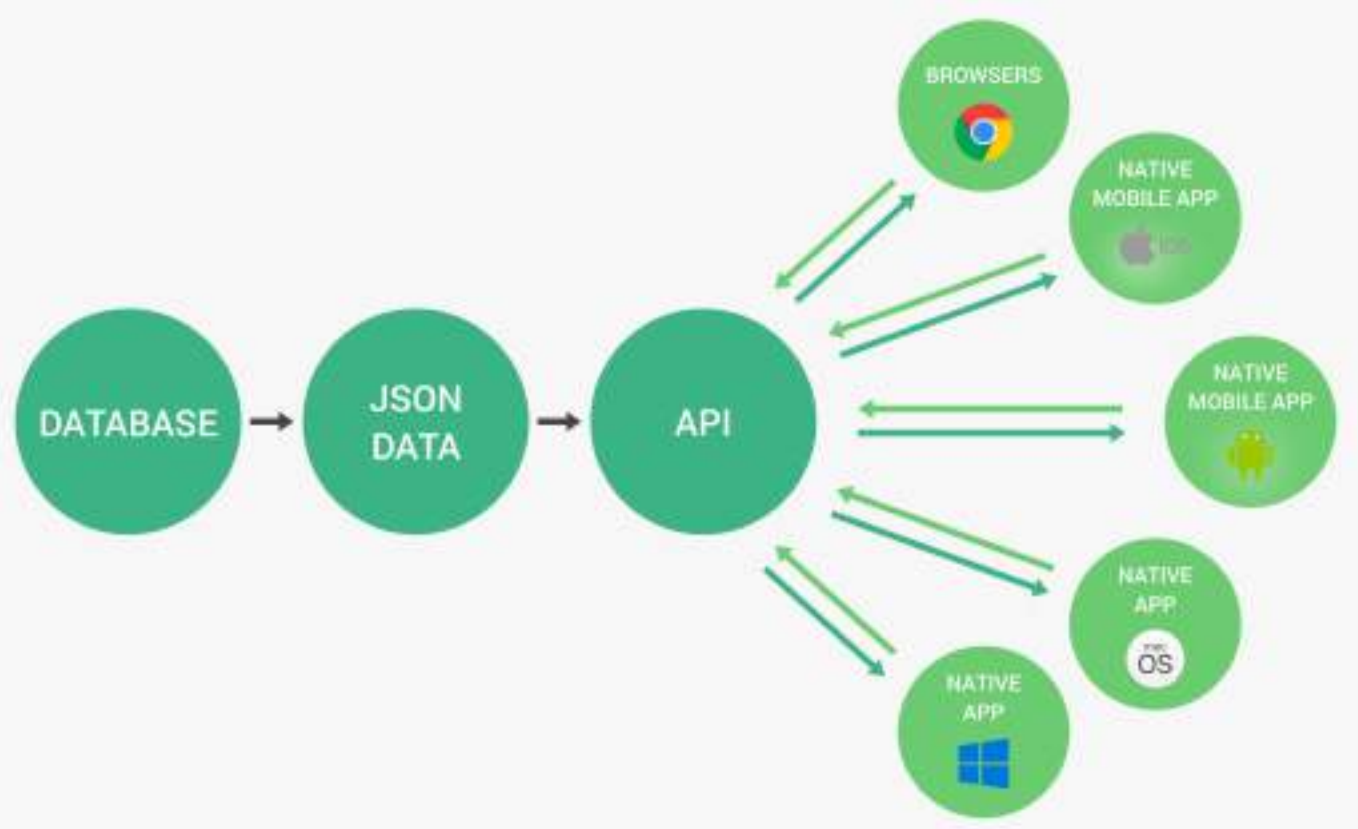
APIS AND RESTFUL API DESIGN

WHAT IS AN API ANYWAY?

API

Application **P**rogramming **I**nterface: a piece of software that can be used by another piece of software, in order to allow applications to talk to each other.

👉 **Web APIs**



👉 **But, "Application" can be other things:**

- 👉 Node.js' `fs` or `http` APIs ("node APIs");
- 👉 Browser's DOM JavaScript API;
- 👉 With object-oriented programming, when exposing methods to the public, we're creating an API;
- 👉 ...

THE REST ARCHITECTURE

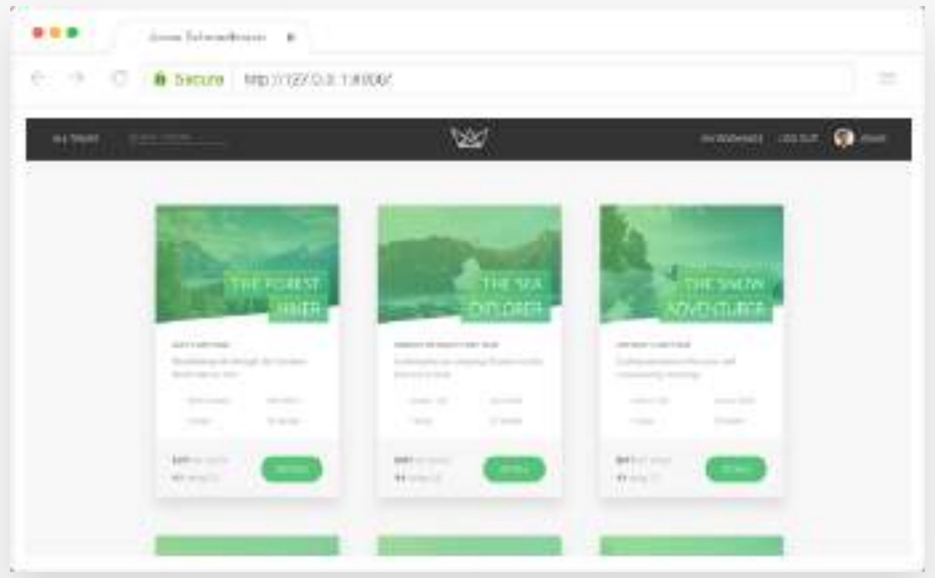
1. Separate API into logical **resources**
2. Expose structured, **resource-based URLs**
3. Use **HTTP methods** (verbs)
4. Send data as **JSON** (usually)
5. Be **stateless**

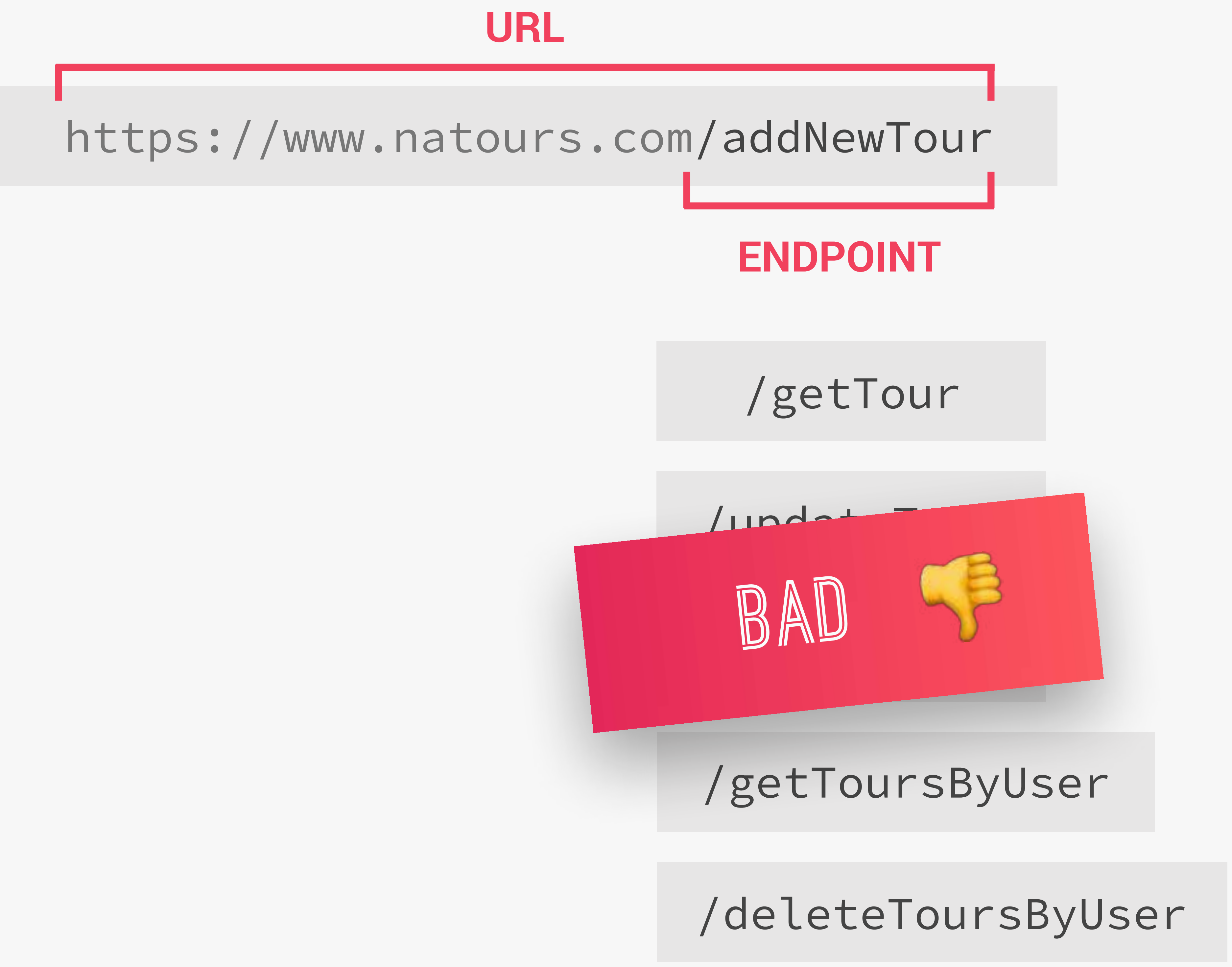
THE REST ARCHITECTURE

1. Separate API into logical **resources**
2. Expose structured, **resource-based URLs**
3. Use **HTTP methods** (verbs)
4. Send data as **JSON** (usually)
5. Be **stateless**

👉 **Resource:** Object or representation of something, which has data associated to it. Any information that can be **named** can be a resource.

`tours` `users` `reviews`





URL

```
https://www.natours.com/addNewTour
```

ENDPOINT

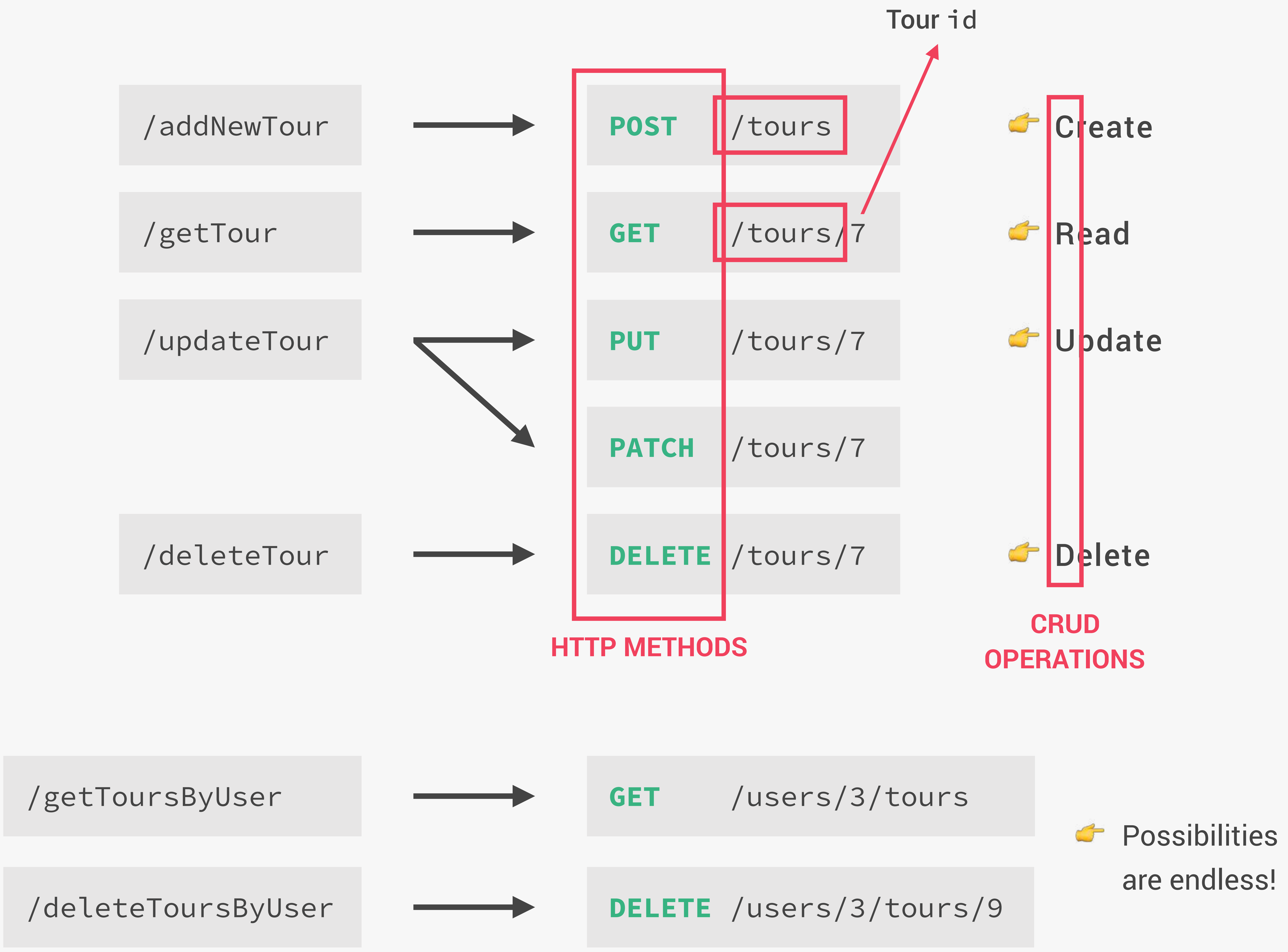
```
/getTour
/updat T
/getToursByUser
/deleteToursByUser
```

BAD 👎

👉 Endpoints should contain **only resources** (nouns), and use **HTTP methods** for actions!

