



Subscribe here

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SLIDES FOR THEORY LECTURES

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

Follow me here



@JONASSCHMEDTMAN



TABLE OF CONTENTS: THEORY LECTURES (CLICK THE TITLES)

- [1 What Is Node.js and Why Use It?](#)
- [2 Blocking and Non-Blocking: Asynchronous Nature of Node.js](#)
- [3 An Overview of How the Web Works](#)
- [4 Front-End vs. Back-End Web Development](#)
- [5 Static vs Dynamic vs API](#)
- [6 Node, V8, Libuv and C++](#)
- [7 Processes, Threads and the Thread Pool](#)
- [8 The Node.js Event Loop](#)
- [9 Events and Event-Driven Architecture](#)
- [10 Introduction to Streams](#)
- [11 How Requiring Modules Really Works](#)
- [12 What is Express?](#)
- [13 APIs and RESTful API Design](#)
- [14 Middleware and the Request-Response Cycle](#)
- [15 What is MongoDB?](#)
- [16 What Is Mongoose?](#)
- [17 Intro to Back-End Architecture: MVC, Types of Logic, and More](#)
- [18 An Overview of Error Handling](#)
- [19 How Authentication with JWT Works](#)
- [20 Security Best Practices](#)
- [21 MongoDB Data Modelling](#)
- [22 Designing Our Data Model](#)
- [23 Credit Card Payments with Stripe](#)
- [24 Final Considerations](#)

SECTION 2 – INTRODUCTION TO NODE.JS



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

INTRODUCTION TO NODEJS

LECTURE

WHAT IS NODEJS AND WHY USE IT?



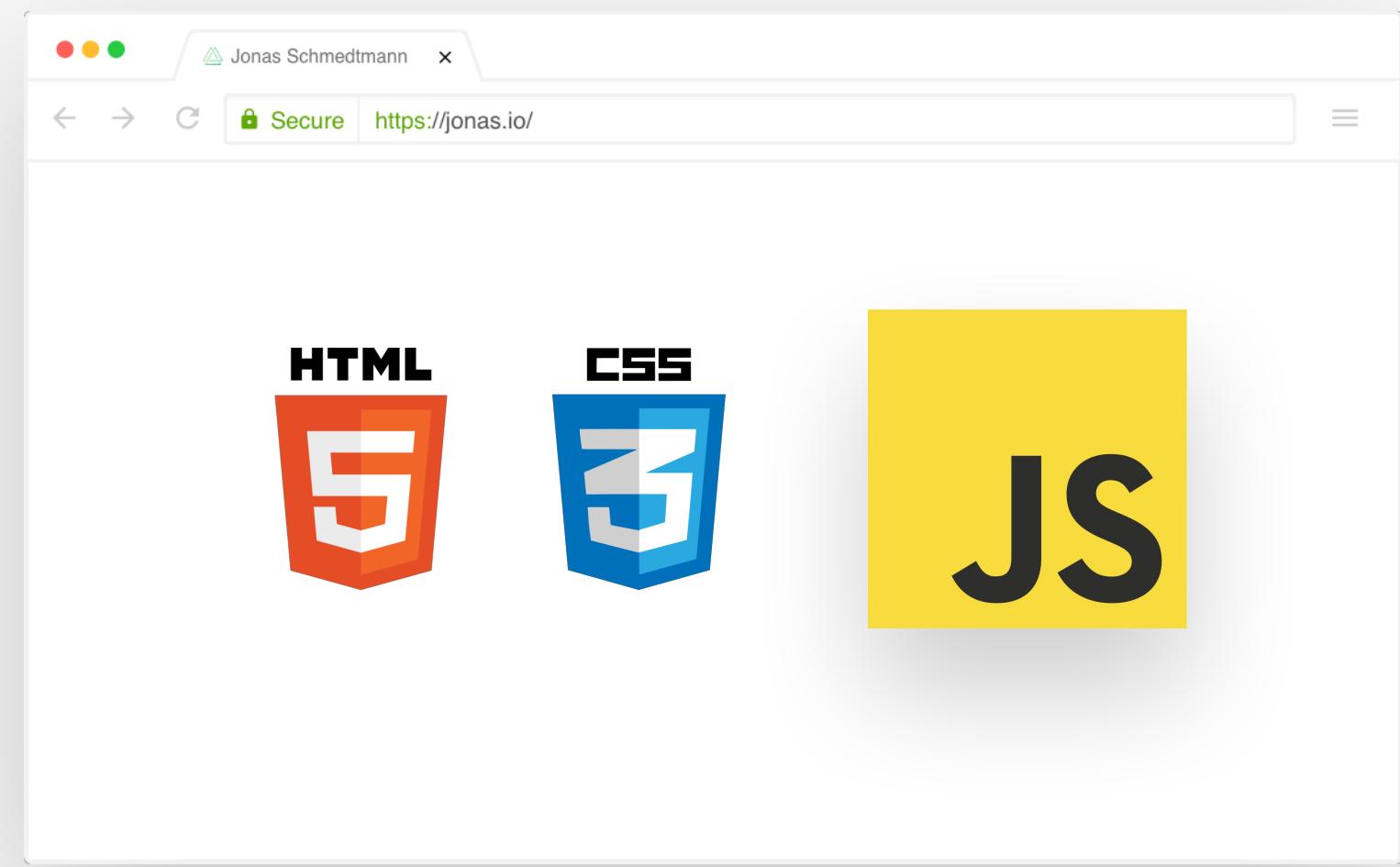
@JONASSCHMEDTMAN

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).





JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

INTRODUCTION TO NODE.JS

LECTURE

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



@JONASSCHMEDTMAN

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



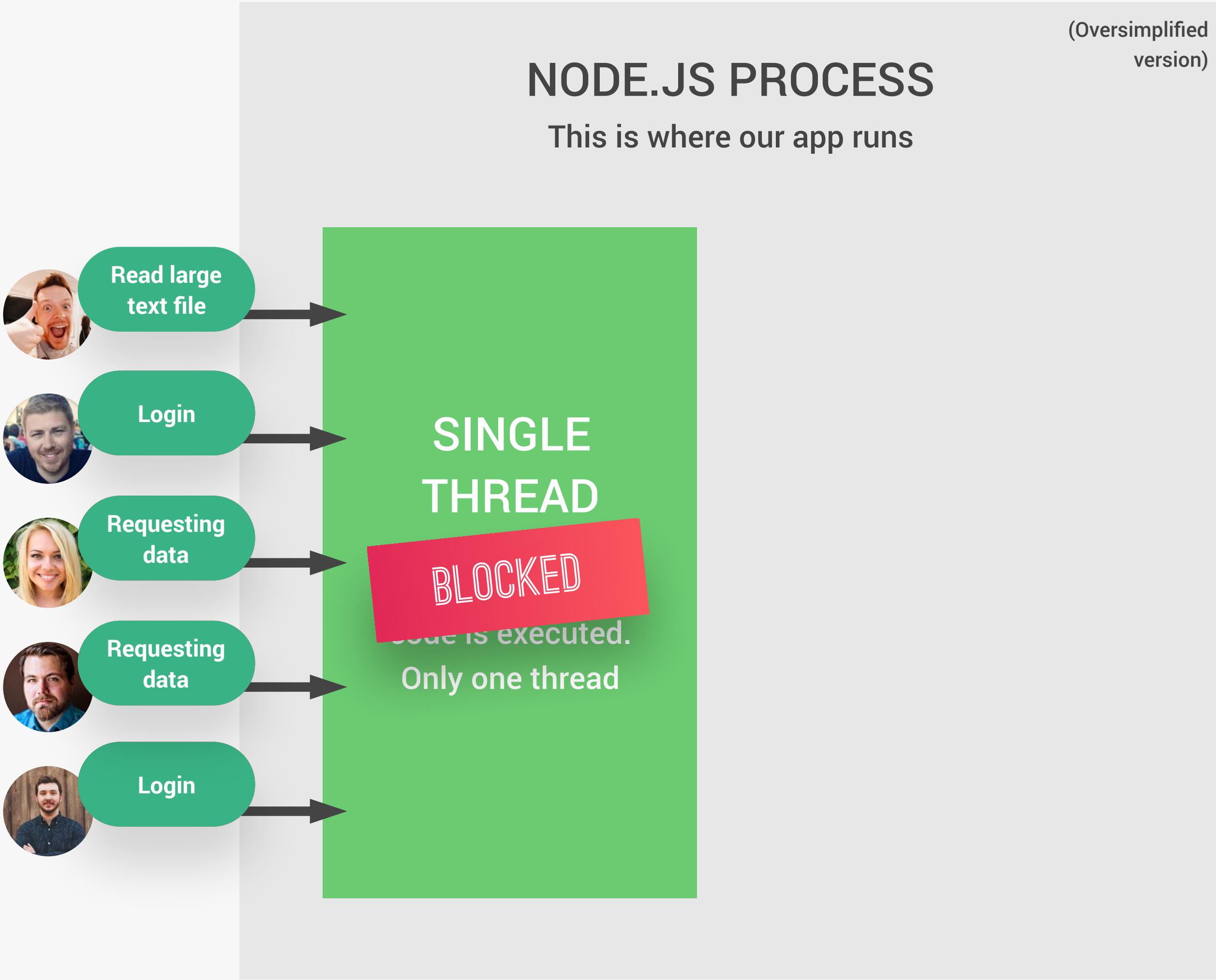
ASYNCHRONOUS



NON-BLOCKING



THE ASYNCHRONOUS NATURE OF NODE.JS: AN OVERVIEW



SYNCHRONOUS WAY

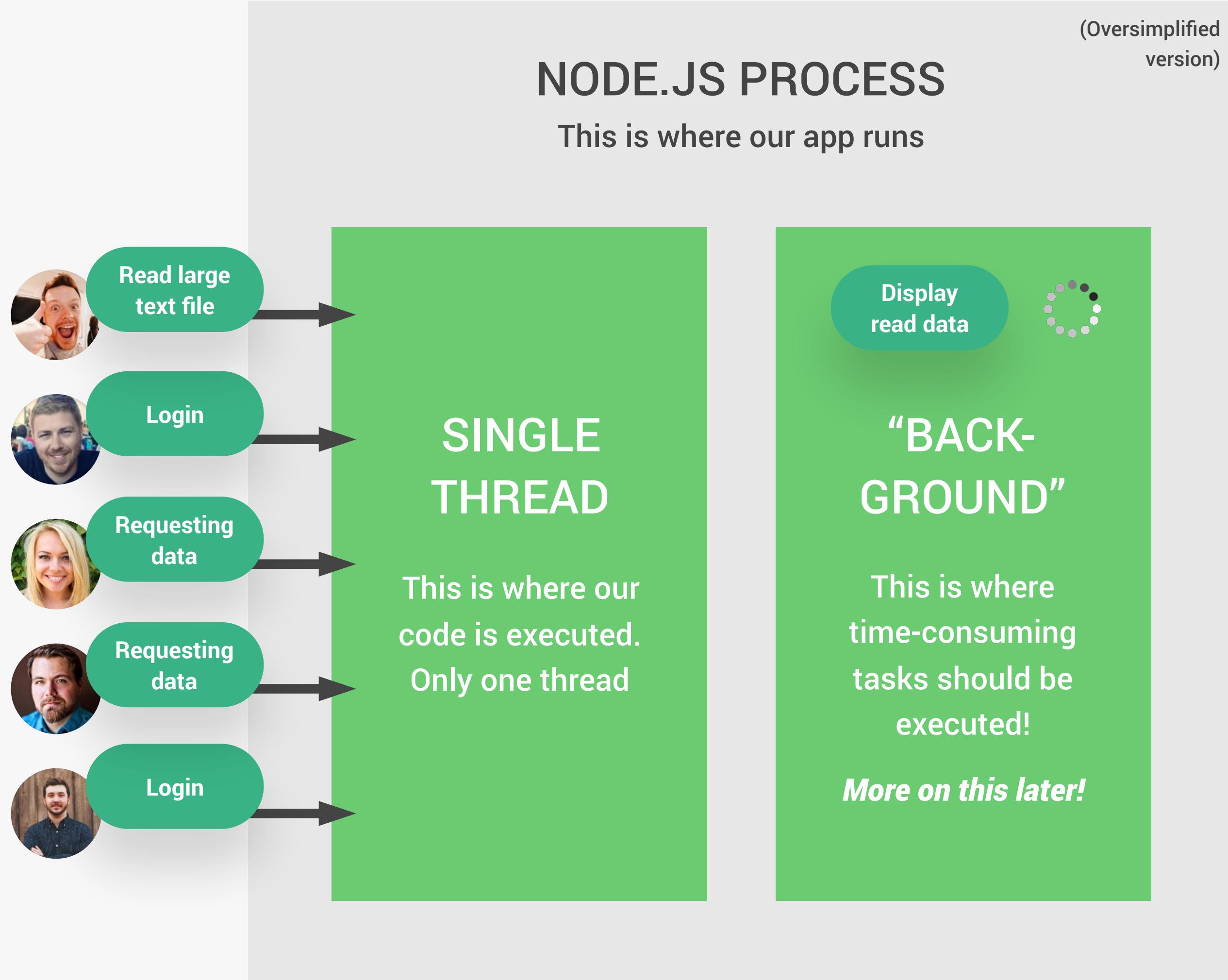


```
const fs = require('fs');

// 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



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