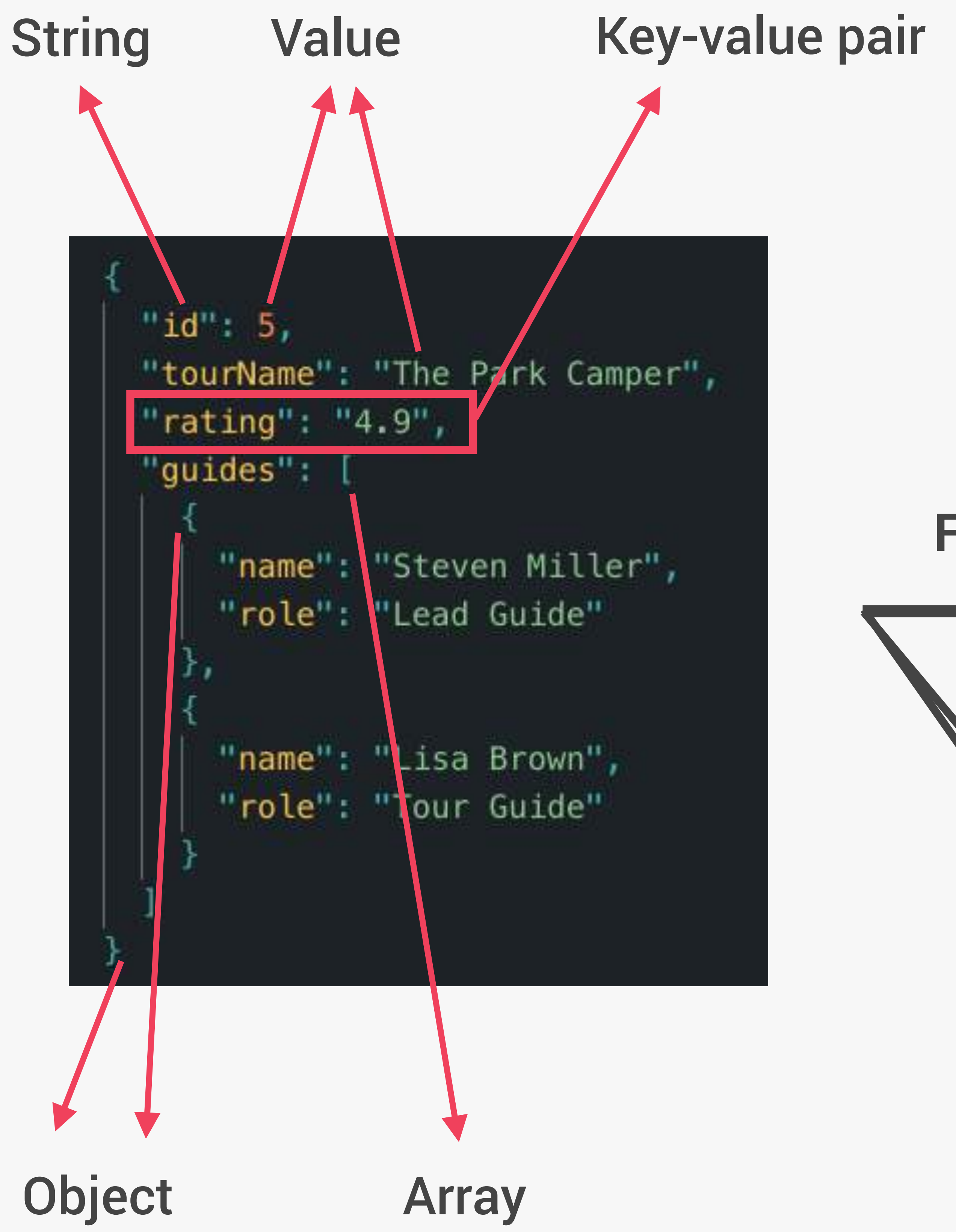
THE REST ARCHITECTURE

1. Separate API into logical **resources**
2. Expose structured, **resource-based URLs**
3. Use **HTTP methods** (verbs)
4. Send data as **JSON** (usually)
5. Be **stateless**



THE REST ARCHITECTURE

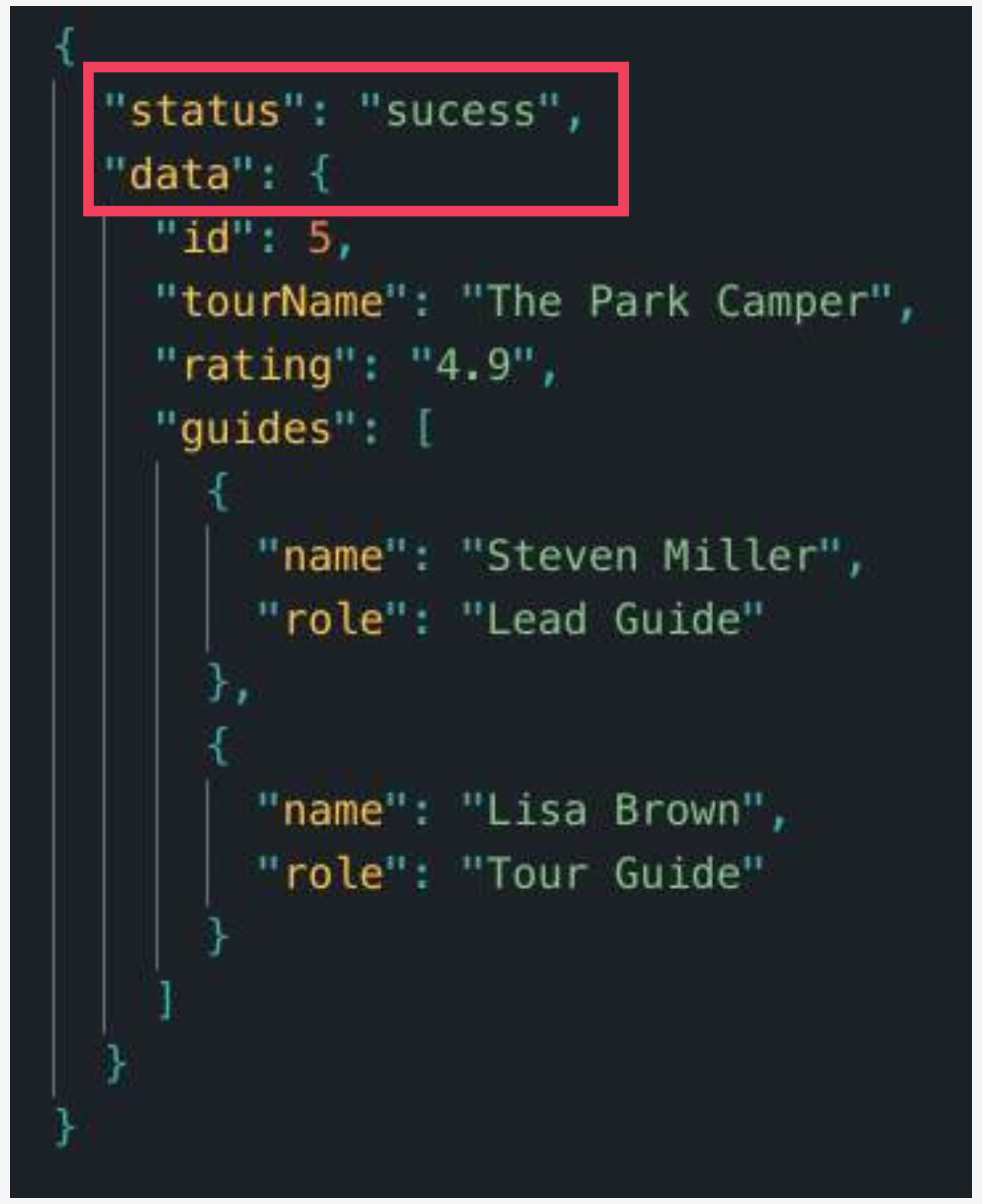
1. Separate API into logical **resources**
2. Expose structured, **resource-based URLs**
3. Use **HTTP methods** (verbs)
4. Send data as **JSON** (usually)
5. Be **stateless**



```
{
  "id": 5,
  "tourName": "The Park Camper",
  "rating": "4.9",
  "guides": [
    {
      "name": "Steven Miller",
      "role": "Lead Guide"
    },
    {
      "name": "Lisa Brown",
      "role": "Tour Guide"
    }
  ]
}
```

RESPONSE FORMATTING

- **JSend**



```
{
  "status": "sucess",
  "data": {
    "id": 5,
    "tourName": "The Park Camper",
    "rating": "4.9",
    "guides": [
      {
        "name": "Steven Miller",
        "role": "Lead Guide"
      },
      {
        "name": "Lisa Brown",
        "role": "Tour Guide"
      }
    ]
  }
}
```

- **JSON:API**
- **OData JSON Protocol**
- **...**

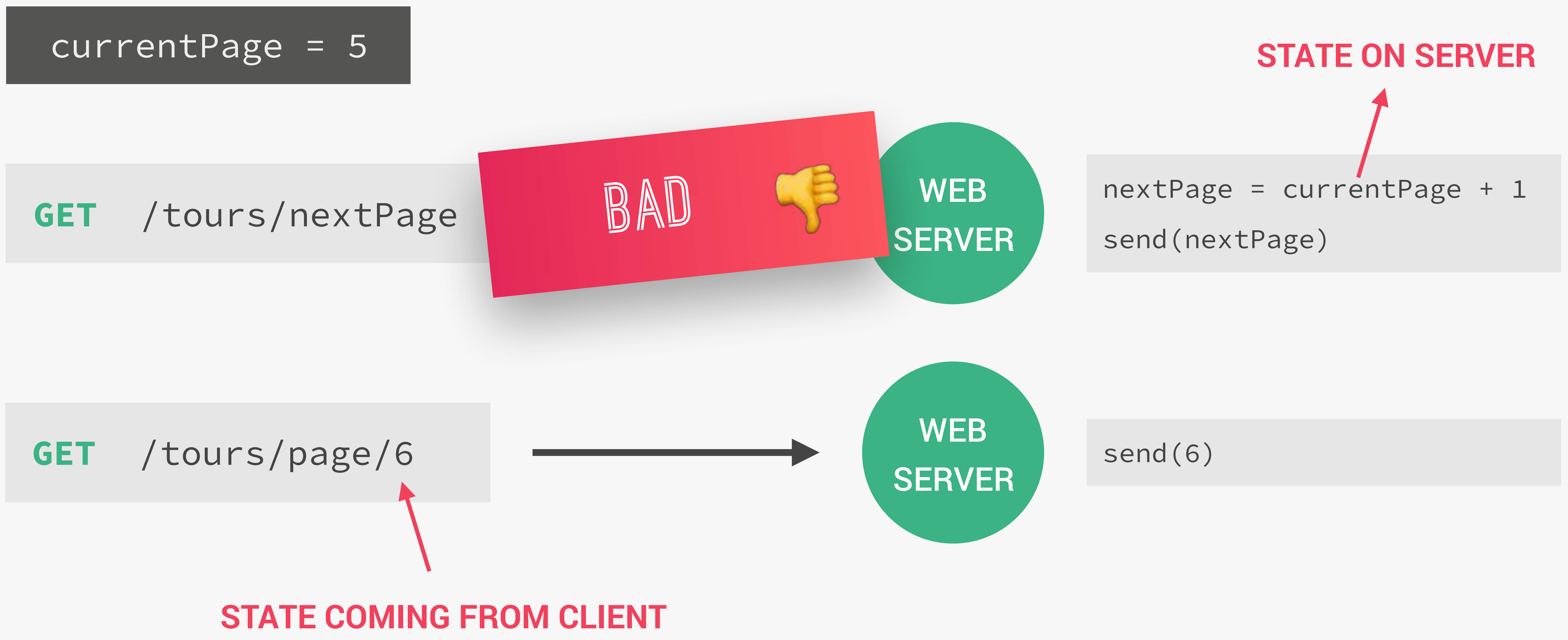
https://www.natours.com/tours/5

THE REST ARCHITECTURE

1. Separate API into logical **resources**
2. Expose structured, **resource-based URLs**
3. Use **HTTP methods** (verbs)
4. Send data as **JSON** (usually)
5. Be **stateless**

- **Stateless RESTful API:** All state is handled **on the client**. This means that each request must contain **all** the information necessary to process a certain request. The server should **not** have to remember previous requests.

- **Examples of state:** `loggedIn` `currentPage`



```
currentPage = 5
```

```
GET /tours/nextPage
```

```
nextPage = currentPage + 1
send(nextPage)
```

```
GET /tours/page/6
```

```
send(6)
```

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

EXPRESS: LET'S START BUILDING THE NATOURS API!

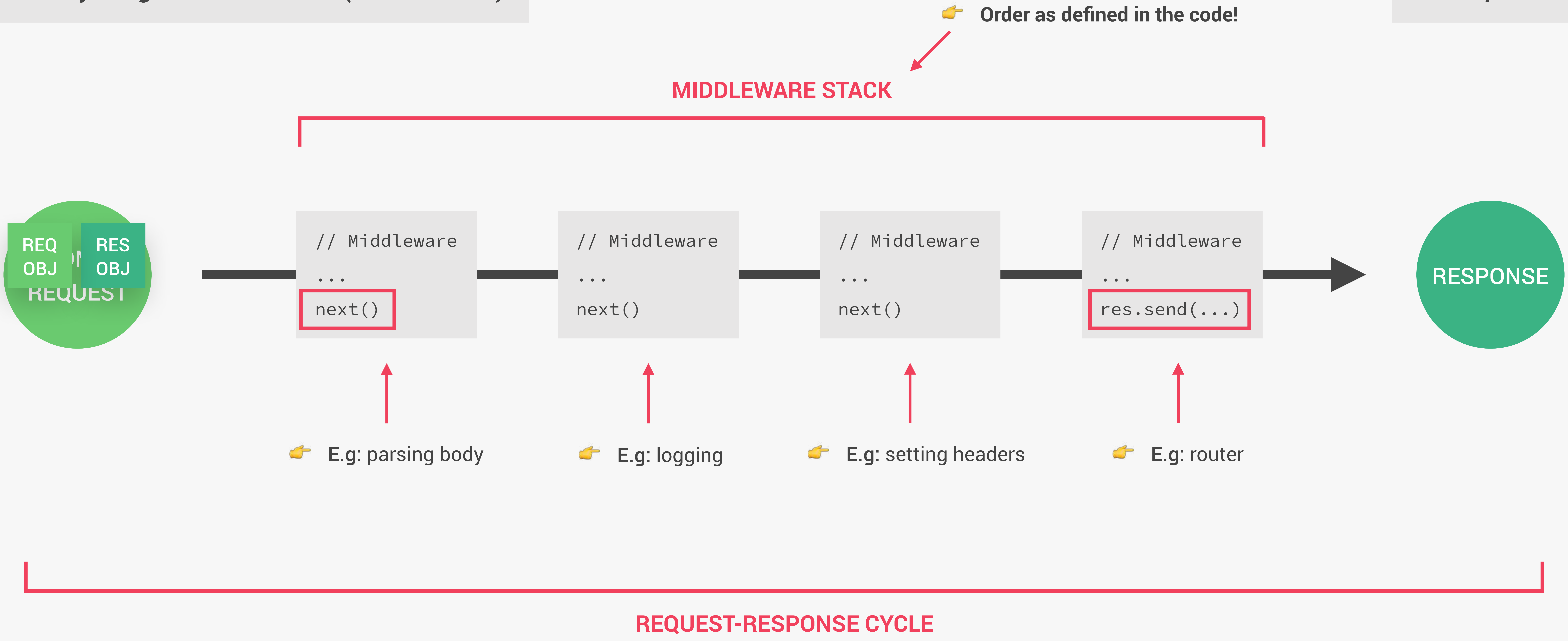
LECTURE

MIDDLEWARE AND THE REQUEST-RESPONSE CYCLE

THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE

👉 *“Everything is middleware” (even routers)*

👉 *“Pipeline”*



👉 **Order as defined in the code!**

MIDDLEWARE STACK

```
// Middleware
...
next()
```

```
// Middleware
...
next()
```

```
// Middleware
...
next()
```

```
// Middleware
...
res.send(...)
```

👉 E.g: parsing body

👉 E.g: logging

👉 E.g: setting headers

👉 E.g: router

REQUEST-RESPONSE CYCLE

SECTION 7 — INTRODUCTION TO MONGODB

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

INTRODUCTION TO MONGODB

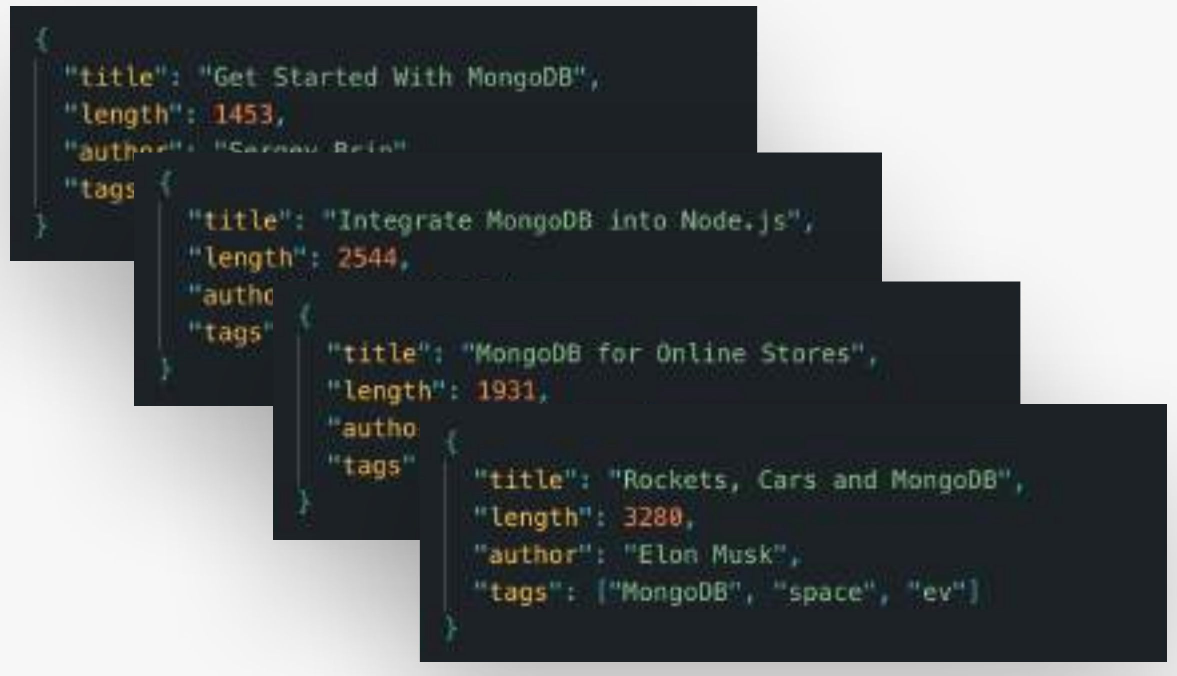
LECTURE

WHAT IS MONGODB?

MONGODB: AN OVERVIEW

>

>



```
{
  "title": "Rockets, Cars and MongoDB",
  "length": 3280,
  "author": "Elon Musk",
  "tags": ["MongoDB", "space", "ev"]
}
```

DATABASE

COLLECTIONS

("Tables")

DOCUMENTS

("Rows")

👉 NoSQL

blog → post

users → user

reviews → review

WHAT IS MONGODB?

MONGODB

> *"MongoDB is a document database with the scalability and flexibility that you want with the querying and indexing that you need"*

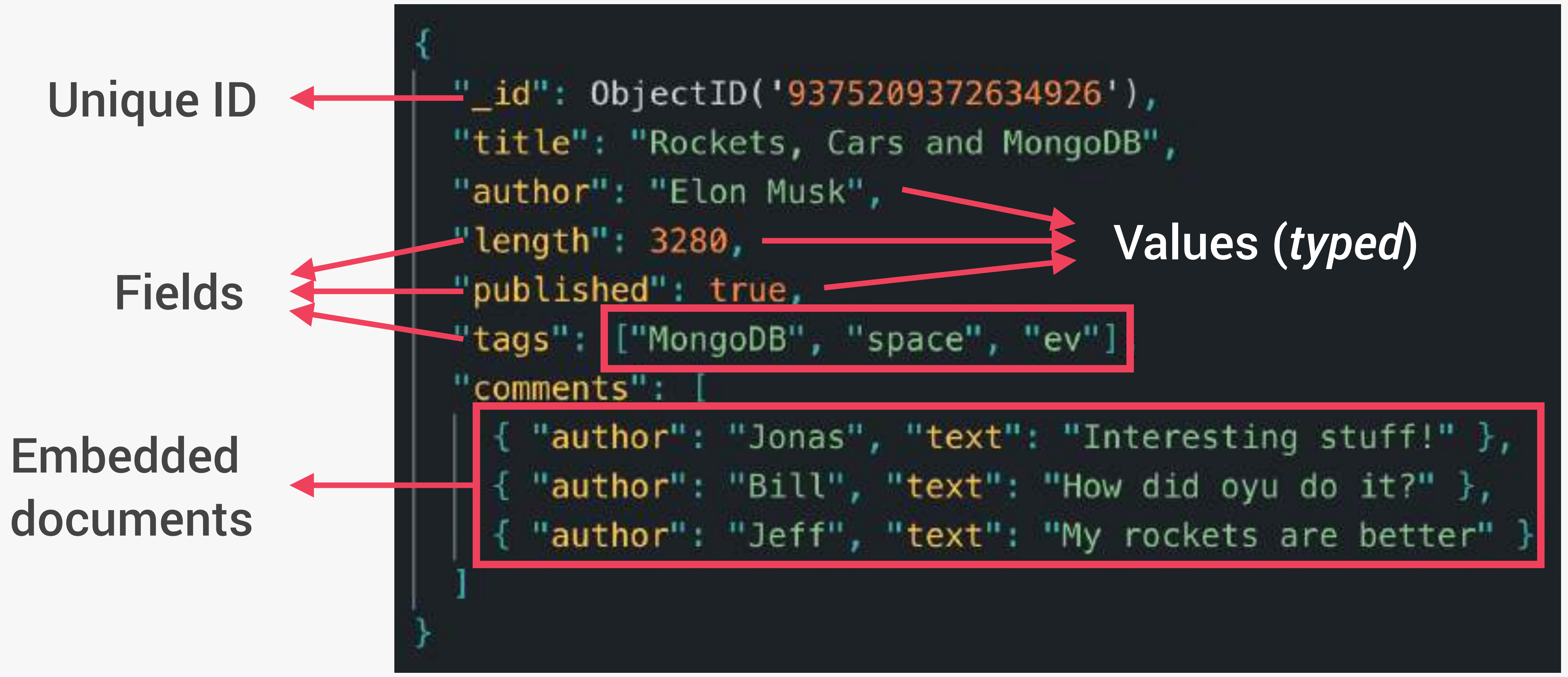
KEY MONGODB FEATURES:

- 👉 **Document based:** MongoDB stores data in documents (field-value pair data structures, NoSQL);
- 👉 **Scalable:** Very easy to distribute data across multiple machines as your users and amount of data grows;
- 👉 **Flexible:** No document data schema required, so each document can have different number and type of fields;
- 👉 **Performant:** Embedded data models, indexing, sharding, flexible documents, native duplication, etc.
- 👉 Free and open-source, published under the SSPL License.

DOCUMENTS, BSON AND EMBEDDING

DOCUMENT STRUCTURE

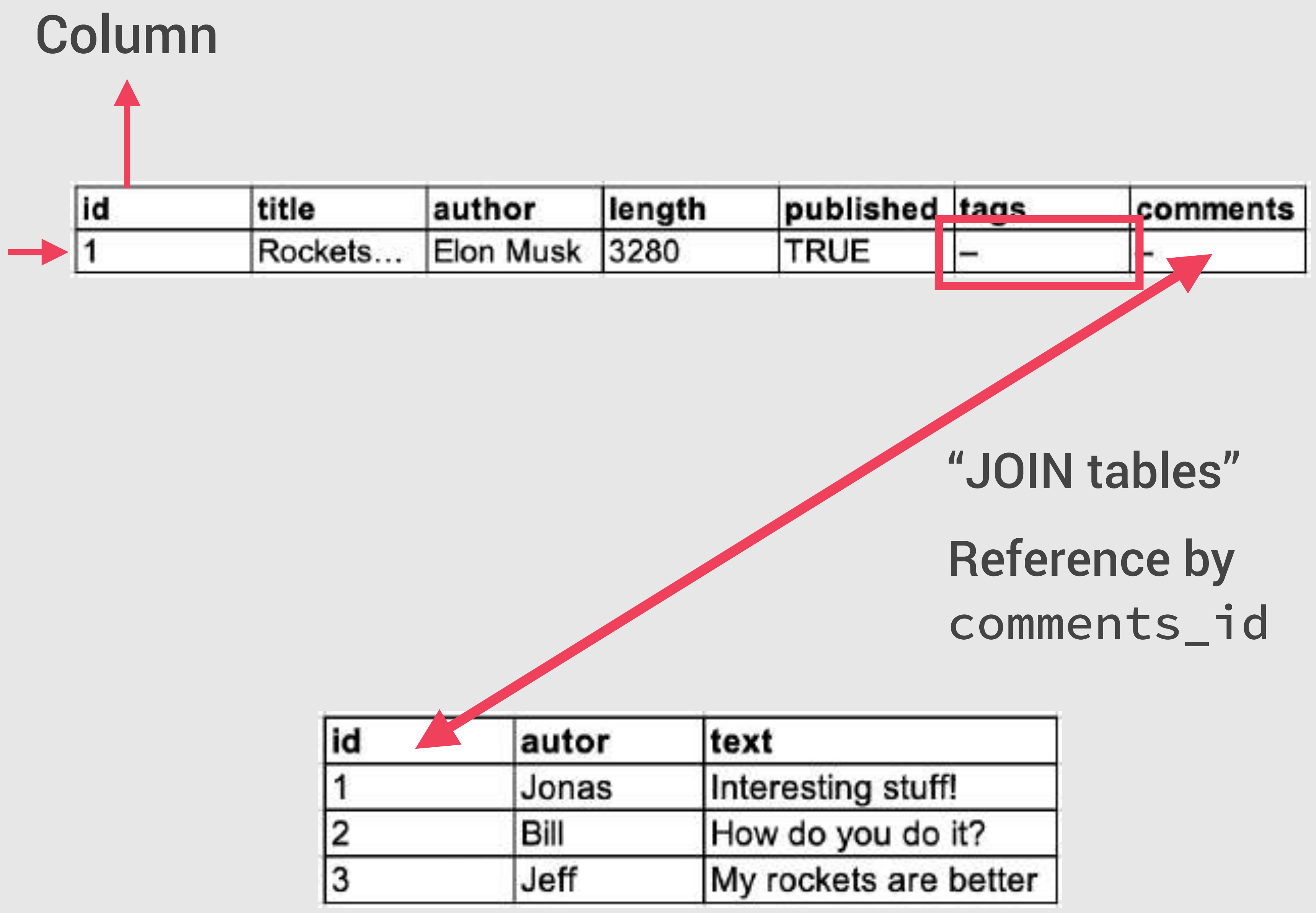
- 👉 **BSON:** Data format MongoDB uses for data storage. Like JSON, **but typed**. So MongoDB documents are typed.



```
{
  "_id": ObjectID('9375209372634926'),
  "title": "Rockets, Cars and MongoDB",
  "author": "Elon Musk",
  "length": 3280,
  "published": true,
  "tags": ["MongoDB", "space", "ev"]
  "comments": [
    { "author": "Jonas", "text": "Interesting stuff!" },
    { "author": "Bill", "text": "How did oyu do it?" },
    { "author": "Jeff", "text": "My rockets are better" }
  ]
}
```

- 👉 **Embedding/Denormalizing:** Including related data into a single document. This allows for quicker access and easier data models (it's not always the best solution though).

RELATIONAL DATABASE



id	title	author	length	published	tags	comments
1	Rockets…	Elon Musk	3280	TRUE	–	-

id	autor	text
1	Jonas	Interesting stuff!
2	Bill	How do you do it?
3	Jeff	My rockets are better

- 👉 **Data is always normalized**

SECTION 8 — USING MONGODB WITH MONGOOSE

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

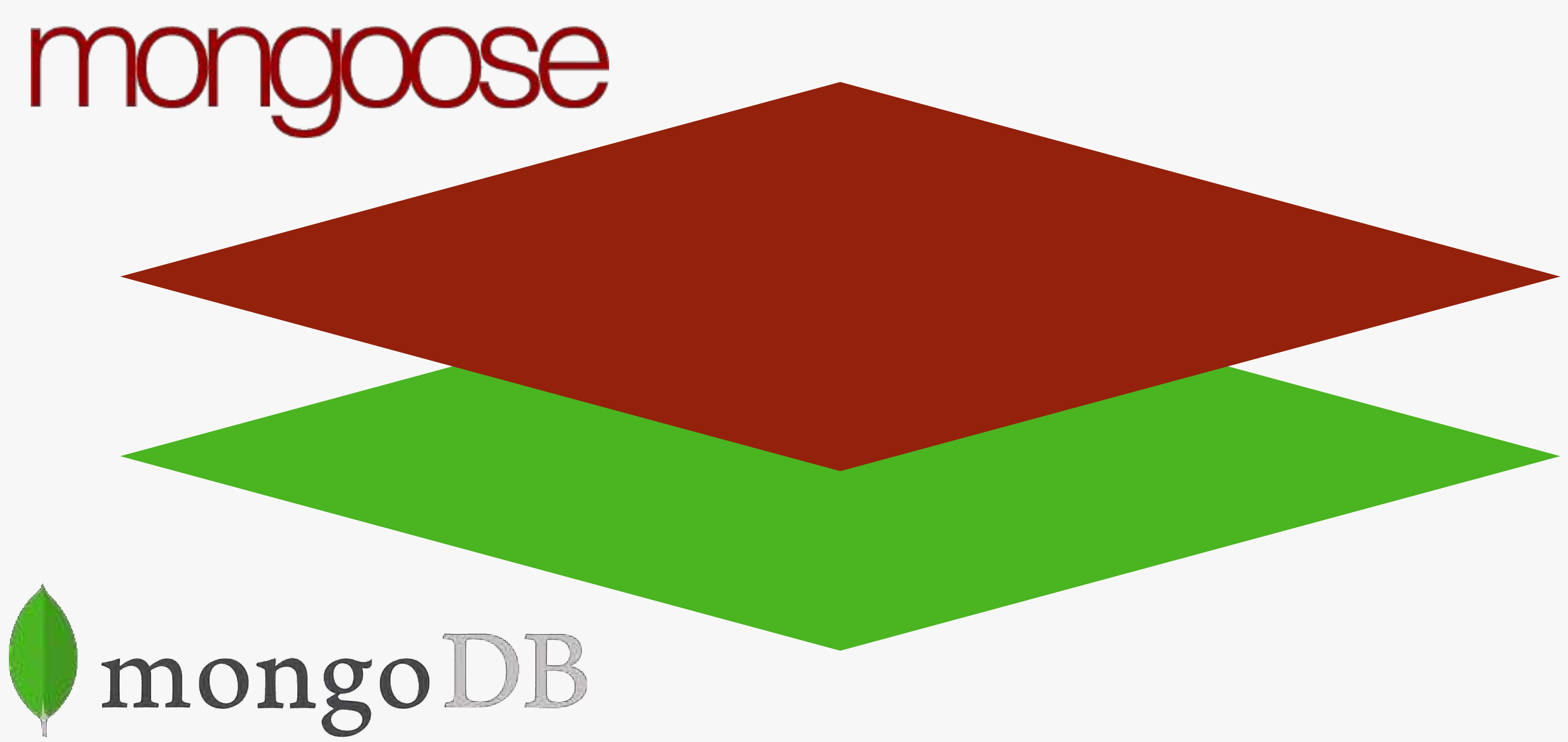
SECTION

USING MONGODB WITH MONGOOSE

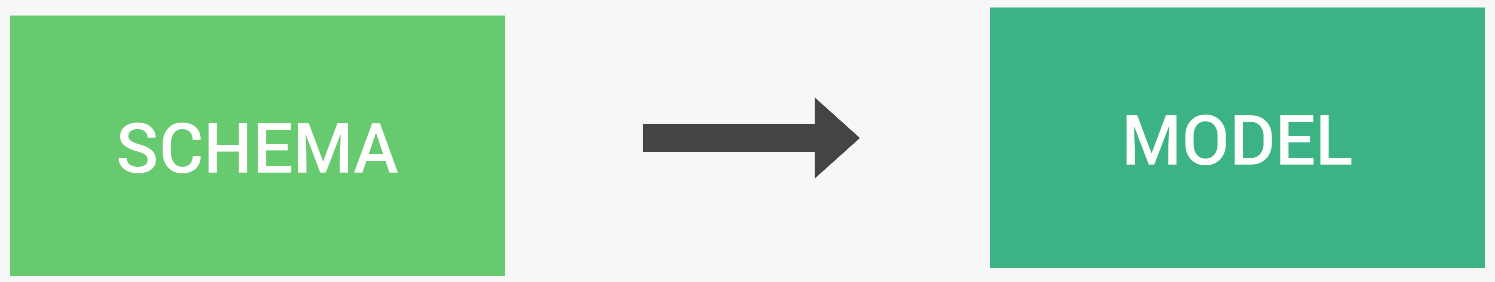
LECTURE

WHAT IS MONGOOSE?