```
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

SECTION

INTRODUCTION TO BACK-END WEB
DEVELOPMENT

LECTURE

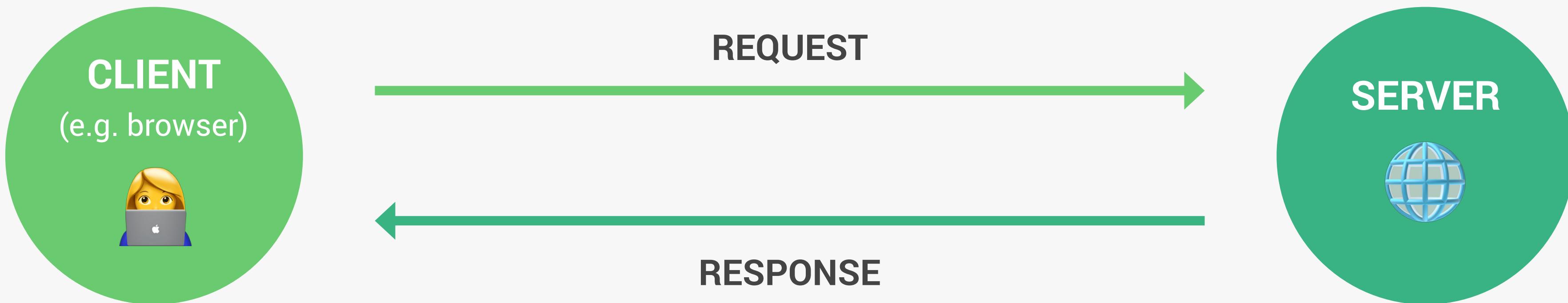
AN OVERVIEW OF HOW THE WEB WORKS



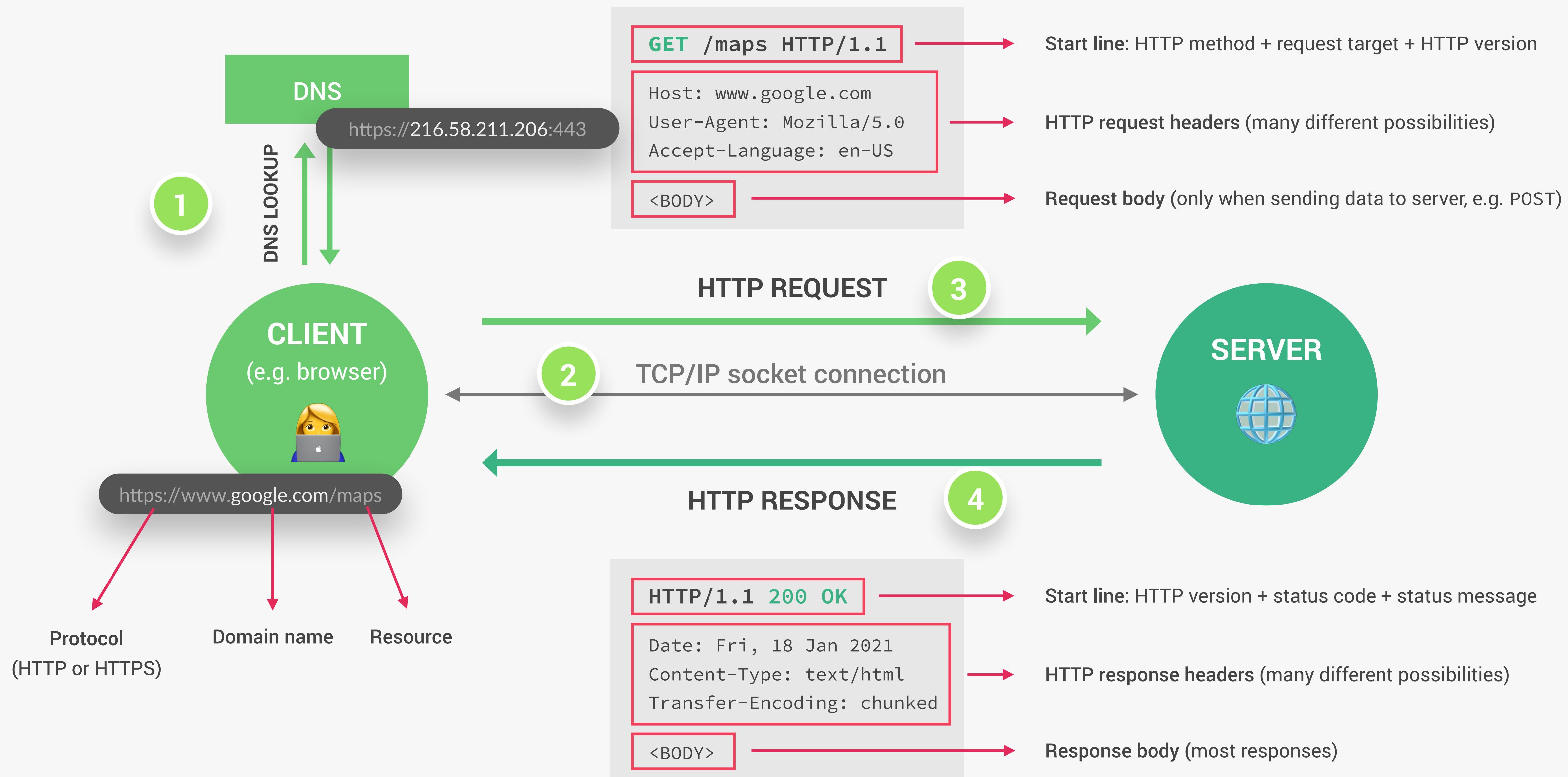
@JONASSCHMEDTMAN

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE

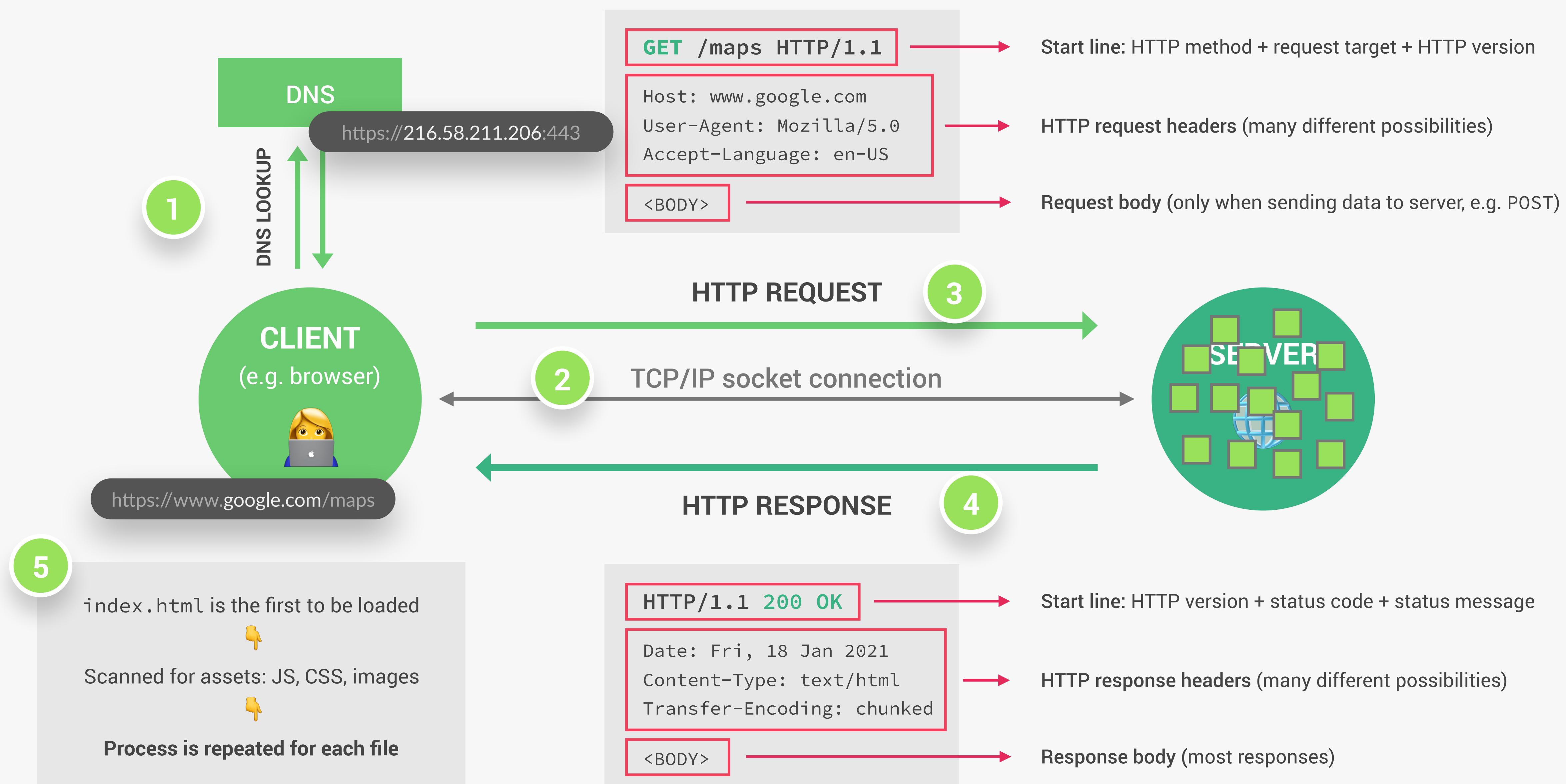
👉 Request-response model or Client-server architecture