WHAT IS MONGOOSE, AND WHY USE IT?



- Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js, a higher level of abstraction;
- Mongoose allows for rapid and simple development of mongoDB database interactions;
- Features: schemas to model data and relationships, easy data validation, simple query API, middleware, etc;
- **Mongoose schema:** where we model our data, by describing the structure of the data, default values, and validation;
- **Mongoose model:** a wrapper for the schema, providing an interface to the database for CRUD operations.


SCHEMA → MODEL

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

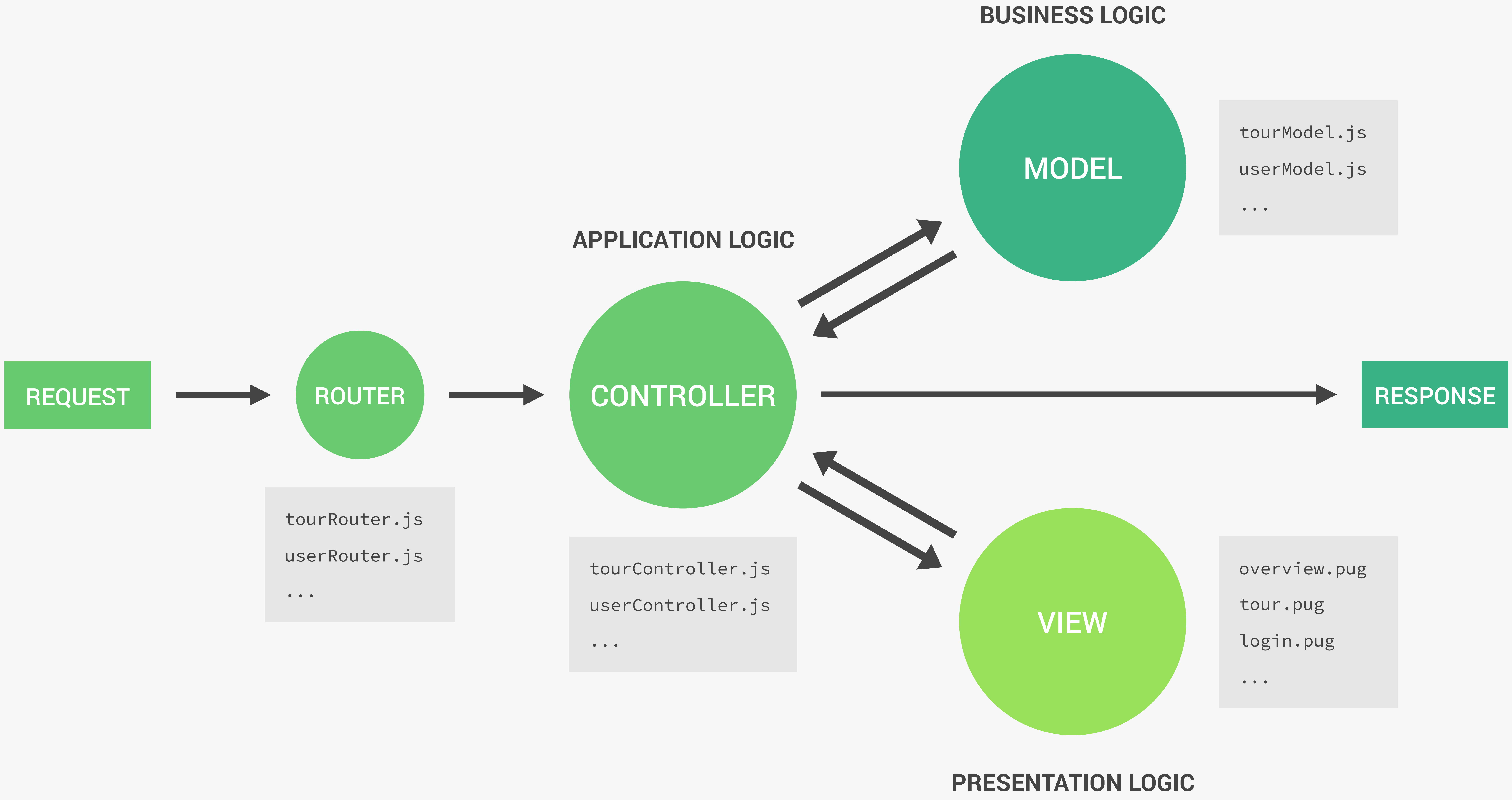
SECTION

USING MONGODB WITH MONGOOSE

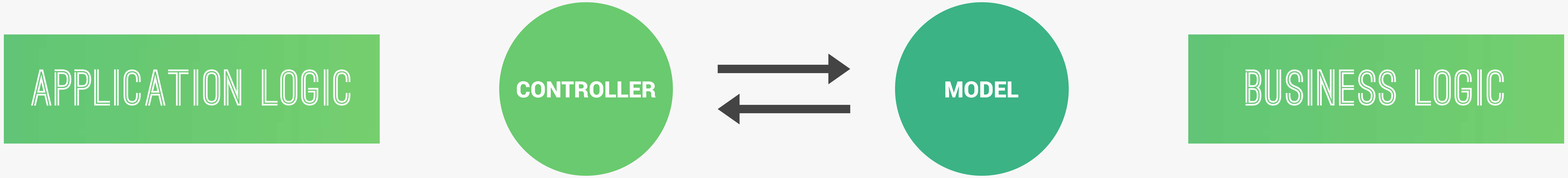
LECTURE

INTRO TO BACK-END ARCHITECTURE: MVC, TYPES OF LOGIC, AND MORE

MVC ARCHITECTURE IN OUR EXPRESS APP



APPLICATION VS. BUSINESS LOGIC



- Code that is only concerned about the application's implementation, not the underlying business problem we're trying to solve (e.g. showing and selling tours);
- Concerned about managing requests and responses;
- About the app's more technical aspects;
- Bridge between model and view layers.

- Code that actually solves the business problem we set out to solve;
- Directly related to business rules, how the business works, and business needs;
- Examples:
  - Creating new tours in the database;
  - Checking if user's password is correct;
  - Validating user input data;
  - Ensuring only users who bought a tour can review it.

- **Fat models/thin controllers:** offload as much logic as possible into the models, and keep the controllers as simple and lean as possible.

SECTION 9 — ERROR HANDLING WITH EXPRESS

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

ERROR HANDLING WITH EXPRESS

LECTURE

AN OVERVIEW OF ERROR HANDLING

ERROR HANDLING IN EXPRESS: AN OVERVIEW

OPERATIONAL ERRORS

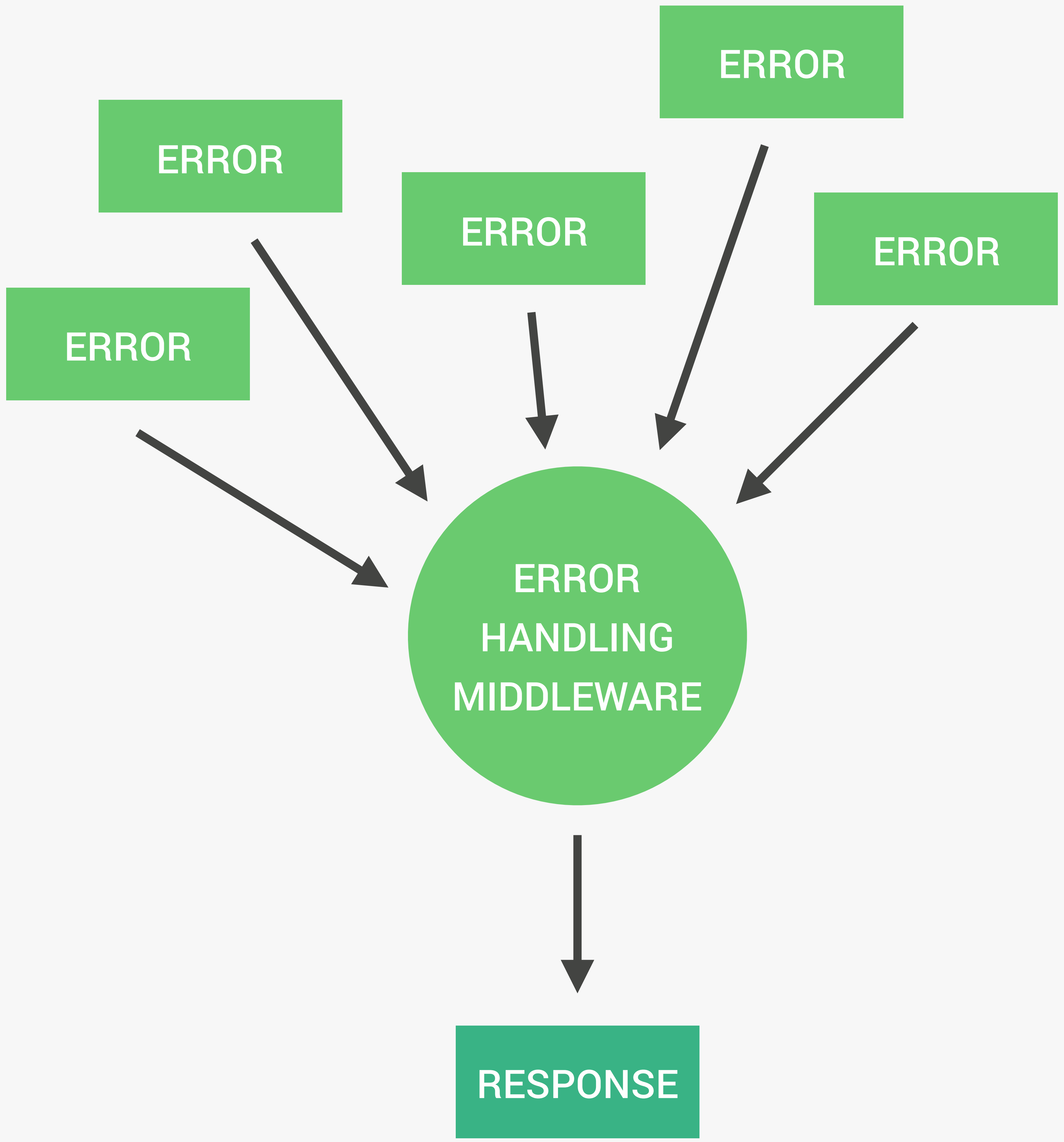
Problems that we can predict will happen at some point, so we just need to handle them in advance.

- Invalid path accessed;
- Invalid user input (validator error from mongoose);
- Failed to connect to server;
- Failed to connect to database;
- Request timeout;
- Etc...

PROGRAMMING ERRORS

Bugs that we developers introduce into our code. Difficult to find and handle.

- Reading properties on `undefined`;
- Passing a number where an object is expected;
- Using `await` without `async`;
- Using `req.query` instead of `req.body`;
- Etc...



SECTION 10 — AUTHENTICATION, AUTHORIZATION AND SECURITY

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

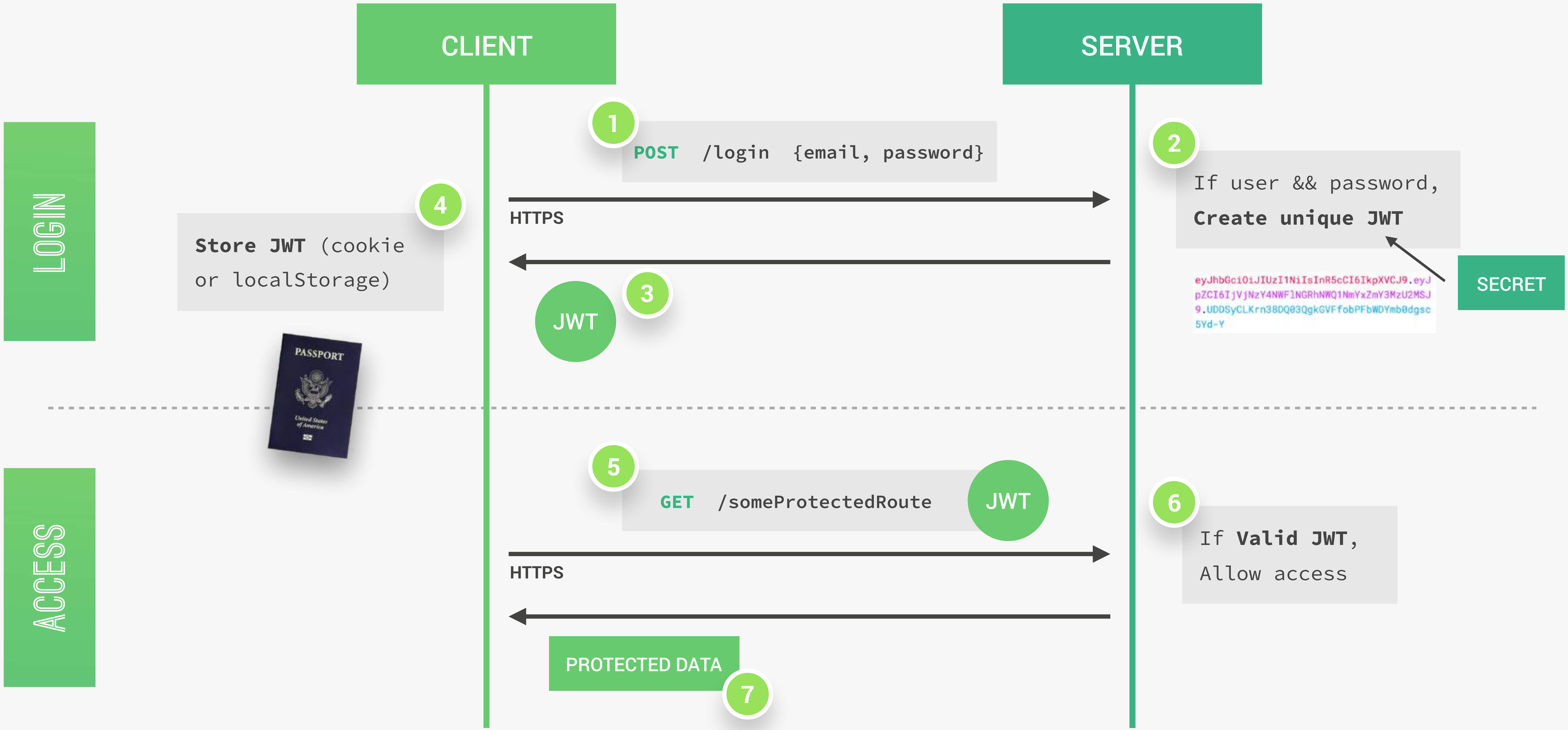
SECTION

AUTHENTICATION, AUTHORIZATION AND SECURITY

LECTURE

HOW AUTHENTICATION WITH JWT WORKS

HOW JSON WEB TOKEN (JWT) AUTHENTICATION WORKS



CLIENT

SERVER

LOGIN

1 POST /login {email, password}

HTTPS

2 If user && password,
Create unique JWT

SECRET

eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjVjNzY4NWFlNGRhNWQ1NmYxZmY3MzU2MSJ9.UDDSyCLKrn38DQ03QgkGVFfobPFbWDYmb0dgsc5Yd-Y

3 JWT

4 **Store JWT** (cookie or localStorage)