WHAT HAPPENS WHEN WE ACCESS A WEBPAGE



WHAT HAPPENS WHEN WE ACCESS A WEBPAGE





JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

INTRODUCTION TO BACK-END WEB
DEVELOPMENT

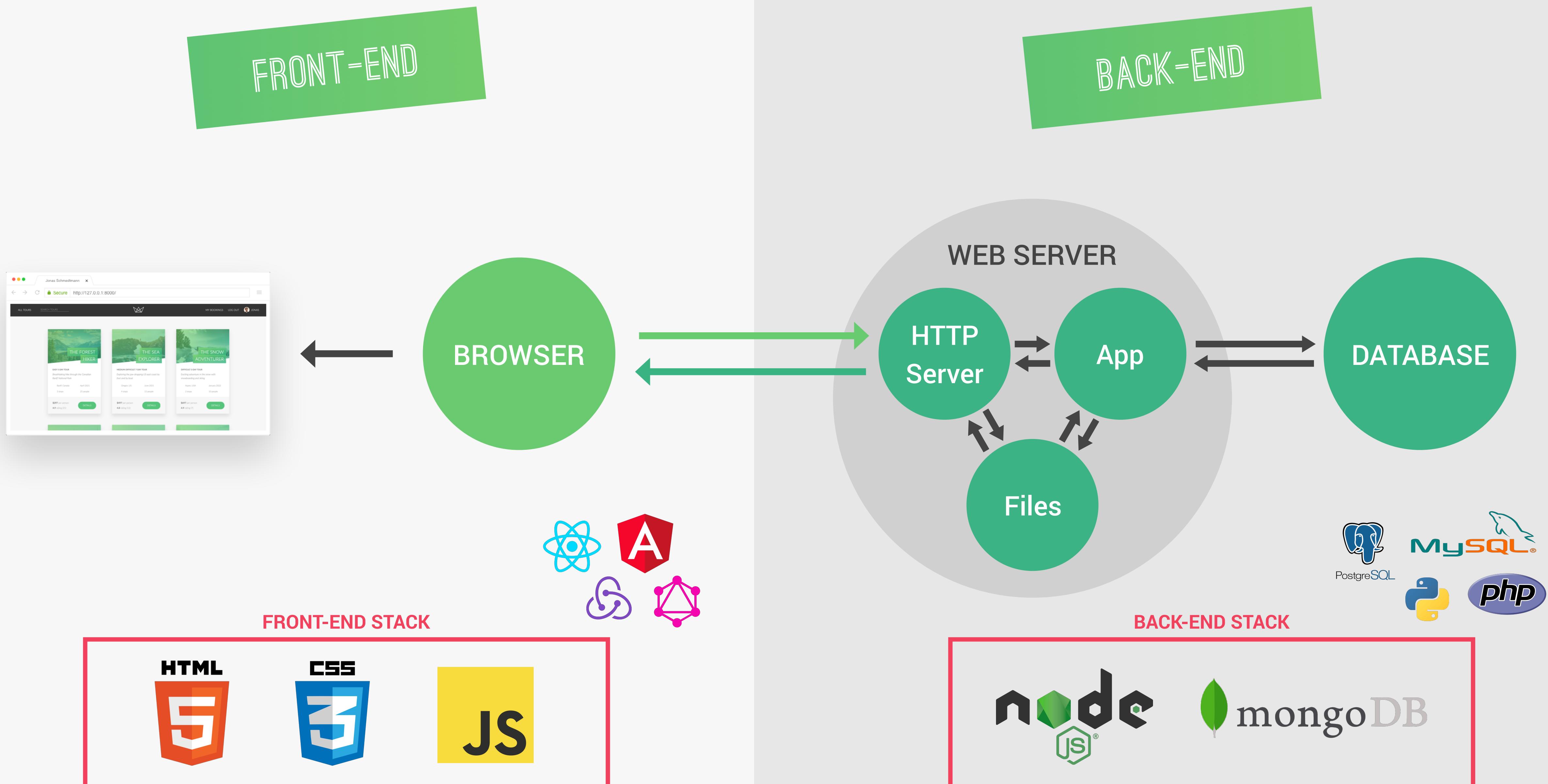
LECTURE

FRONT-END VS. BACK-END WEB
DEVELOPMENT



@JONASSCHMEDTMAN

FRONT-END AND BACK-END





JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

INTRODUCTION TO BACK-END WEB
DEVELOPMENT

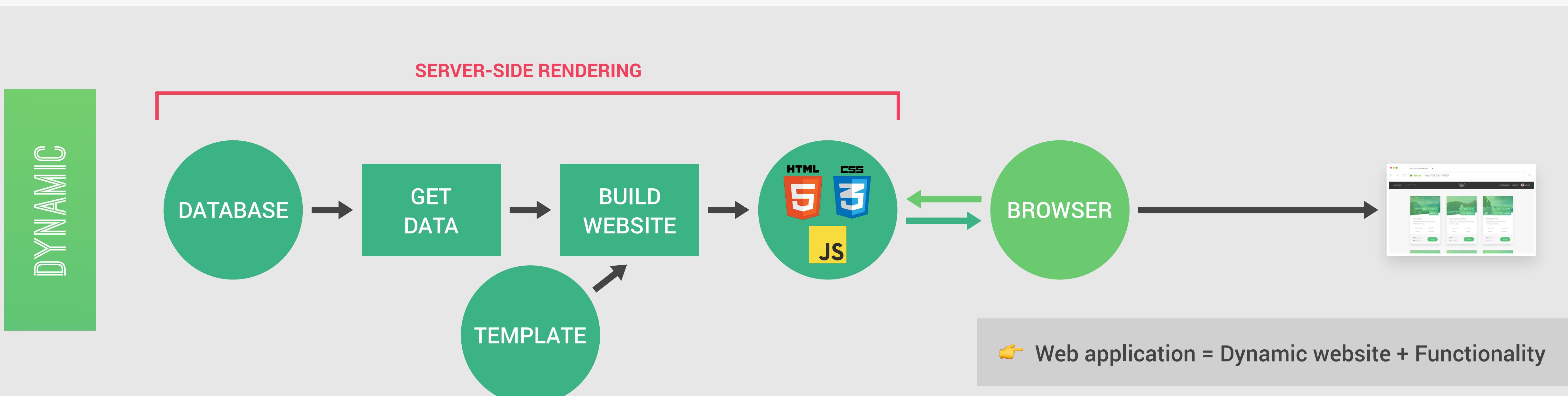
LECTURE

STATIC VS DYNAMIC VS API



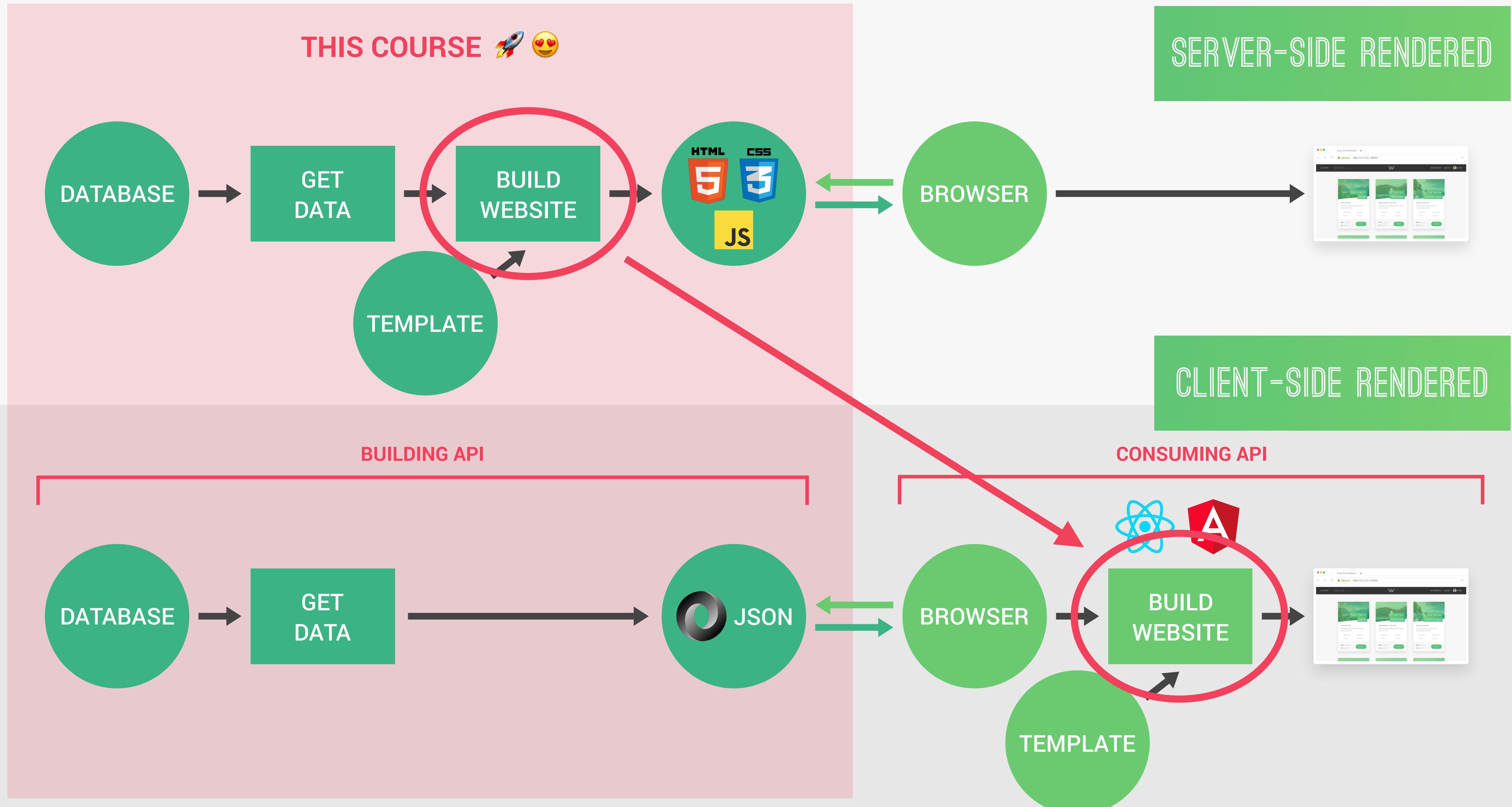