PASSPORT

United States of America

ACCESS

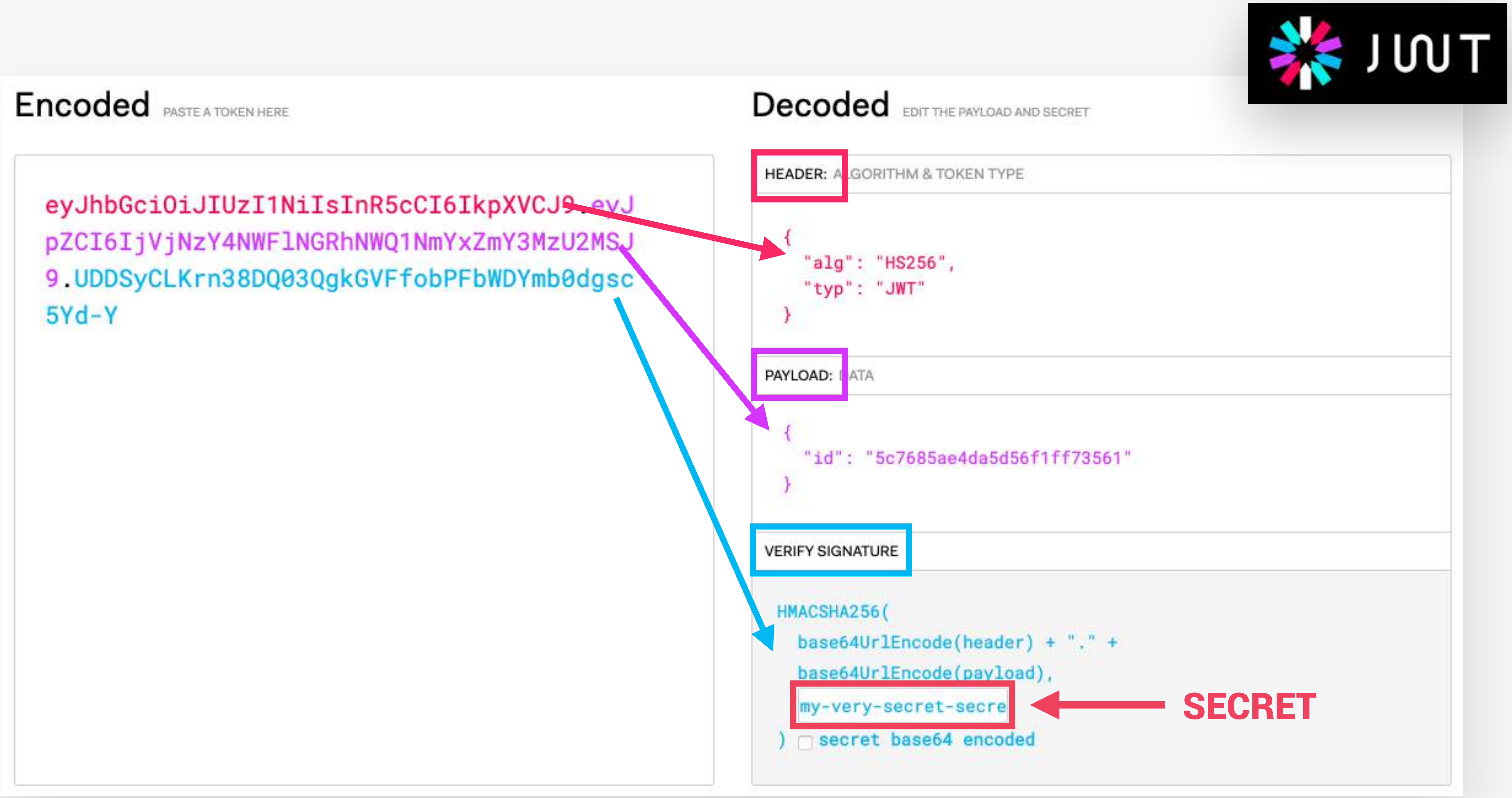
5 GET /someProtectedRoute JWT

HTTPS

6 If **Valid JWT**,
Allow access

PROTECTED DATA 7

WHAT A JWT LOOKS LIKE



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
pZCI6IjVjNzY4NWFlNGRhNWQ1NmYxZmY3MzU2MSJ
9.UDDSyCLKrn38DQ03QgkGVFfobPFbWDYmb0dgsc
5Yd-Y
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

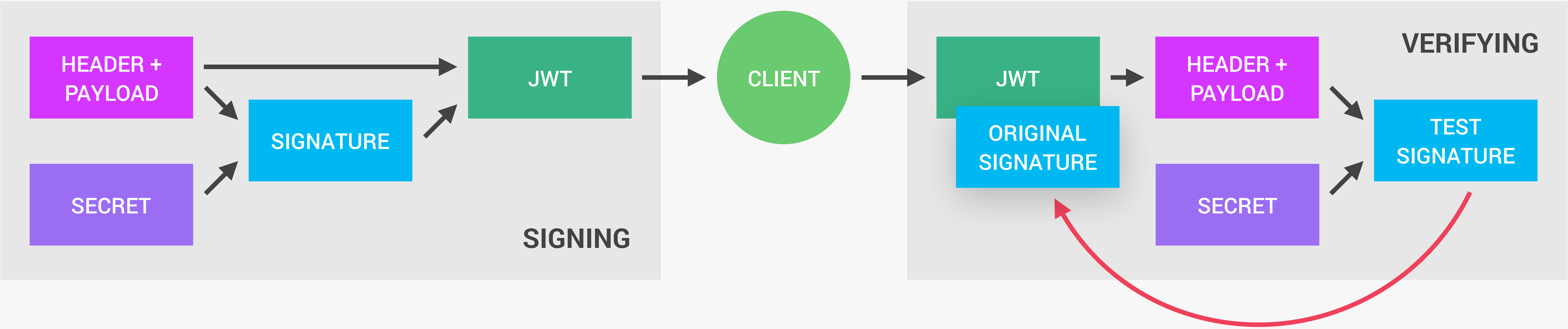
PAYLOAD: DATA

```
{
  "id": "5c7685ae4da5d56f1ff73561"
}
```

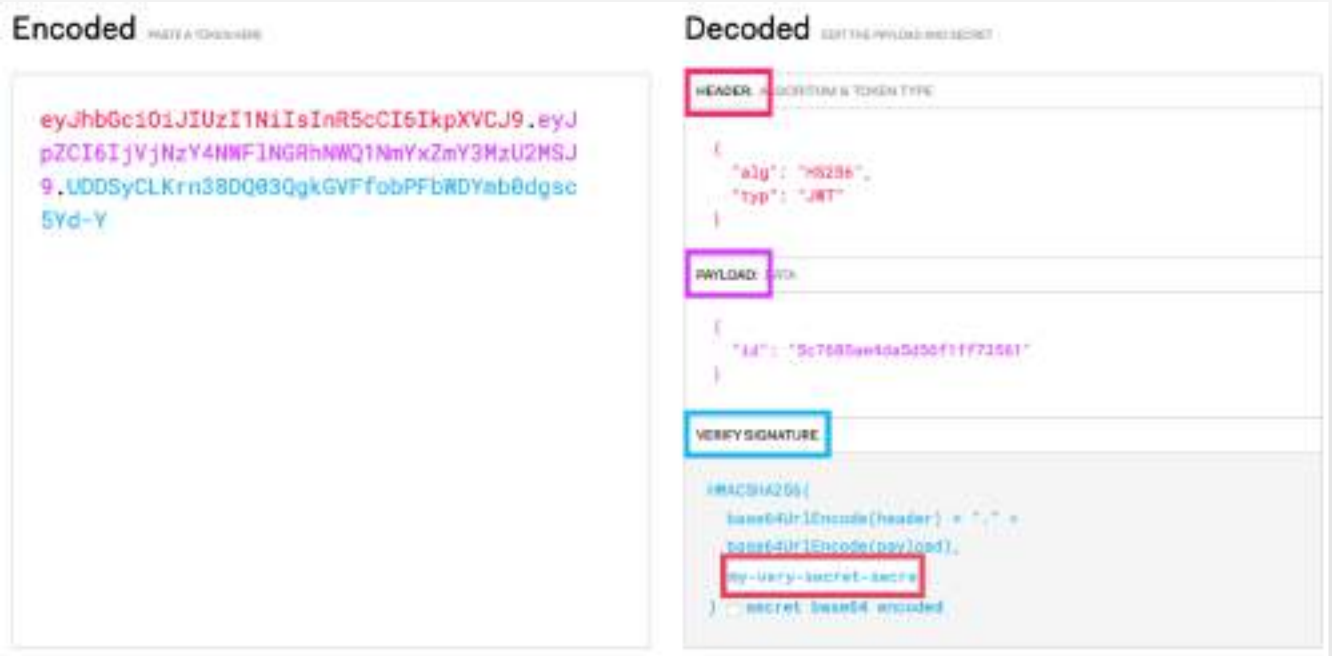
VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  my-very-secret-secre
) □ secret base64 encoded
```

HOW SIGNING AND VERIFYING WORKS



COMPARE WITH ORIGINAL SIGNATURE



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ
pZCI6IjVjNzY4NWFlNGRhNWQ1NmYxZmY3MzU2MSJ
9.UDDSyCLKrn38DQ03QgkGVFfobPFbWDYmb0dgsc
5Yd-Y
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

```
{
  "alg": "HS256",
  "typ": "JWT"
}
```

PAYLOAD: DATA

```
{
  "id": "5c7685ae4da5d56f1ff73561"
}
```

VERIFY SIGNATURE

```
HMACSHA256(
  base64UrlEncode(header) + "." +
  base64UrlEncode(payload),
  my-very-secret-secre
) ☐ secret base64 encoded
```

`test signature === signature` 👉 Data has not been modified 👉 **Authenticated**

`test signature !== signature` 👉 Data has been modified 👉 **Not authenticated**

👉 Without the secret, one will be able to manipulate the JWT data, because they cannot create a valid signature for the new data!

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

AUTHENTICATION, AUTHORIZATION AND SECURITY

LECTURE

SECURITY BEST PRACTICES

SECURITY BEST PRACTICES AND SUGGESTIONS

👉 COMPROMISED DATABASE

- ✅ Strongly encrypt passwords with salt and hash (`bcrypt`)
- ✅ Strongly encrypt password reset tokens (SHA 256)

👉 BRUTE FORCE ATTACKS

- ✅ Use `bcrypt` (to make login requests slow)
- ↘️ Implement rate limiting (`express-rate-limit`)
- ⚛️ Implement maximum login attempts

👉 CROSS-SITE SCRIPTING (XSS) ATTACKS

- ↘️ Store JWT in HTTPOnly cookies
- ↘️ Sanitize user input data
- ↘️ Set special HTTP headers (`helmet` package)

👉 DENIAL-OF-SERVICE (DOS) ATTACK

- ↘️ Implement rate limiting (`express-rate-limit`)
- ↘️ Limit body payload (in `body-parser`)
- ✅ Avoid evil regular expressions

👉 NOSQL QUERY INJECTION

- ✅ Use `mongoose` for MongoDB (because of SchemaTypes)
- ↘️ Sanitize user input data

👉 OTHER BEST PRACTICES AND SUGGESTIONS

- ✅ Always use HTTPS
- ✅ Create random password reset tokens with expiry dates
- ✅ Deny access to JWT after password change
- ✅ Don’t commit sensitive config data to Git
- ✅ Don’t send error details to clients
- ⚛️ Prevent Cross-Site Request Forgery (`csurf` package)
- ⚛️ Require re-authentication before a high-value action
- ⚛️ Implement a blacklist of untrusted JWT
- ⚛️ Confirm user email address after first creating account
- ⚛️ Keep user logged in with refresh tokens
- ⚛️ Implement two-factor authentication
- ↘️ Prevent parameter pollution causing Uncaught Exceptions

SECTION 11 — MODELLING DATA AND ADVANCED MONGOOSE

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

MODELLING DATA AND ADVANCED MONGOOSE

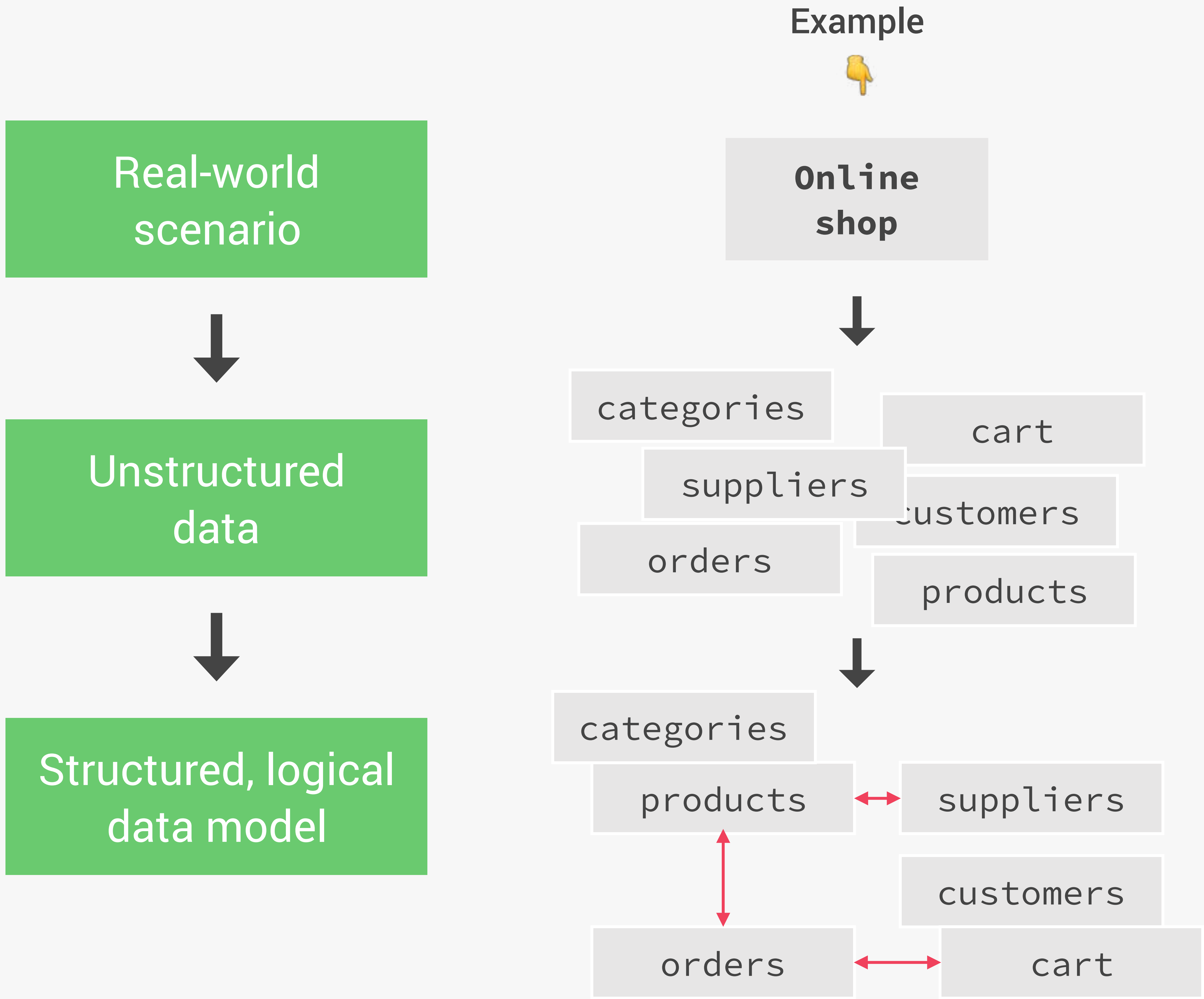
LECTURE

MONGODB DATA MODELLING

@JONASSCHMEDTMAN

"DATA... WHAT? 🤔"

DATA MODELLING



Real-world scenario

Unstructured data

Structured, logical data model

Example 👇

Online shop

categories

cart

suppliers

customers

orders

products

categories

products

suppliers

customers

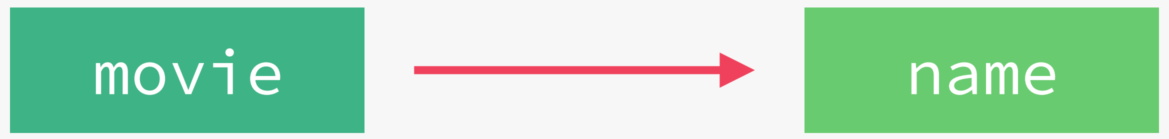
orders

cart

1. Different types of **relationships** between data
2. **Referencing**/normalization vs. **embedding**/denormalization
3. **Embedding** or **referencing** other documents?
4. **Types** of referencing

1. TYPES OF RELATIONSHIPS BETWEEN DATA

1:1



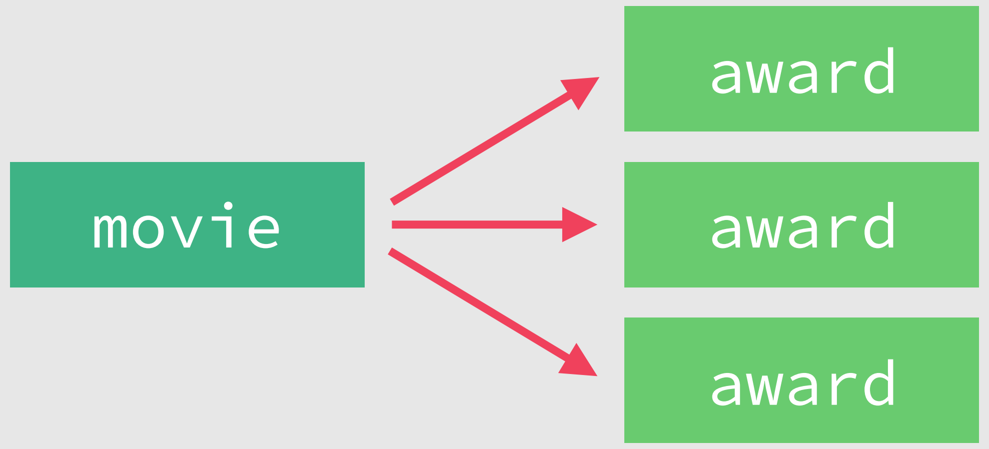
movie

name

*(**1** movie can only have **1** name)*

1:MANY

👉 1:FEW



movie

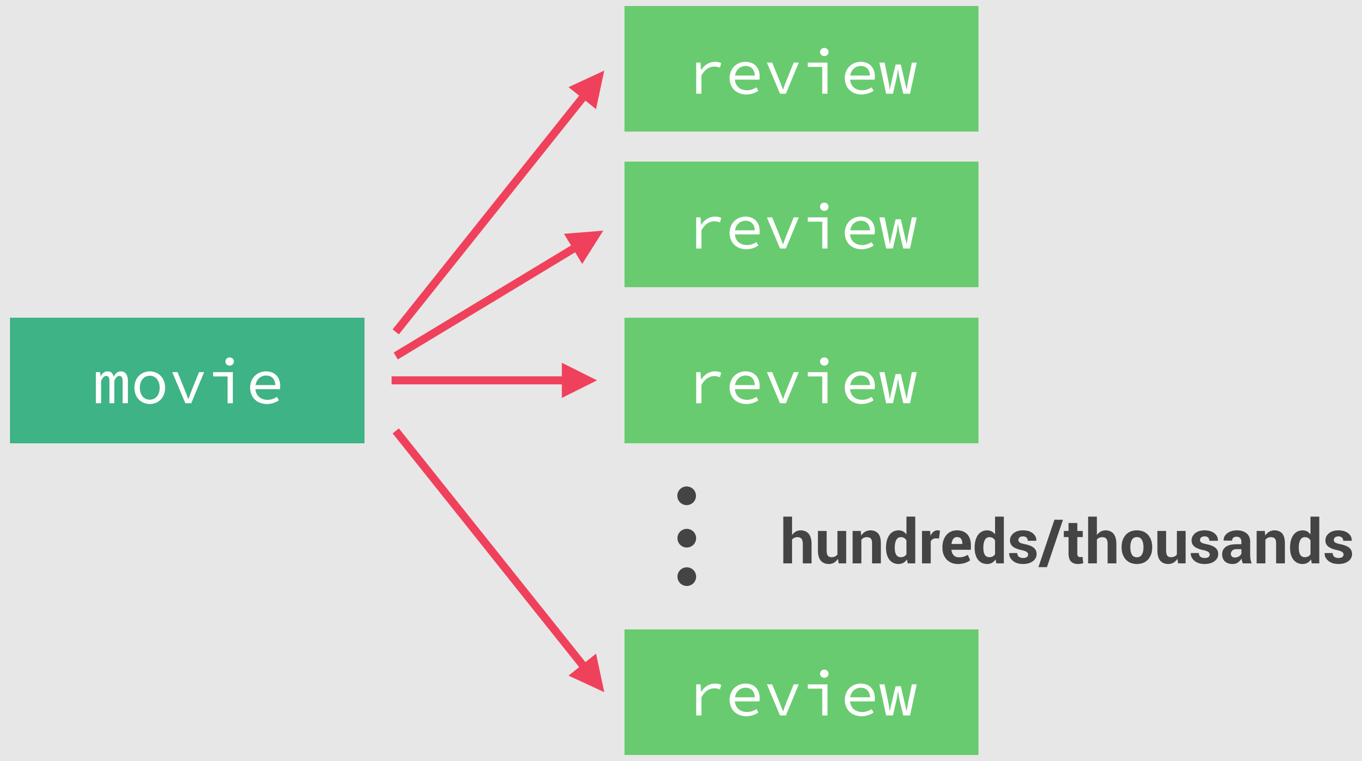
award

award

award

*(**1** movie can win **many** awards)*

👉 1:MANY



movie

review

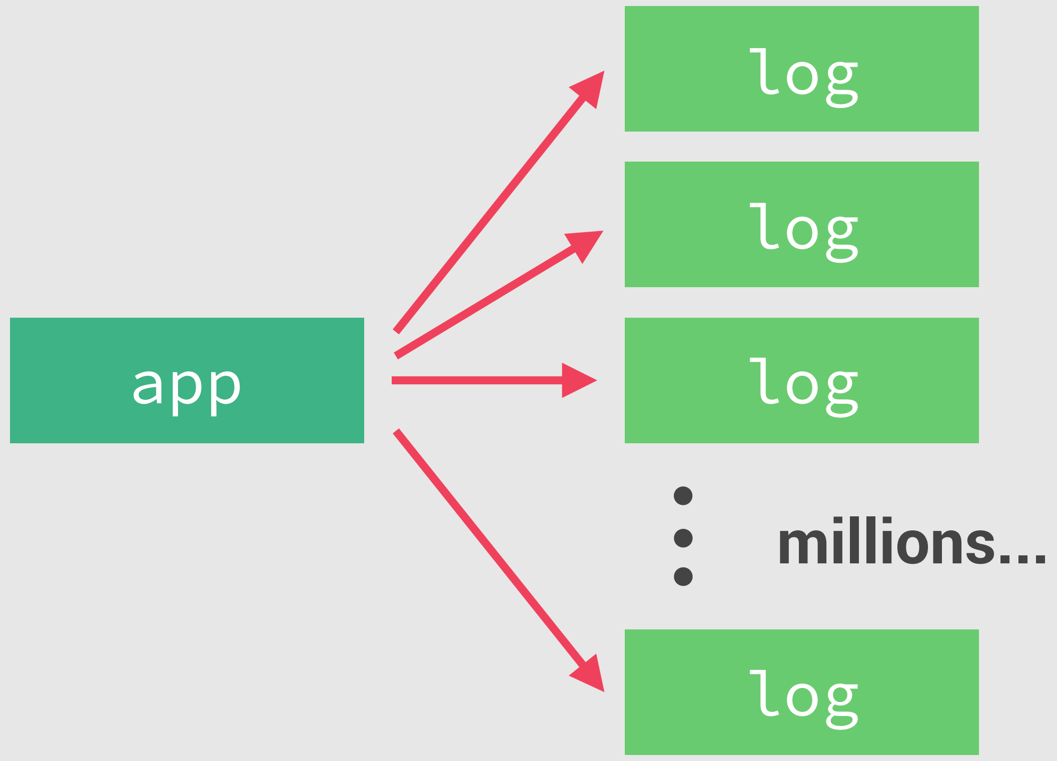
review

review

hundreds/thousands

review

👉 1:TON



app

log

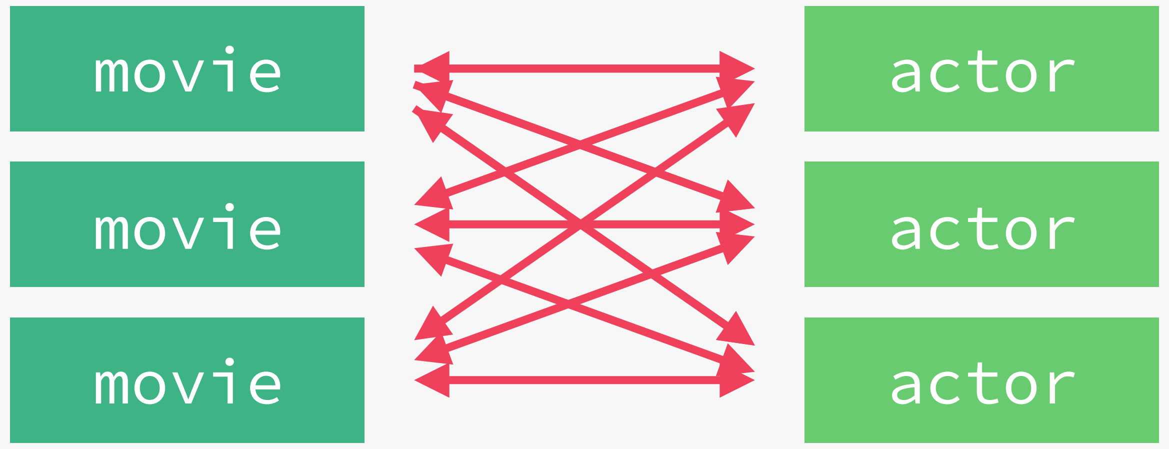
log

log

millions...

log

MANY:MANY



movie

movie

movie

actor

actor

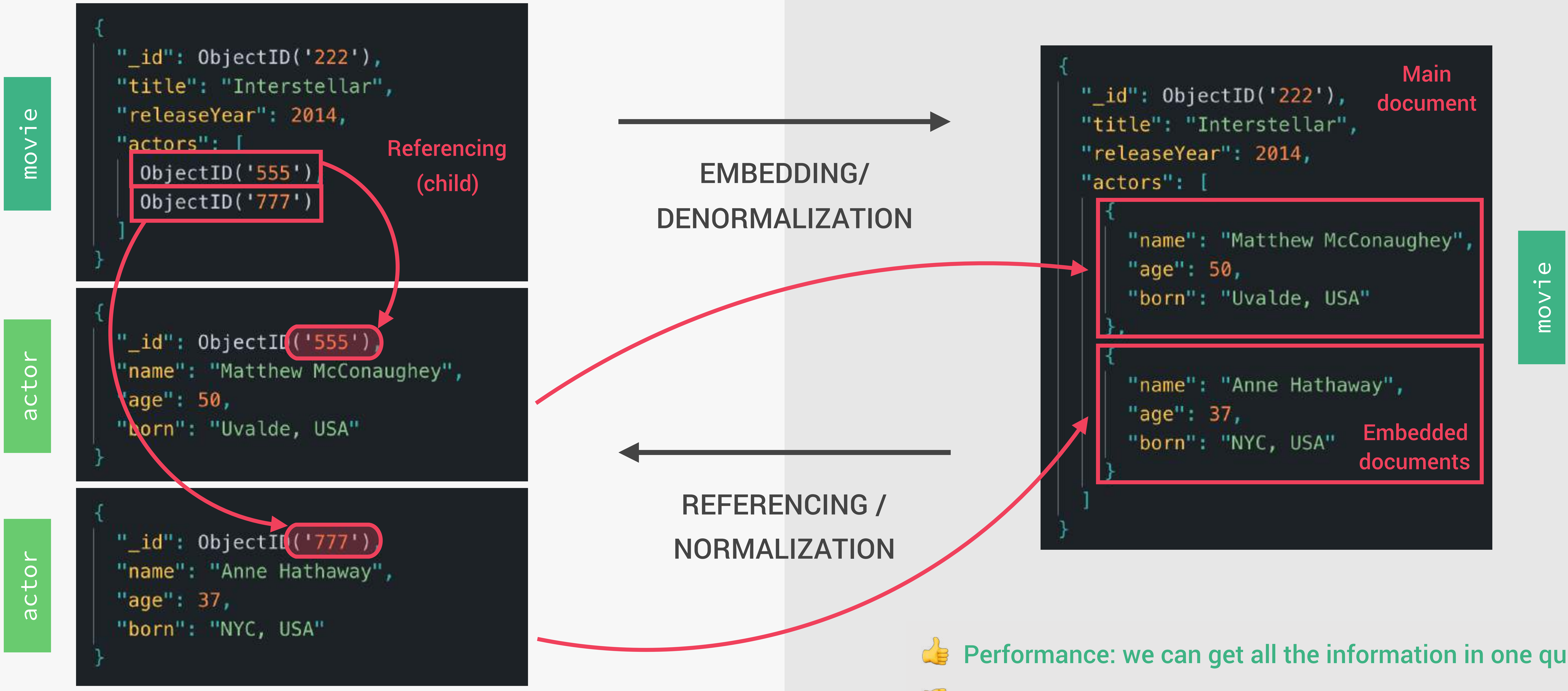
actor

*(One movie can have **many** actors, but one actor can also play in **many** movies)*

2. REFERENCING VS. EMBEDDING

REFERENCED / NORMALIZED

EMBEDDED / DENORMALIZED



```
{
  "_id": ObjectID('222'),
  "title": "Interstellar",
  "releaseYear": 2014,
  "actors": [
    ObjectID('555'),
    ObjectID('777')
  ]
}
```

```
{
  "_id": ObjectID('555'),
  "name": "Matthew McConaughey",
  "age": 50,
  "born": "Uvalde, USA"
}
```

```
{
  "_id": ObjectID('777'),
  "name": "Anne Hathaway",
  "age": 37,
  "born": "NYC, USA"
}
```