@JONASSCHMEDTMAN

STATIC WEBSITES VS DYNAMIC WEBSITES

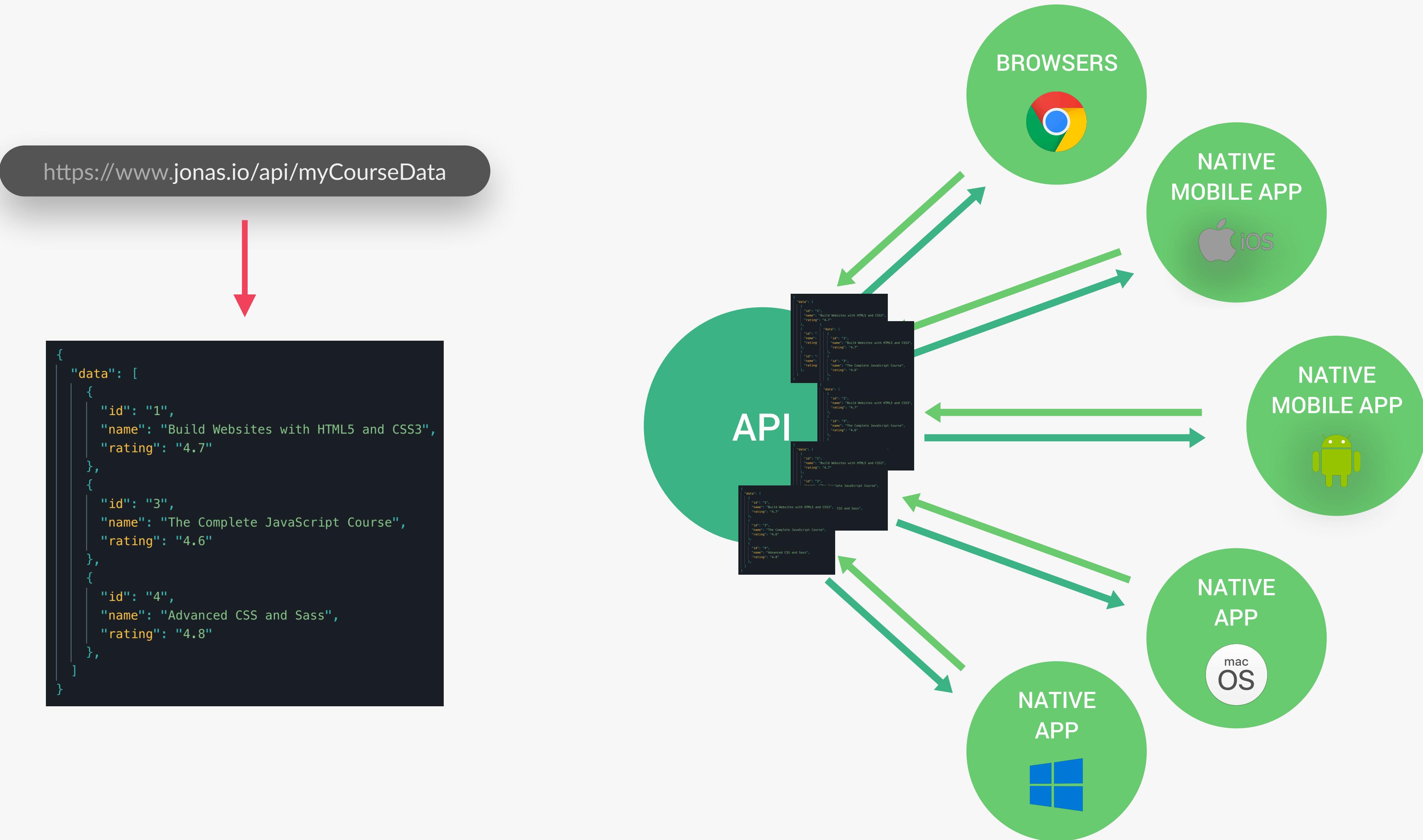


DYNAMIC WEBSITES VS API-POWERED WEBSITES

DYNAMIC



ONE API, MANY CONSUMERS



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



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

HOW NODE.JS WORKS: A LOOK BEHIND
THE SCENES

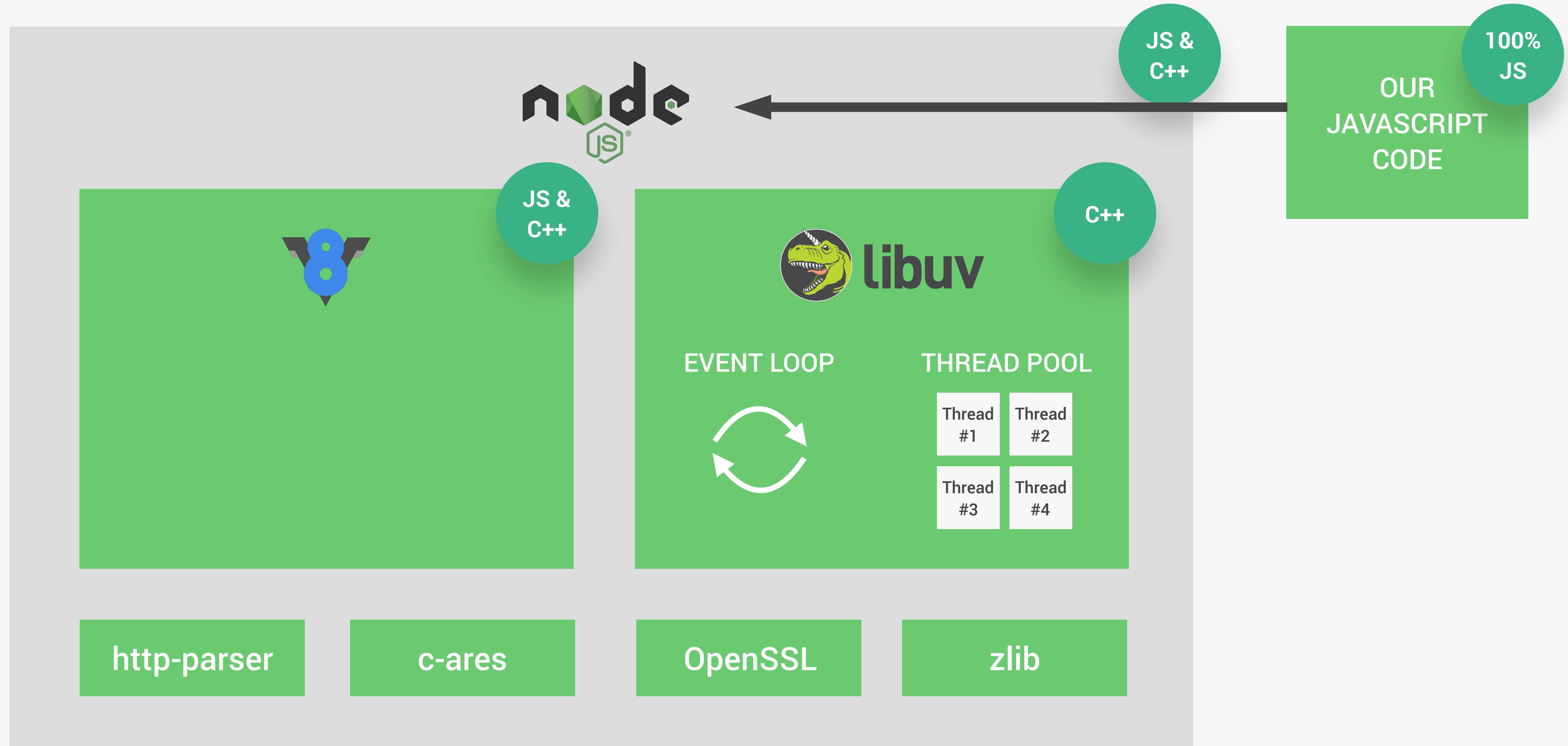
LECTURE

NODE, V8, LIBUV AND C++



@JONASSCHMEDTMAN

THE NODE.JS ARCHITECTURE BEHIND THE SCENES





JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

HOW NODE.JS WORKS: A LOOK BEHIND
THE SCENES

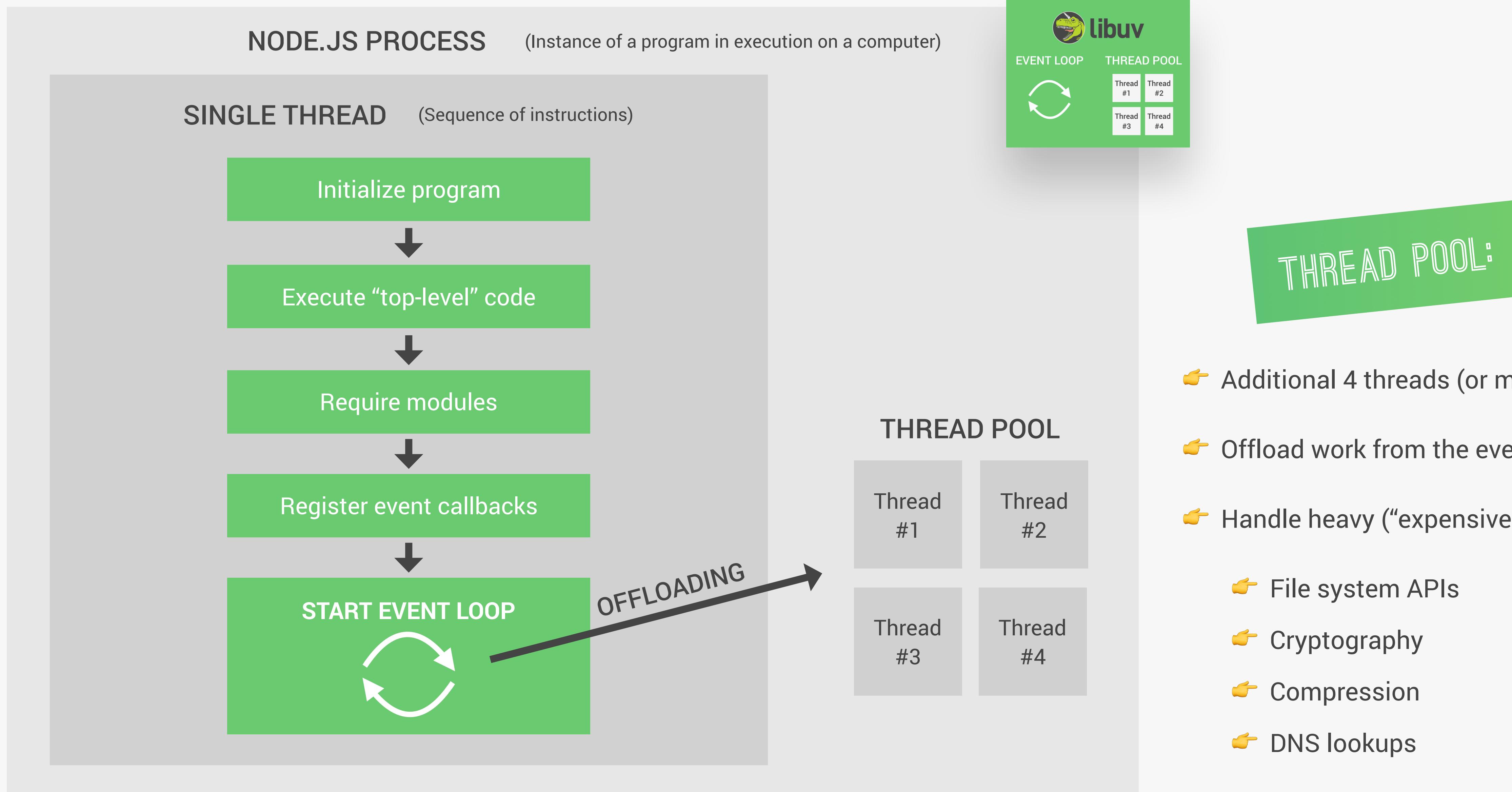
LECTURE

PROCESSES, THREADS AND THE THREAD
POOL



@JONASSCHMEDTMAN

NODE PROCESS AND THREADS





JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

HOW NODE.JS WORKS: A LOOK BEHIND
THE SCENES

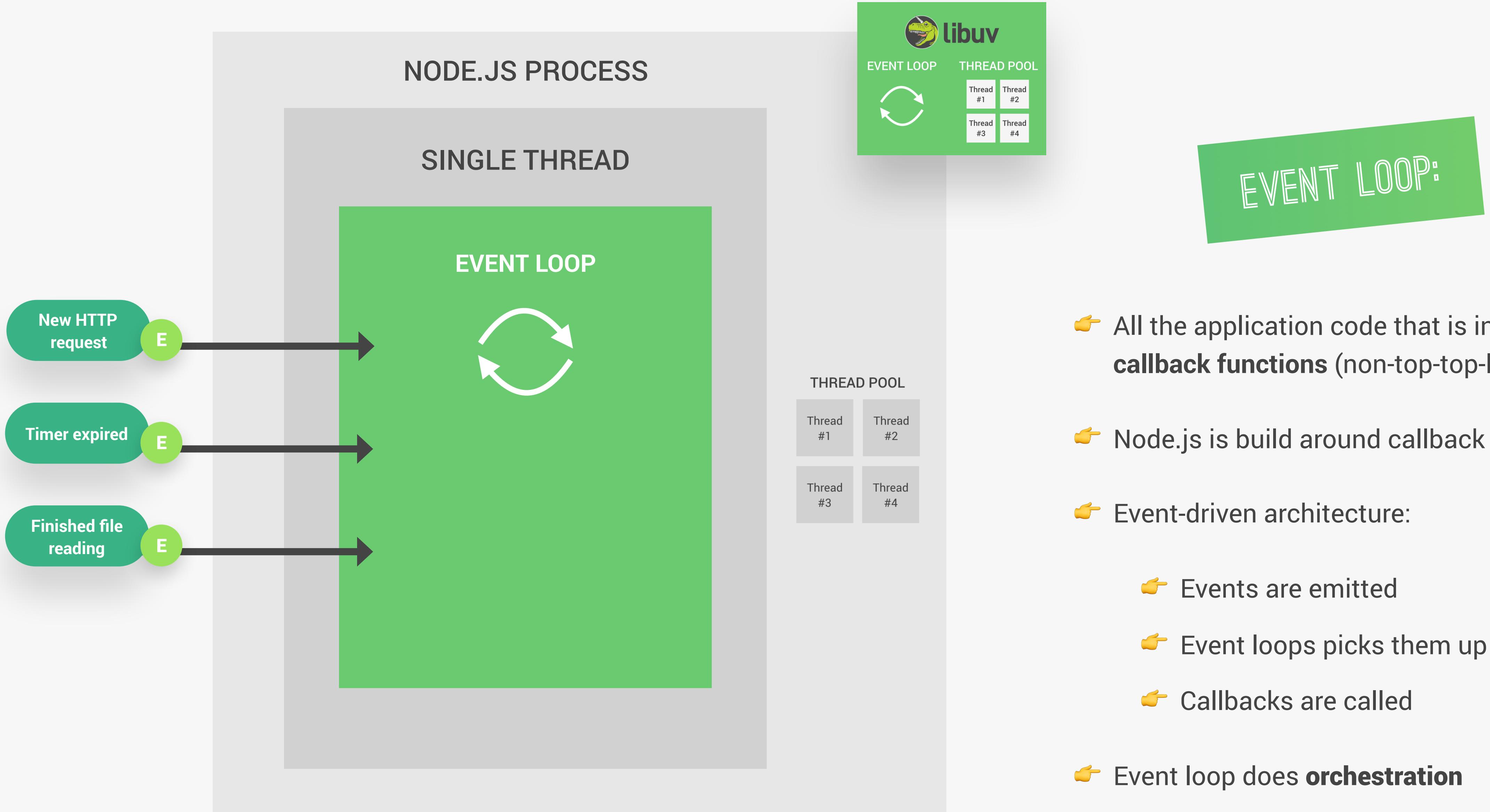
LECTURE

THE NODE.JS EVENT LOOP

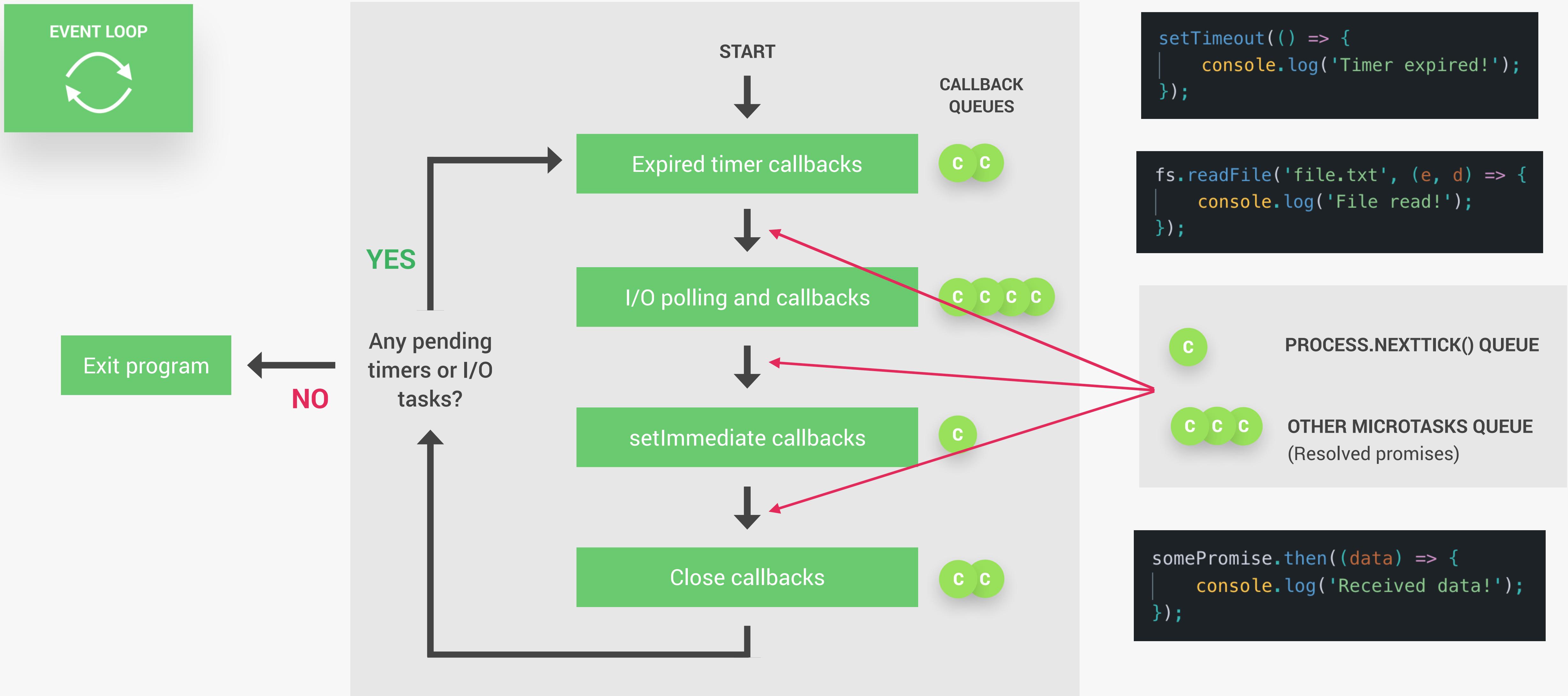


@JONASSCHMEDTMAN

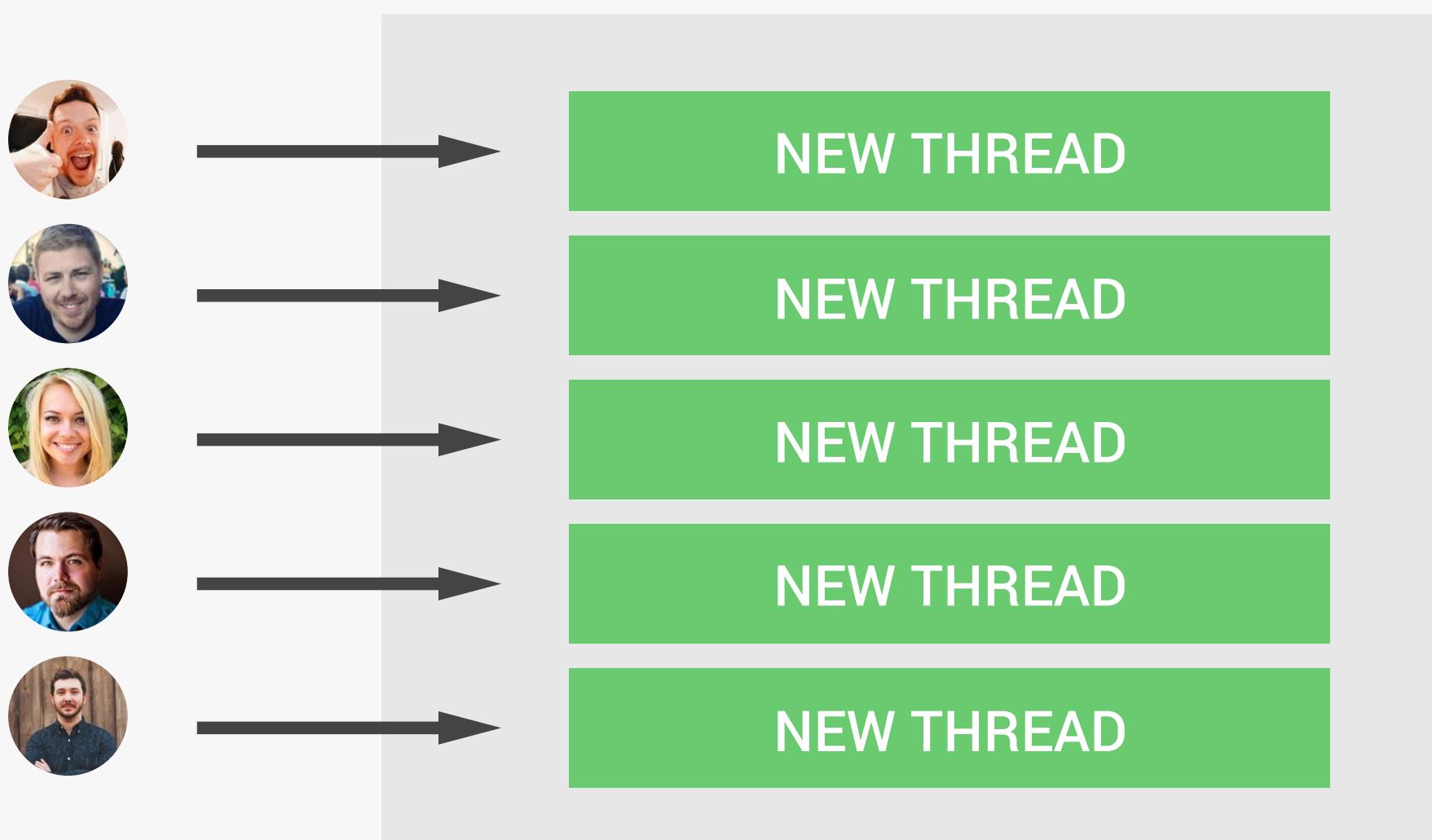
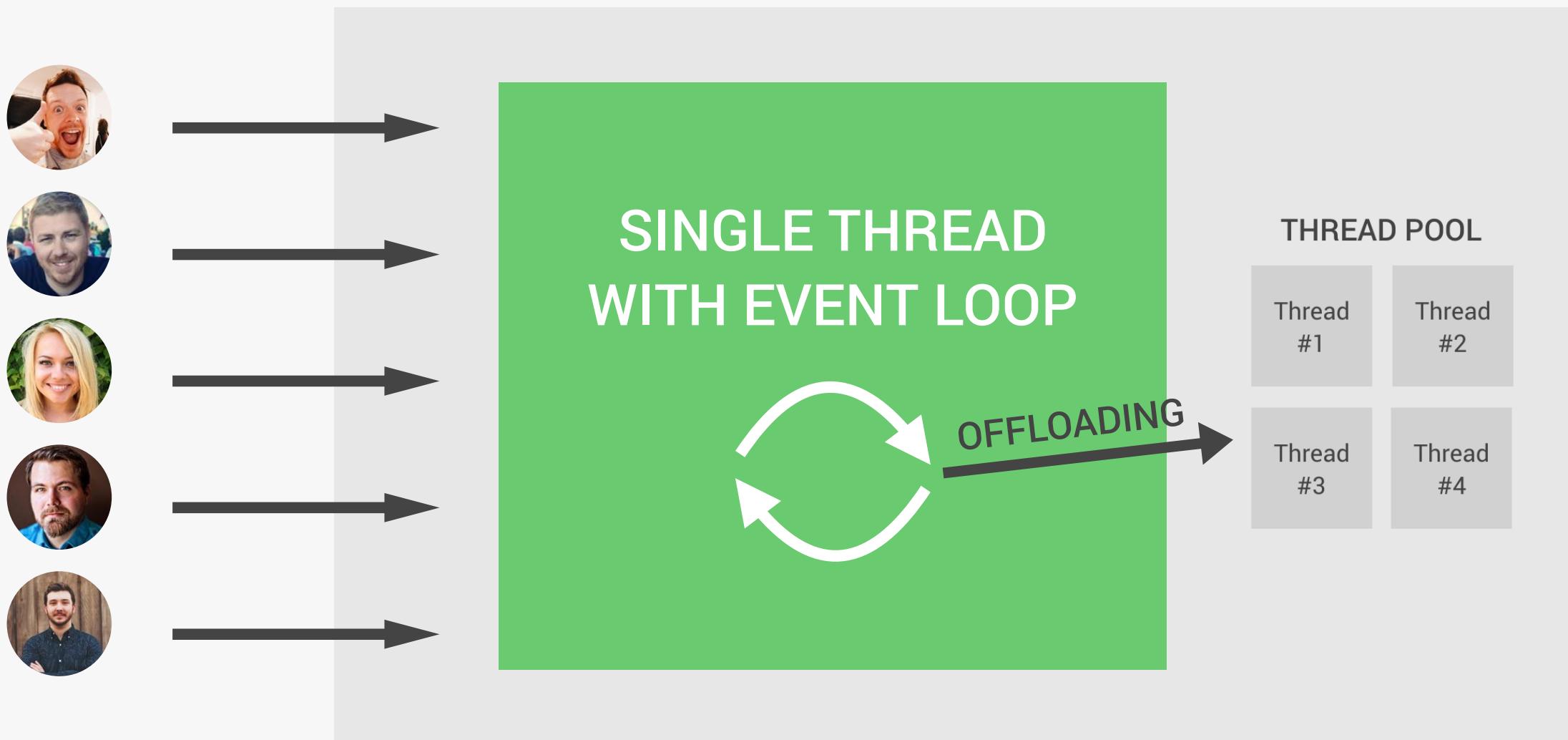
THE HEART OF NODE.JS: THE EVENT LOOP