```
{
  "_id": ObjectID('222'),
  "title": "Interstellar",
  "releaseYear": 2014,
  "actors": [
    {
      "name": "Matthew McConaughey",
      "age": 50,
      "born": "Uvalde, USA"
    },
    {
      "name": "Anne Hathaway",
      "age": 37,
      "born": "NYC, USA"
    }
  ]
}
```

- 👍 Performance: it's easier to query each document on its own
- 👎 We need 2 queries to get data from referenced document

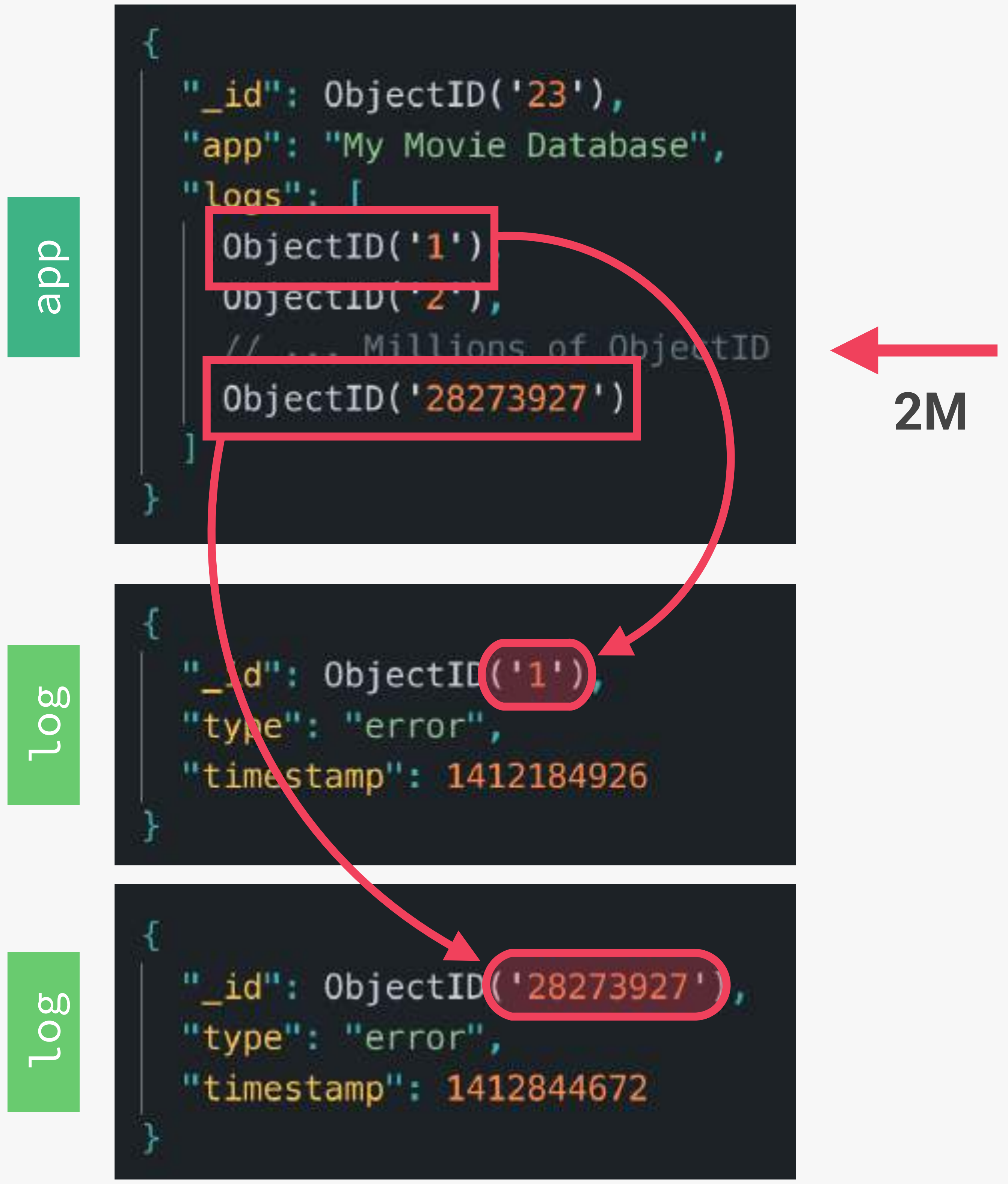
- 👍 Performance: we can get all the information in one query
- 👎 Impossible to query the embedded document on its own

3. WHEN TO EMBED AND WHEN TO REFERENCE? A PRACTICAL FRAMEWORK

👉 Combine all 3 criteria to take decision!	EMBEDDING	REFERENCING
1 RELATIONSHIP TYPE (How two datasets are related to each other)	👉 1:FEW 👉 1:MANY `Movies + Images (100)` ?	👉 1:MANY 👉 1:TON 👉 MANY:MANY
2 DATA ACCESS PATTERNS (How often data is read and written. Read/write ratio)	👉 Data is mostly **read** 👉 Data does **not** change quickly 👉 (**High** read/write ratio) `Movies + Images`	👉 Data is **updated** a lot 👉 (**Low** read/write ratio) `Movies + Reviews`
3 DATA CLOSENESS (How "much" the data is related, how we want to query)	👉 Datasets **really** belong together `User + Email Addresses`	👉 We frequently need to query both datasets **on their own** `Movies + Images`

4. TYPES OF REFERENCING

CHILD REFERENCING



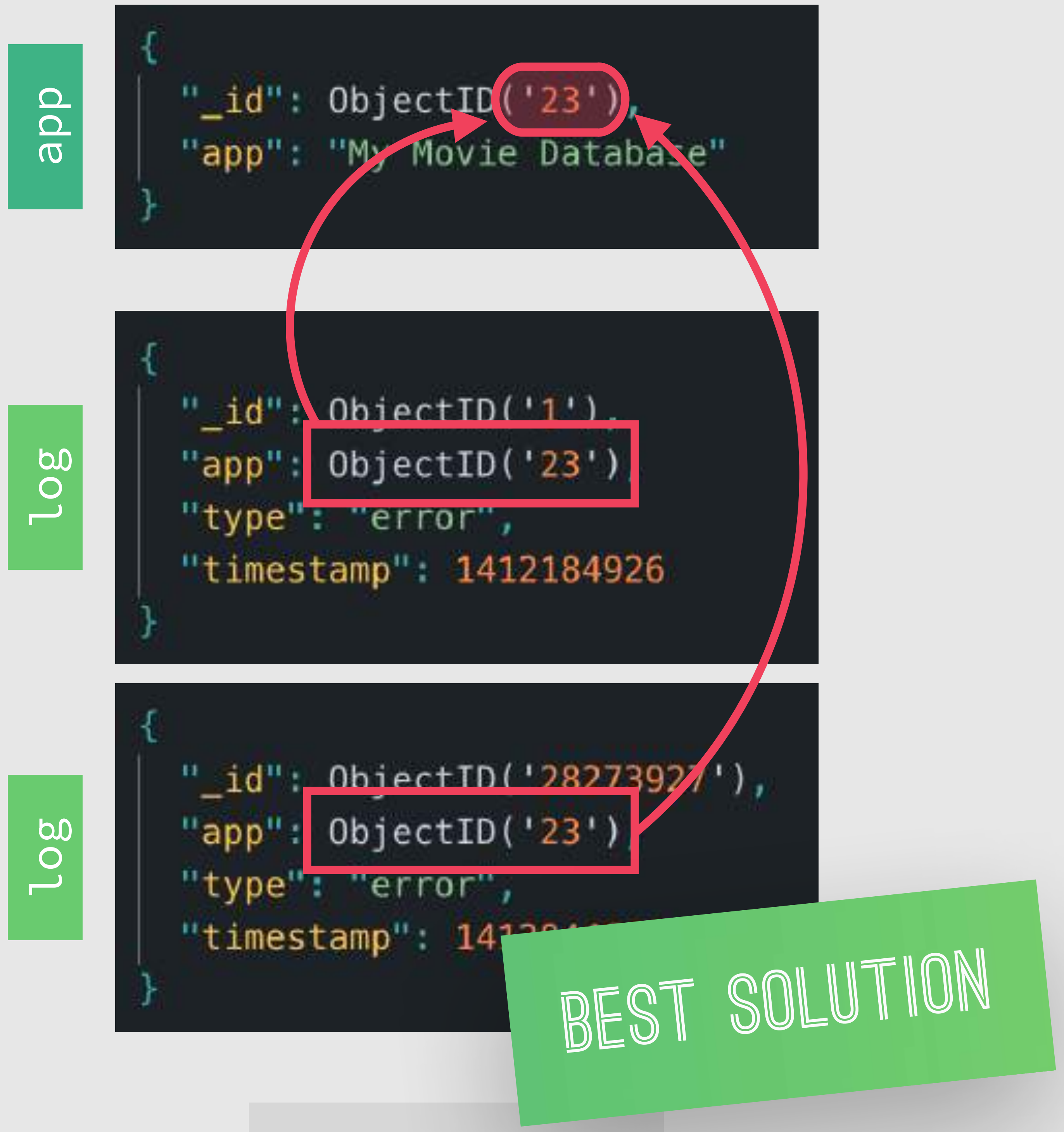
```
{
  "_id": ObjectID('23'),
  "app": "My Movie Database",
  "logs": [
    ObjectID('1')
    ObjectID('2'),
    // ... Millions of ObjectID
    ObjectID('28273927')
  ]
}
```

```
{
  "_id": ObjectID('1'),
  "type": "error",
  "timestamp": 1412184926
}
```

```
{
  "_id": ObjectID('28273927'),
  "type": "error",
  "timestamp": 1412844672
}
```

👉 1:FEW

PARENT REFERENCING



```
{
  "_id": ObjectID('23'),
  "app": "My Movie Database"
}
```

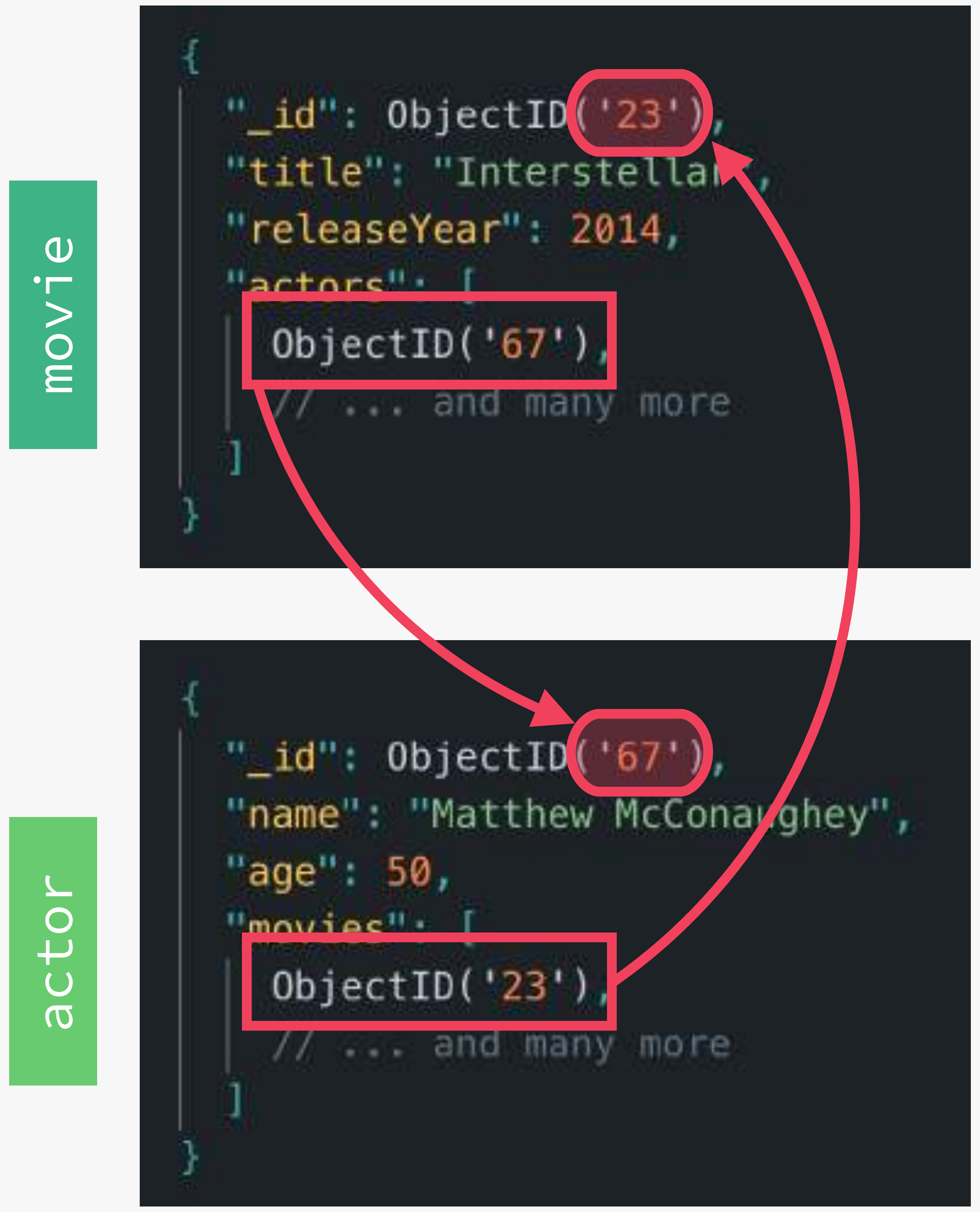
```
{
  "_id": ObjectID('1'),
  "app": ObjectID('23'),
  "type": "error",
  "timestamp": 1412184926
}
```

```
{
  "_id": ObjectID('28273927'),
  "app": ObjectID('23'),
  "type": "error",
  "timestamp": 141
}
```

👉 1:MANY

👉 1:TON

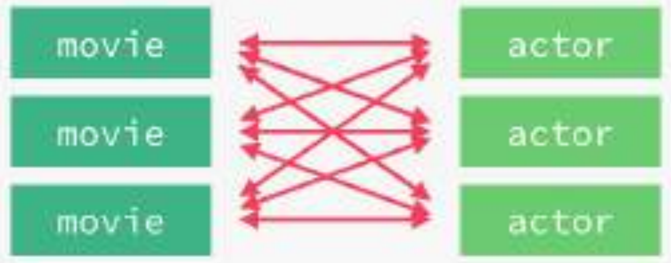
TWO-WAY REFERENCING



```
{
  "_id": ObjectID('23'),
  "title": "Interstellar",
  "releaseYear": 2014,
  "actors": [
    ObjectID('67'),
    // ... and many more
  ]
}
```

```
{
  "_id": ObjectID('67'),
  "name": "Matthew McConaughey",
  "age": 50,
  "movies": [
    ObjectID('23'),
    // ... and many more
  ]
}
```

👉 MANY:MANY



SUMMARY 🥳

👉 The most important principle is: Structure your data to **match the ways that your application queries and updates data**;

👉 In other words: Identify the questions that arise from your **application's use cases** first, and then model your data so that the **questions can get answered** in the most efficient way;

👉 In general, **always favor embedding**, unless there is a good reason not to embed. Especially on 1:FEW and 1:MANY relationships;

👉 A 1:TON or a MANY:MANY relationship is usually a good reason to **reference** instead of embedding;

👉 Also, favor **referencing** when data is updated a lot and if you need to frequently access a dataset on its own;

👉 Use **embedding** when data is mostly read but rarely updated, and when two datasets belong intrinsically together;

👉 Don't allow arrays to grow indefinitely. Therefore, if you need to normalize, use **child referencing** for 1:MANY relationships, and **parent referencing** for 1:TON relationships;

👉 Use **two-way referencing** for MANY:MANY relationships.

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

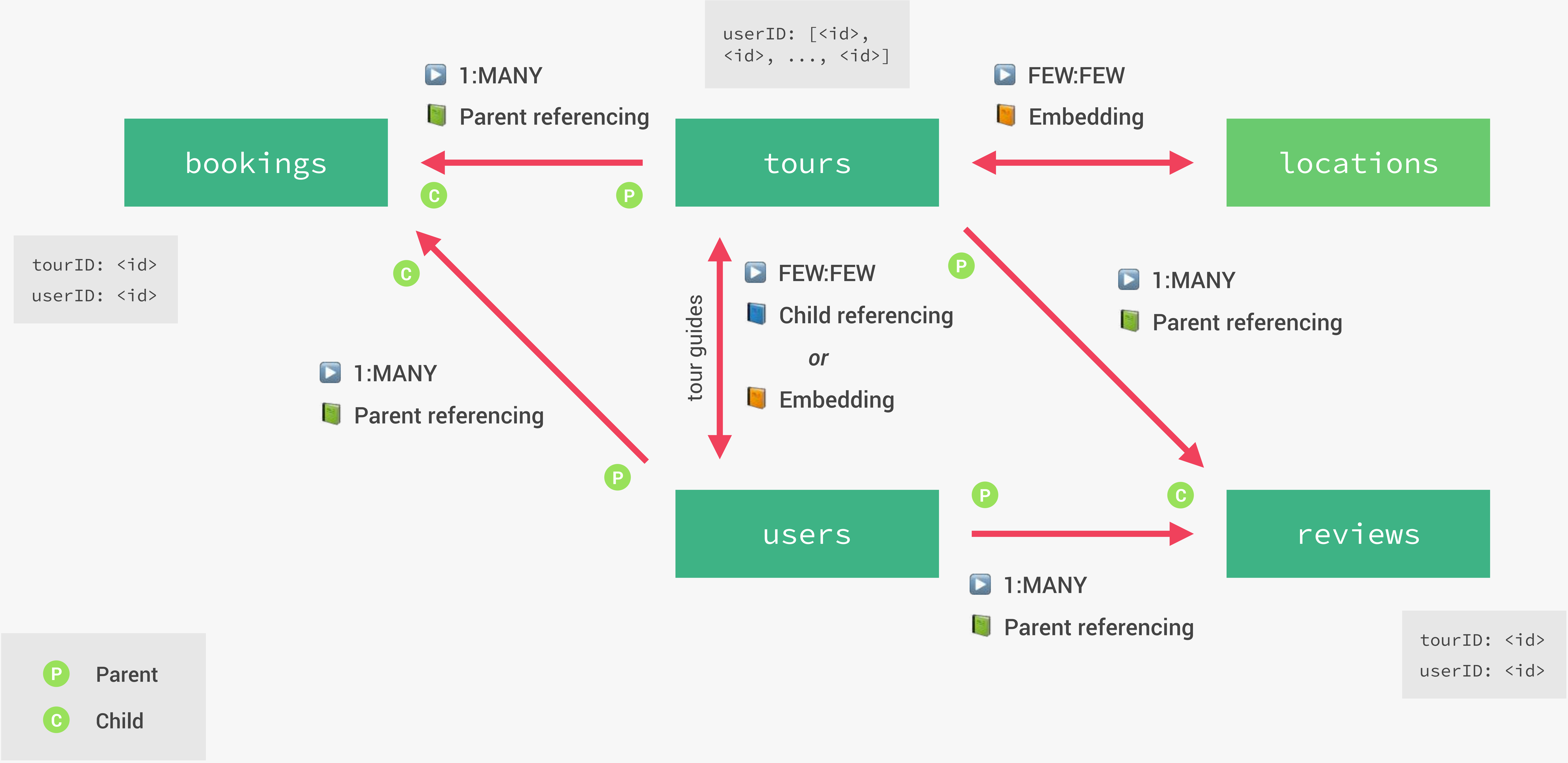
SECTION

MODELLING DATA AND ADVANCED MONGOOSE

LECTURE

DESIGNING OUR DATA MODEL

THE NATOURS DATA MODEL



```
userID: [<id>,
<id>, ..., <id>]
```

bookings ← tours:

- ▶ 1:MANY
- Parent referencing

tours ↔ locations:

- ▶ FEW:FEW
- Embedding

```
tourID: <id>
userID: <id>
```

tours ↔ users (tour guides):

- ▶ FEW:FEW
- Child referencing

or

- Embedding

tours → reviews:

- ▶ 1:MANY
- Parent referencing

users → bookings:

- ▶ 1:MANY
- Parent referencing

users → reviews:

- ▶ 1:MANY
- Parent referencing

```
tourID: <id>
userID: <id>
```

P Parent

C Child

SECTION 13 — ADVANCED FEATURES: PAYMENTS, EMAIL, FILE UPLOADS

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

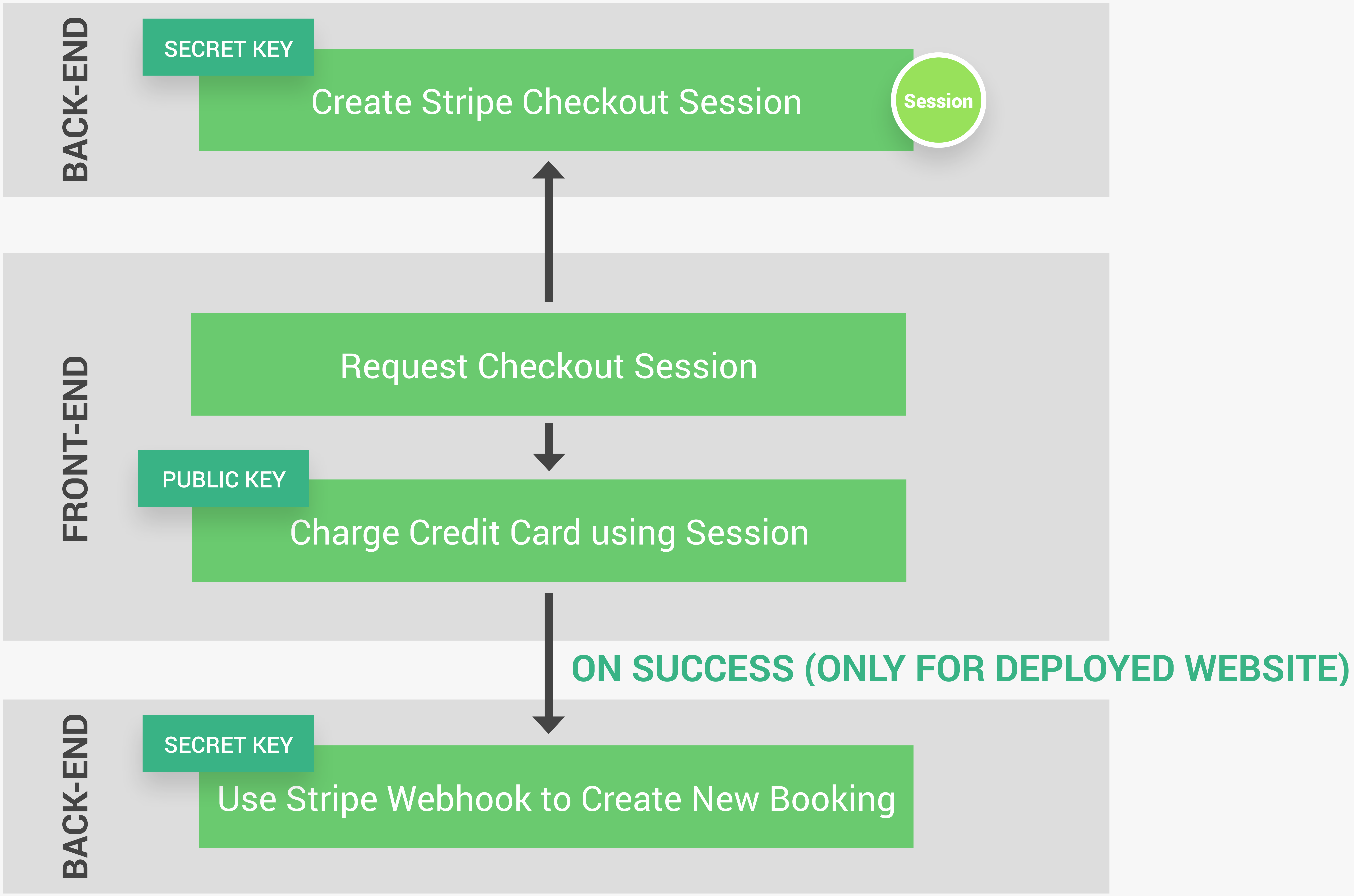
SECTION

ADVANCED FEATURES: PAYMENTS, EMAIL, FILE UPLOADS

LECTURE

CREDIT CARD PAYMENTS WITH STRIPE

STRIPE WORKFLOW



BACK-END

SECRET KEY

Create Stripe Checkout Session

Session

FRONT-END

Request Checkout Session

PUBLIC KEY

Charge Credit Card using Session

ON SUCCESS (ONLY FOR DEPLOYED WEBSITE)

BACK-END

SECRET KEY

Use Stripe Webhook to Create New Booking

JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

@JONASSCHMEDTMAN

SECTION

ADVANCED FEATURES: PAYMENTS, EMAIL, FILE UPLOADS

LECTURE

FINAL CONSIDERATIONS

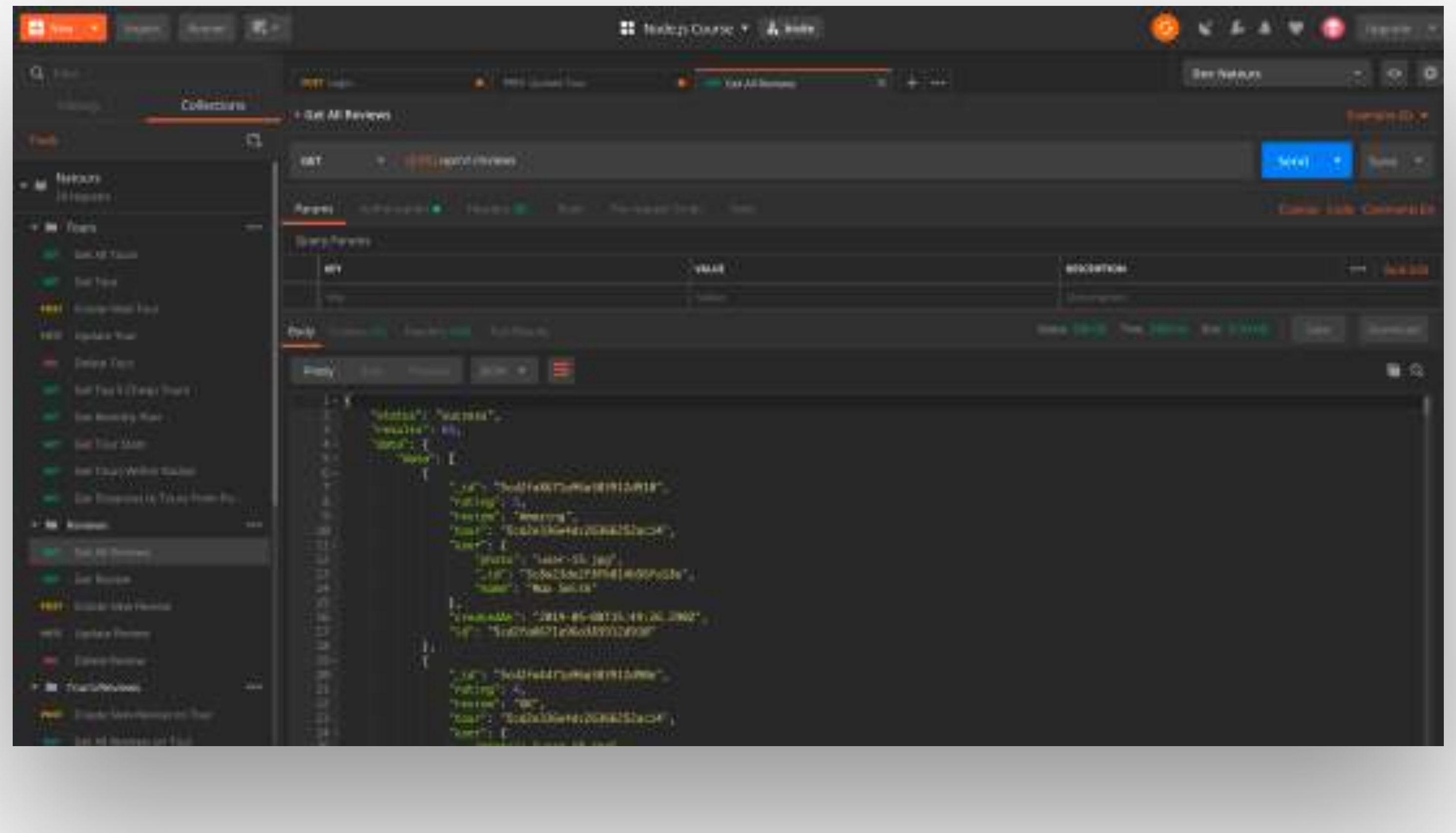
CHALLENGES (API) 🤓

- Implement restriction that users can only review a tour **that they have actually booked**;

- Implement nested **booking** routes: `/tours/:id/bookings` and `/users/:id/bookings`;

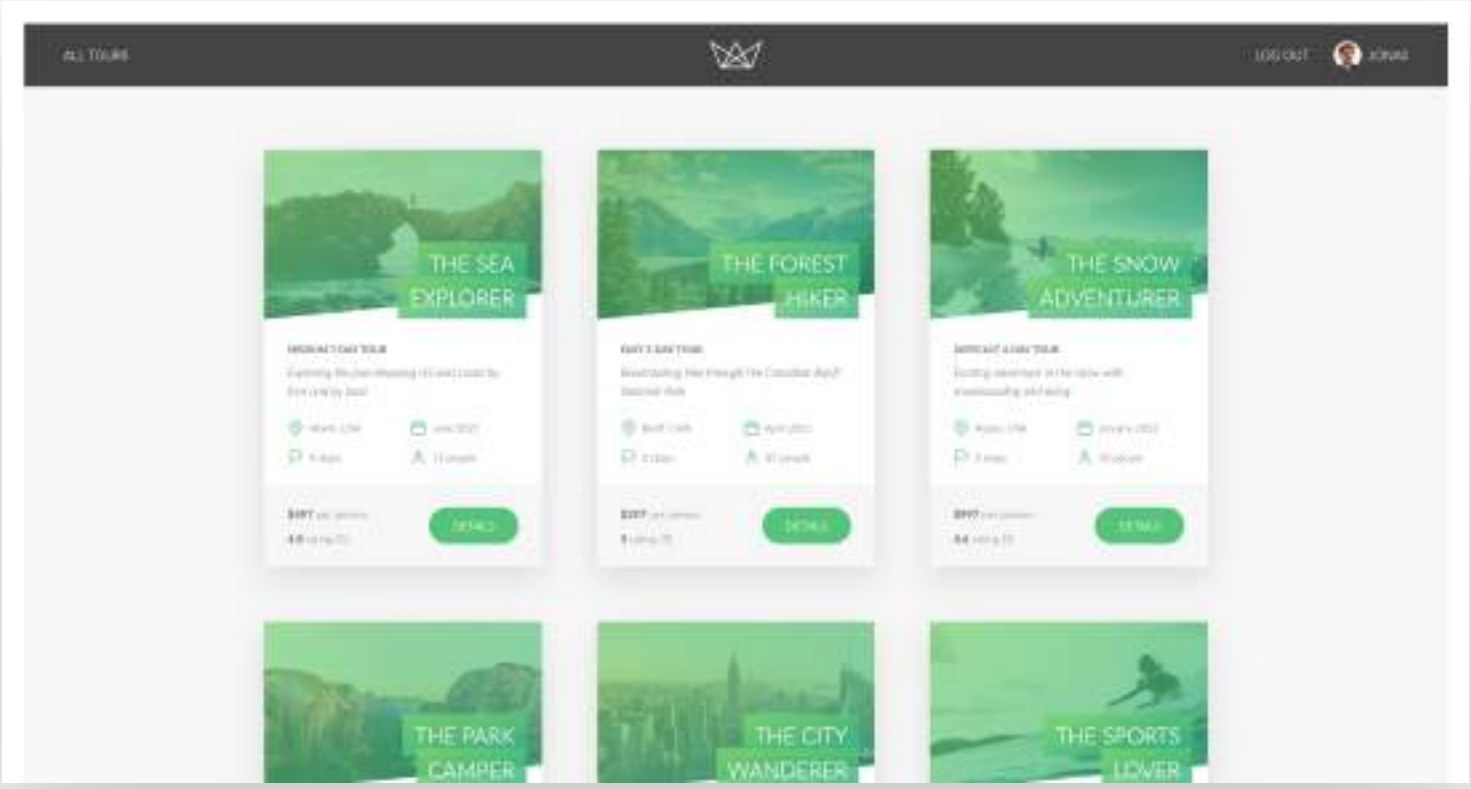
- **Improve tour dates:** add a `participants` and a `soldOut` field to each date. A date then becomes like an instance of the tour. Then, when a user books, they need to select one of the dates. A new booking will increase the number of participants in the date, until it is booked out (`participants > maxGroupSize`). So, when a user wants to book, you need to check if tour on the selected date is still available;

- Implement **advanced authentication features**: confirm user email, keep users logged in with refresh tokens, two-factor authentication, etc.



CHALLENGES (WEBSITE) 🤓

- 👉 Implement a **sign up** from, similar to the login form;
- 👉 On the tour detail page, if a user has taken a tour, allow them **add a review directly on the website**. Implement a form for this;
- 👉 **Hide the entire booking section** on the tour detail page if current user has already booked the tour (also prevent duplicate bookings on the model);
- 👉 **Implement "like tour" functionality**, with favourite tour page;
- 👉 On the user account page, implement the "**My Reviews**" page, where all reviews are displayed, and a user can edit them. *(If you know React ⚛️, this would be an amazing way to use the Natours API and train your skills!);*
- 👉 For administrators, implement all the "**Manage**" pages, where they can CRUD (create, read, update, delete) tours, users, reviews, and bookings.



END