THE EVENT LOOP IN DETAIL



SUMMARY OF THE EVENT LOOP: NODE VS. OTHERS



- 👉 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

SECTION

HOW NODE.JS WORKS: A LOOK BEHIND
THE SCENES

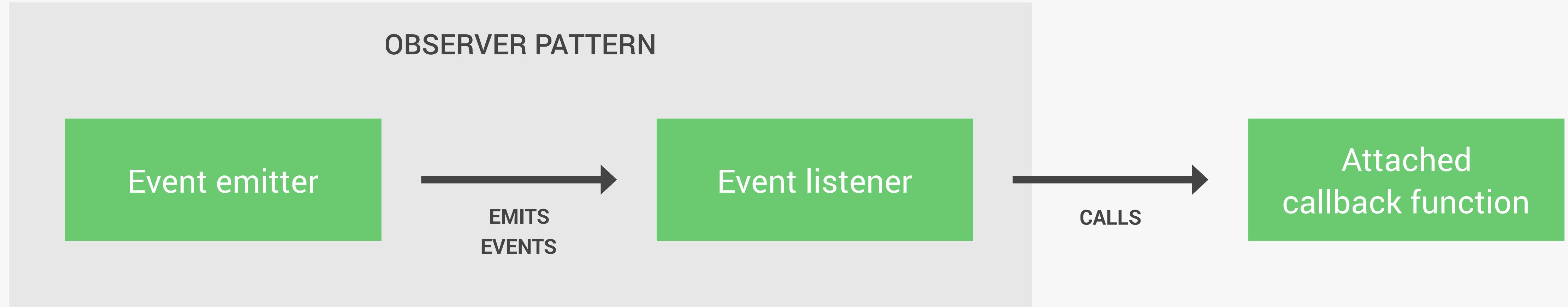
LECTURE

EVENTS AND EVENT-DRIVEN
ARCHITECTURE



@JONASSCHMEDTMAN

THE EVENT-DRIVEN ARCHITECTURE



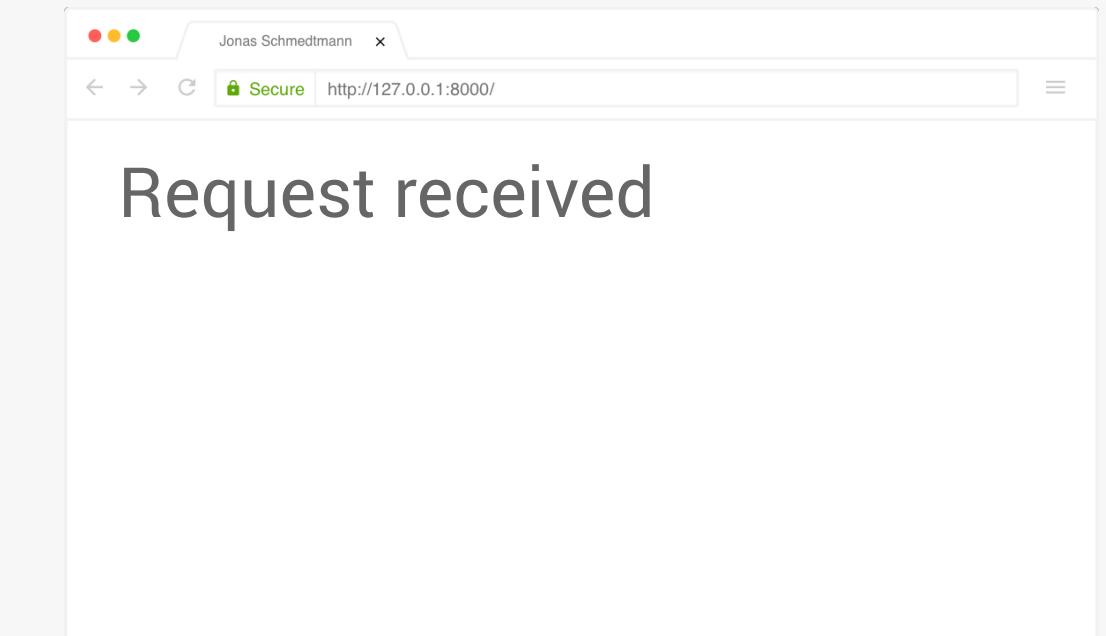
EMITTER

NEW REQUEST
ON SERVER
127.0.0.1:8000

'request'
event

LISTENER

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



👉 Instance of EventEmitter class



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

HOW NODE.JS WORKS: A LOOK BEHIND
THE SCENES

LECTURE

INTRODUCTION TO STREAMS



@JONASSCHMEDTMAN

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.



- 👉 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!

READABLE STREAMS

DESCRIPTION



Streams from which we can read (consume) data

WRITABLE STREAMS

Streams to which we can write data

DUPLEX STREAMS

Streams that are both readable and writable

TRANSFORM STREAMS

Duplex streams that transform data as it is written or read

EXAMPLE



- 👉 http requests
- 👉 fs read streams

- 👉 http responses
- 👉 fs write streams

- 👉 net web socket

- 👉 zlib Gzip creation

IMPORTANT EVENTS



- 👉 data
- 👉 end

- 👉 drain
- 👉 finish

CONSUME STREAMS

IMPORTANT FUNCTIONS



- 👉 `pipe()`
- 👉 `read()`

- 👉 `write()`
- 👉 `end()`



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

HOW NODE.JS WORKS: A LOOK BEHIND
THE SCENES

LECTURE

HOW REQUIRING MODULES REALLY WORKS



@JONASSCHMEDTMAN

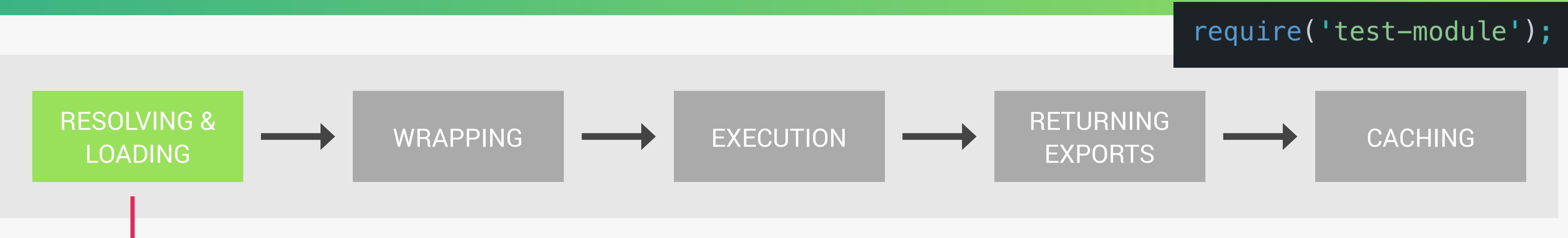
THE COMMONJS MODULE SYSTEM

- 👉 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



👉 Core modules

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

👉 Developer modules

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

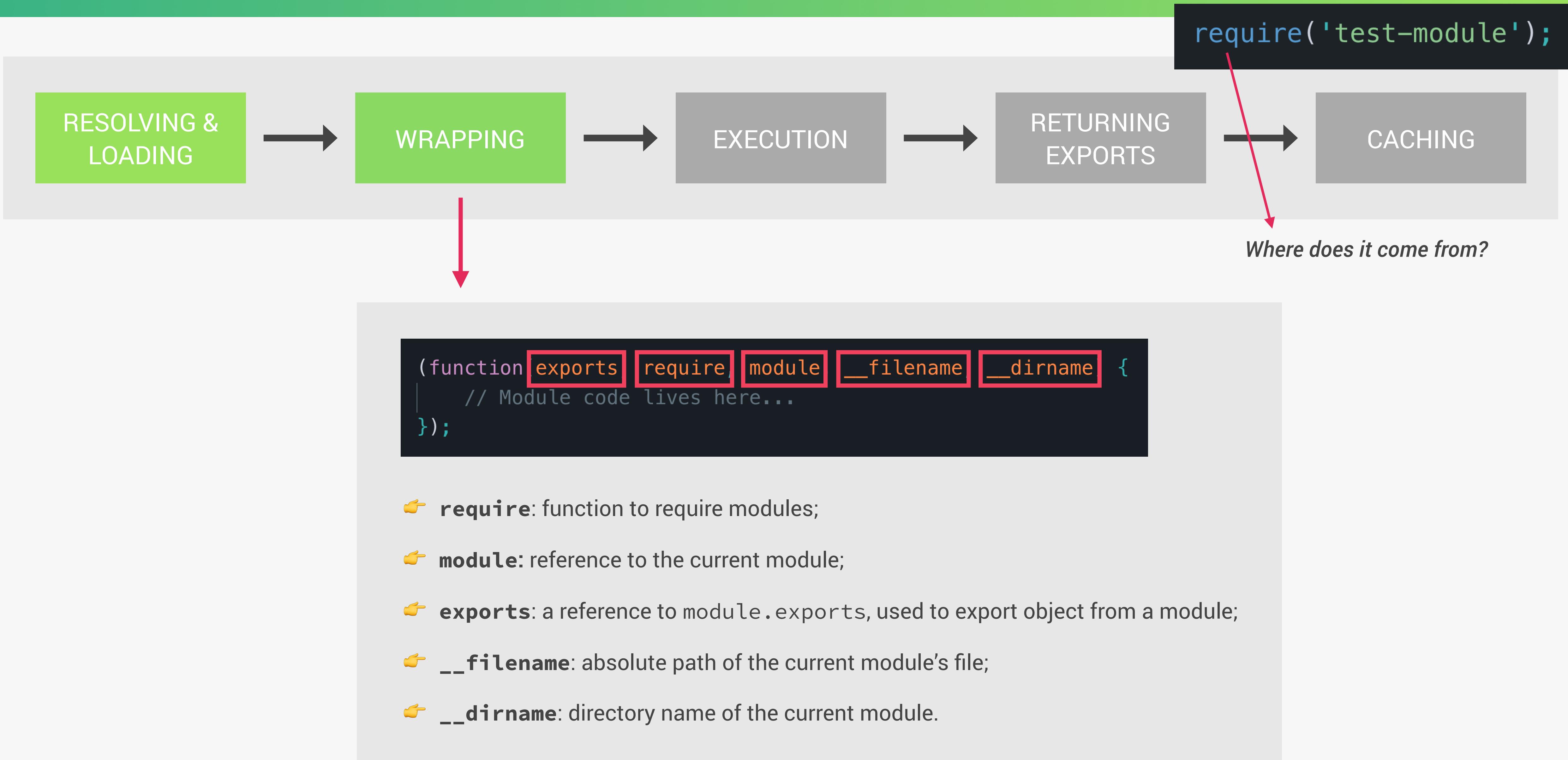
👉 3rd-party modules (from NPM)

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

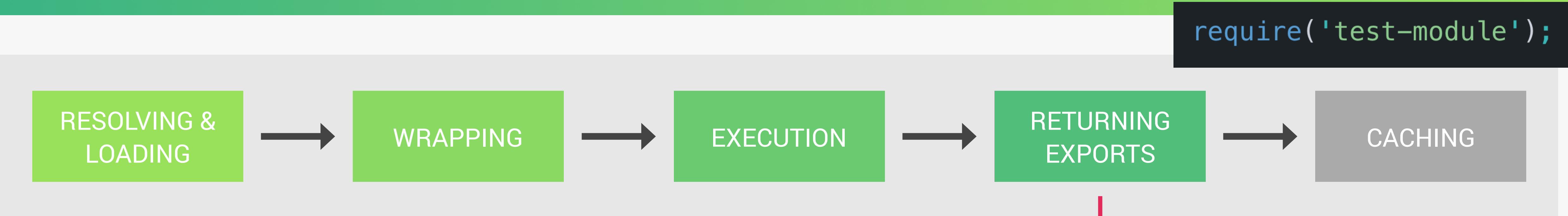
PATH RESOLVING: HOW NODE DECIDES WHICH MODULE TO LOAD

- 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.

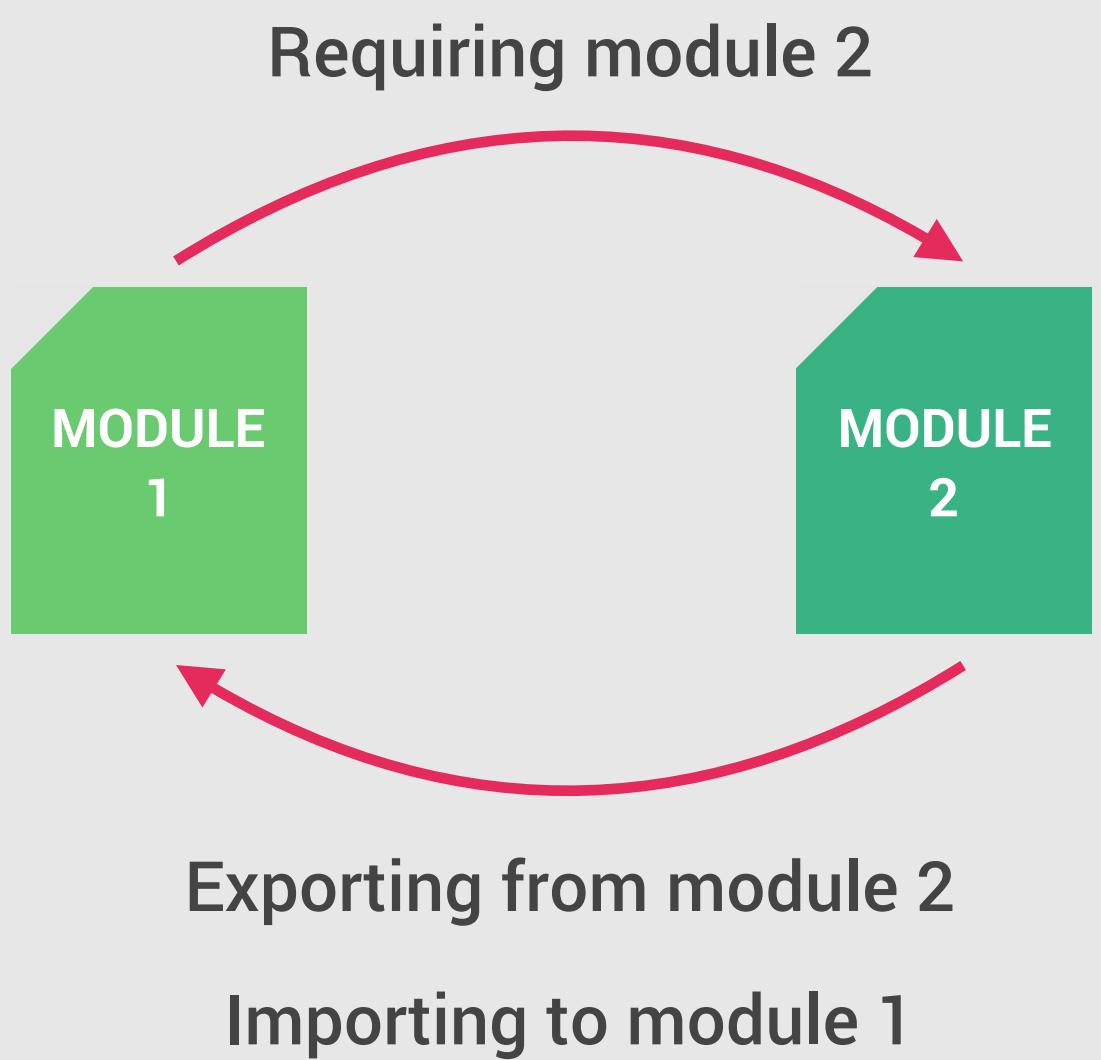
WHAT HAPPENS WHEN WE REQUIRE() A MODULE



WHAT HAPPENS WHEN WE REQUIRE() A 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');
```

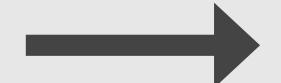
RESOLVING &
LOADING

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

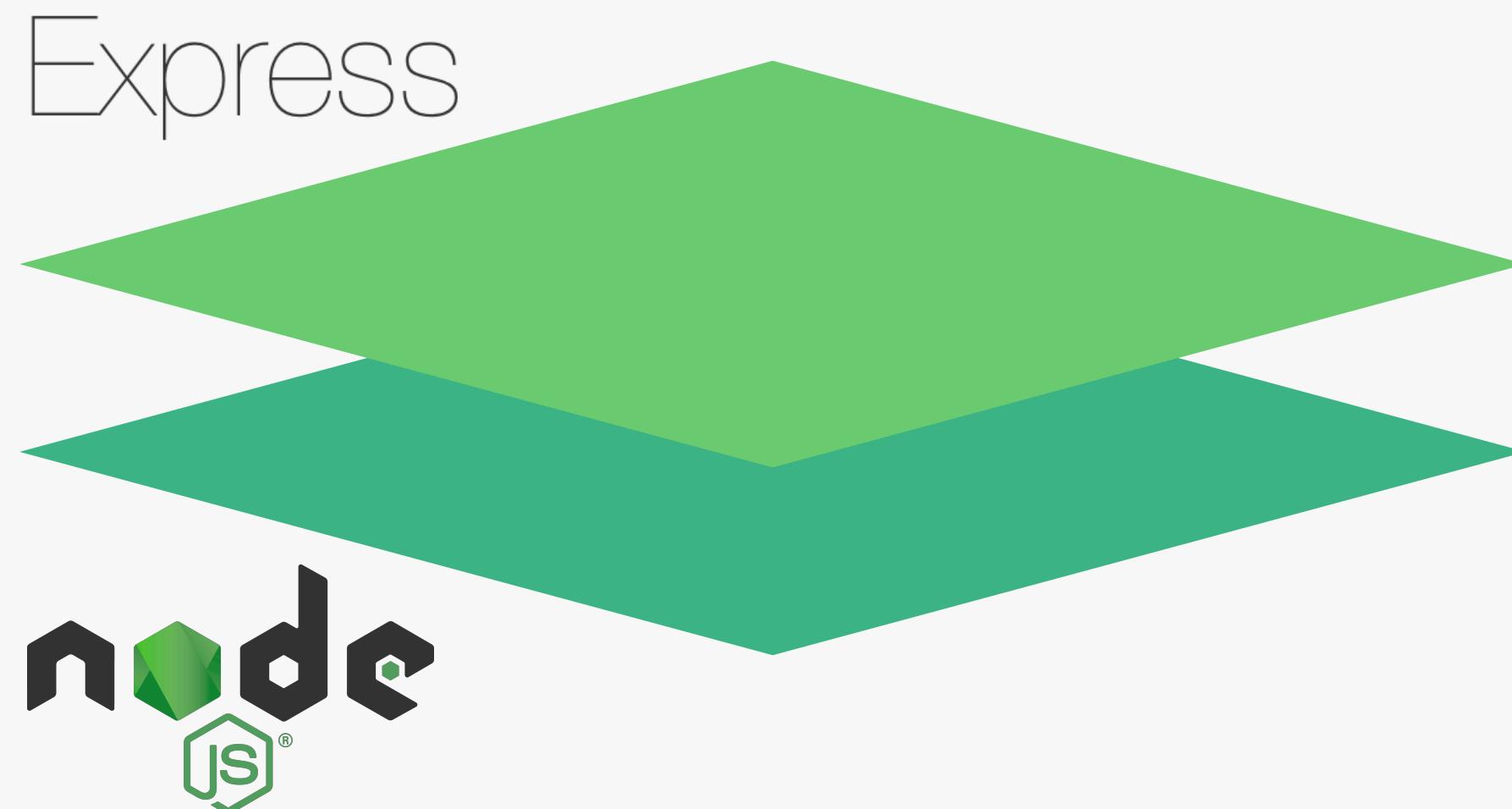
SECTION

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.

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

EXPRESS: LET'S START BUILDING THE
NATOURS API!

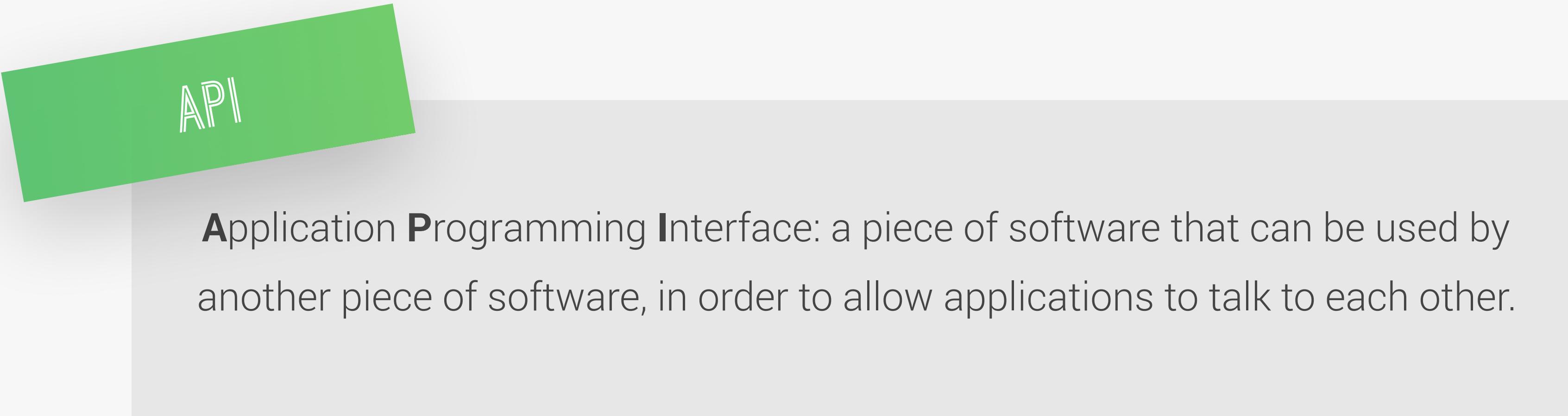
LECTURE

APIS AND RESTFUL API DESIGN

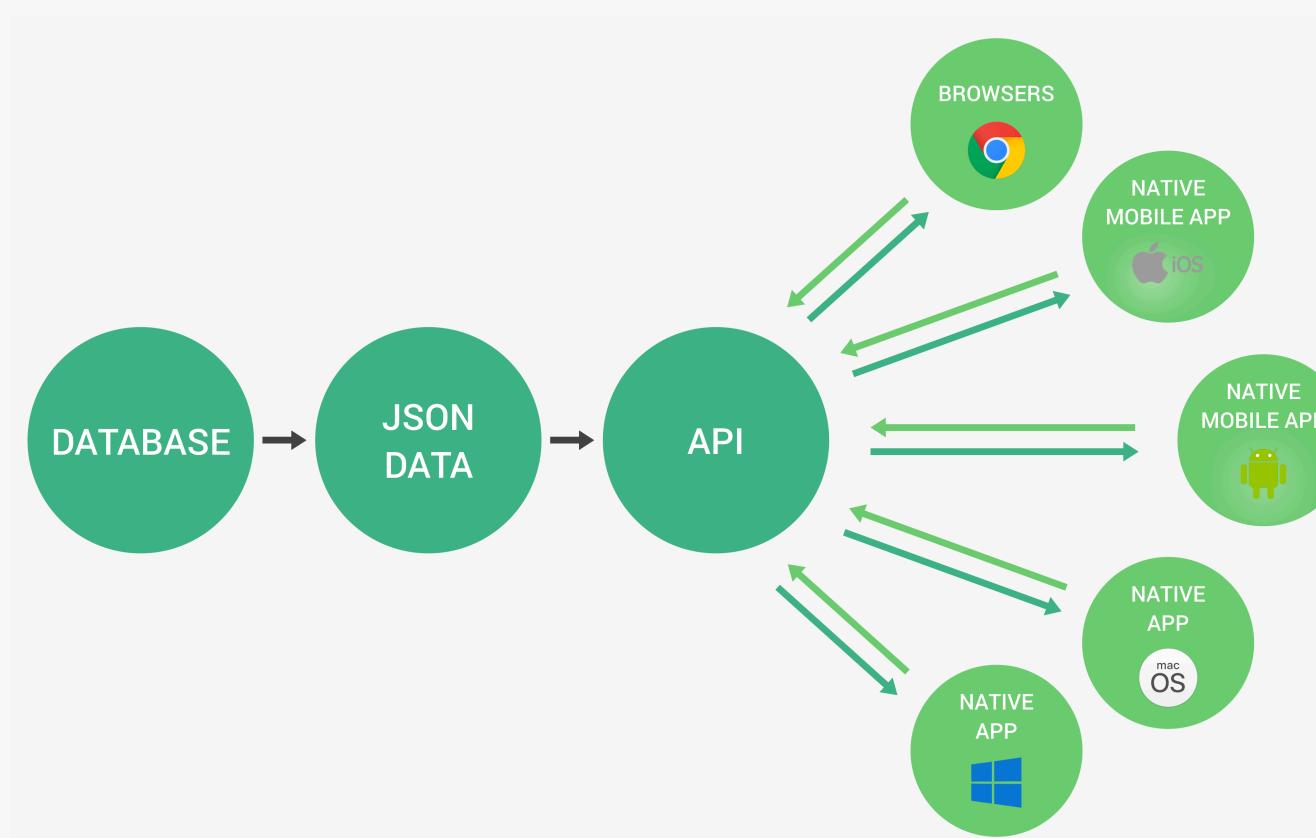


@JONASSCHMEDTMAN

WHAT IS AN API ANYWAY?



👉 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

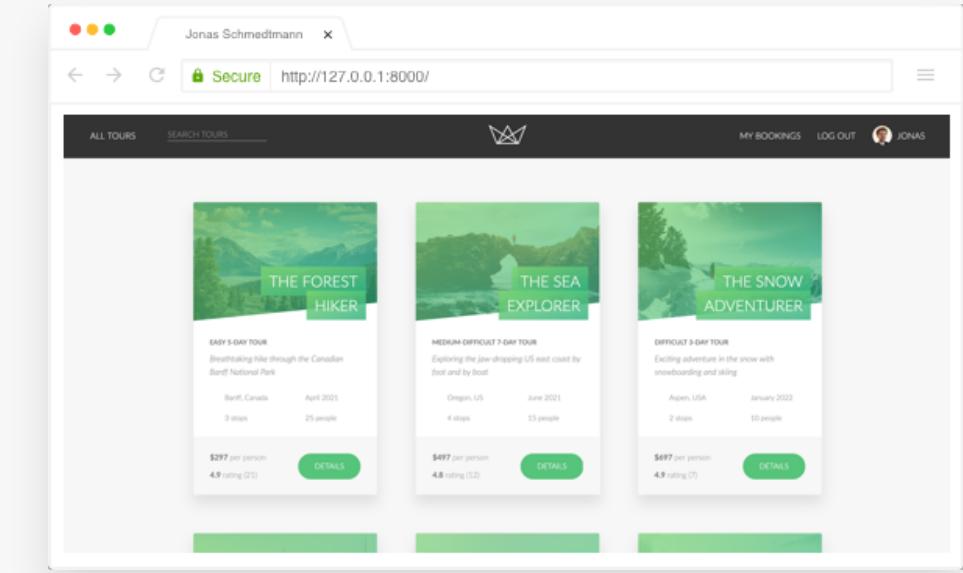
Separate API into logical resources

👉 **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



2

Expose structured, **resource-based URLs**

3

Use **HTTP methods** (verbs)

4

Send data as **JSON** (usually)

5

Be stateless

URL

https://www.natours.com/addNewTour

ENDPOINT

/getTour

/updateTour

BAD

/getToursByUser

/deleteToursByUser

👉 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

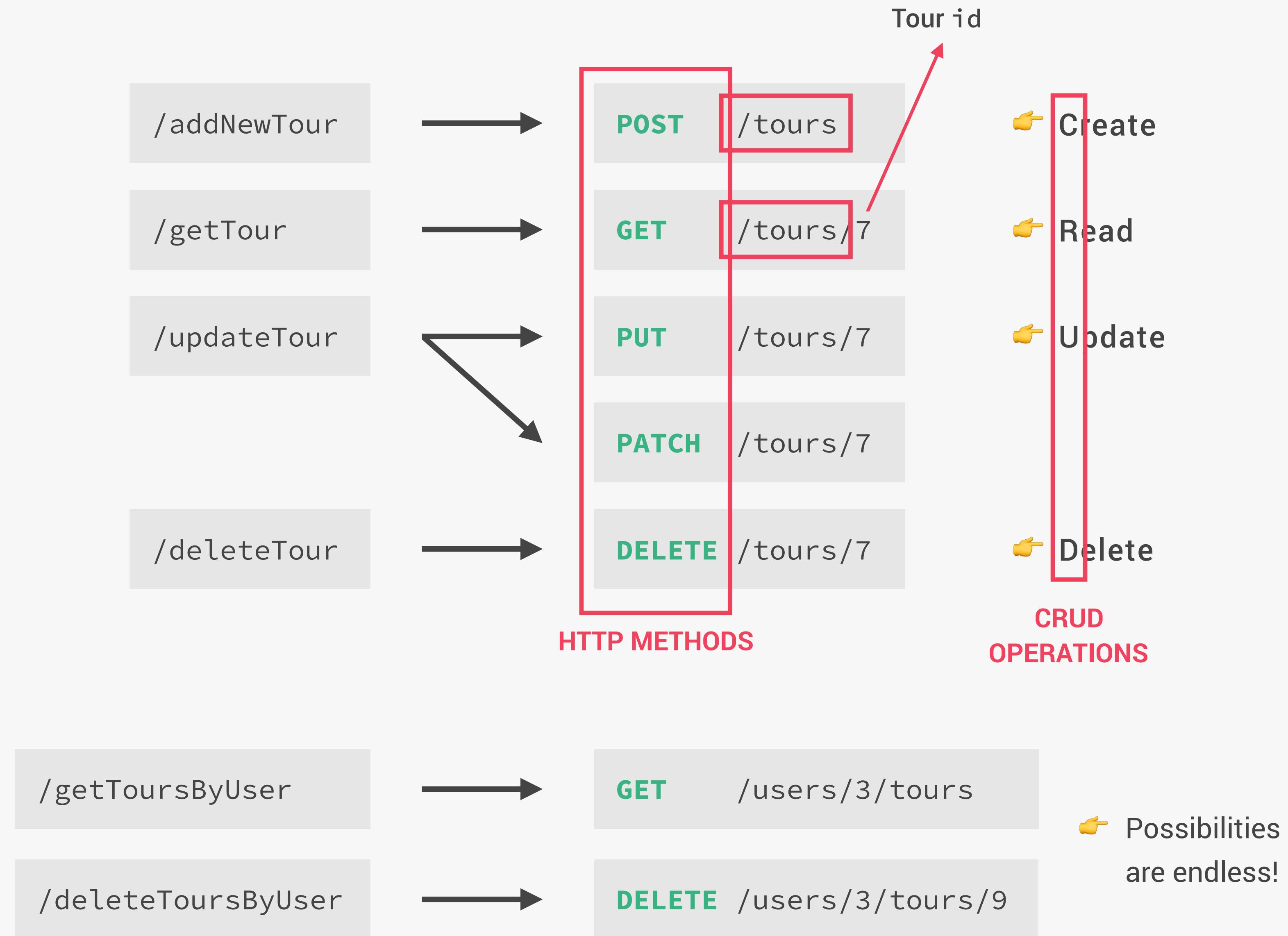
Use **HTTP methods** (verbs)

4

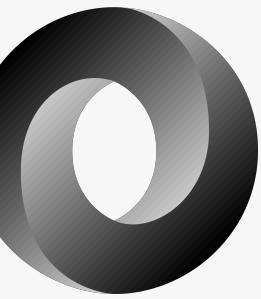
Send data as JSON (usually)

5

Be stateless



THE REST ARCHITECTURE



1

Separate API into logical resources

2

Expose structured, resource-based URLs

3

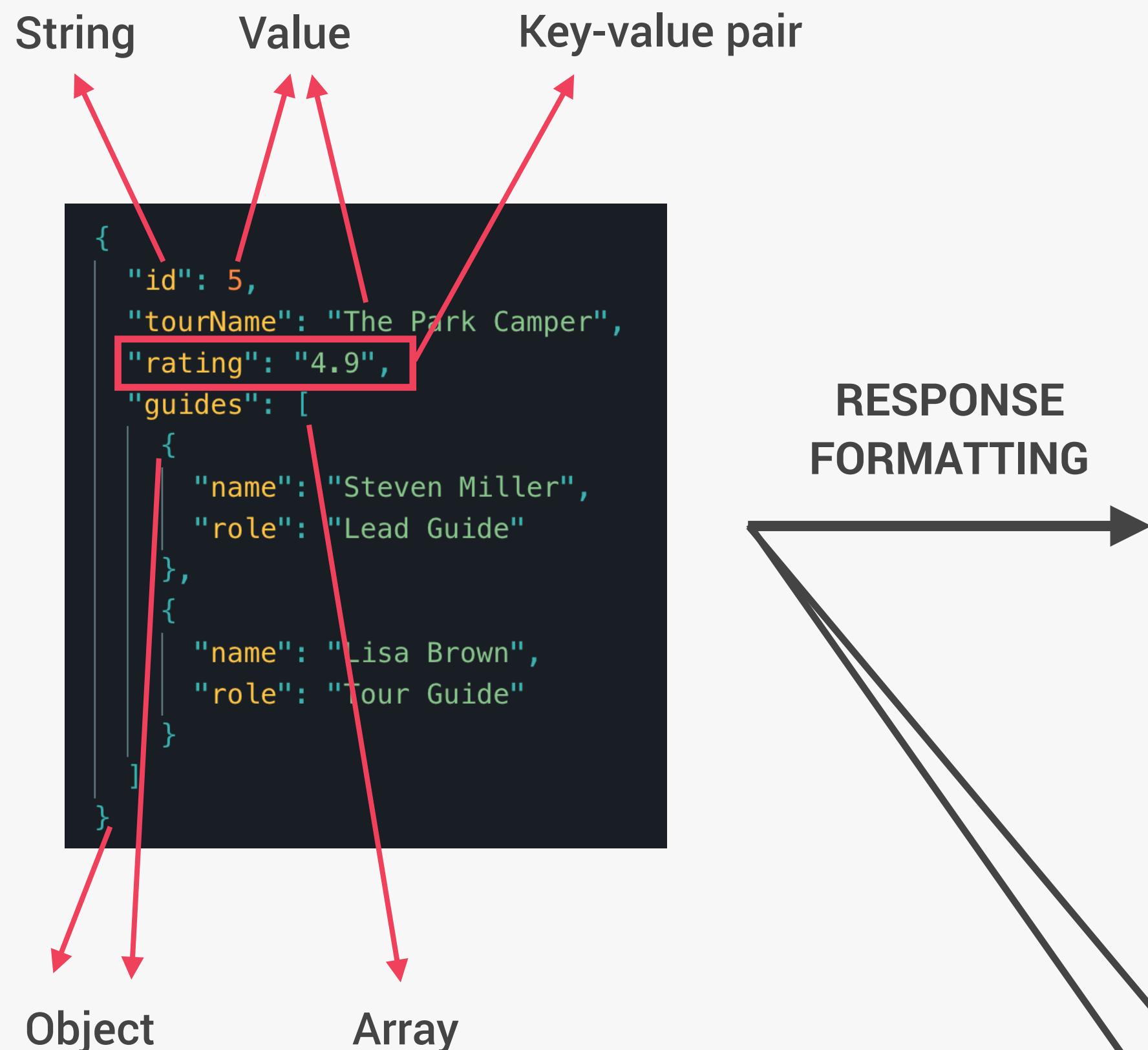
Use HTTP methods (verbs)

4

Send data as JSON (usually)

5

Be stateless



RESPONSE
FORMATTING

👉 JSend

```
{  
  "status": "success",  
  "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

BAD



WEB SERVER

STATE ON SERVER

nextPage = currentPage + 1
send(nextPage)

GET /tours/page/6

WEB SERVER

send(6)

STATE COMING FROM CLIENT



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP



@JONASSCHMEDTMAN

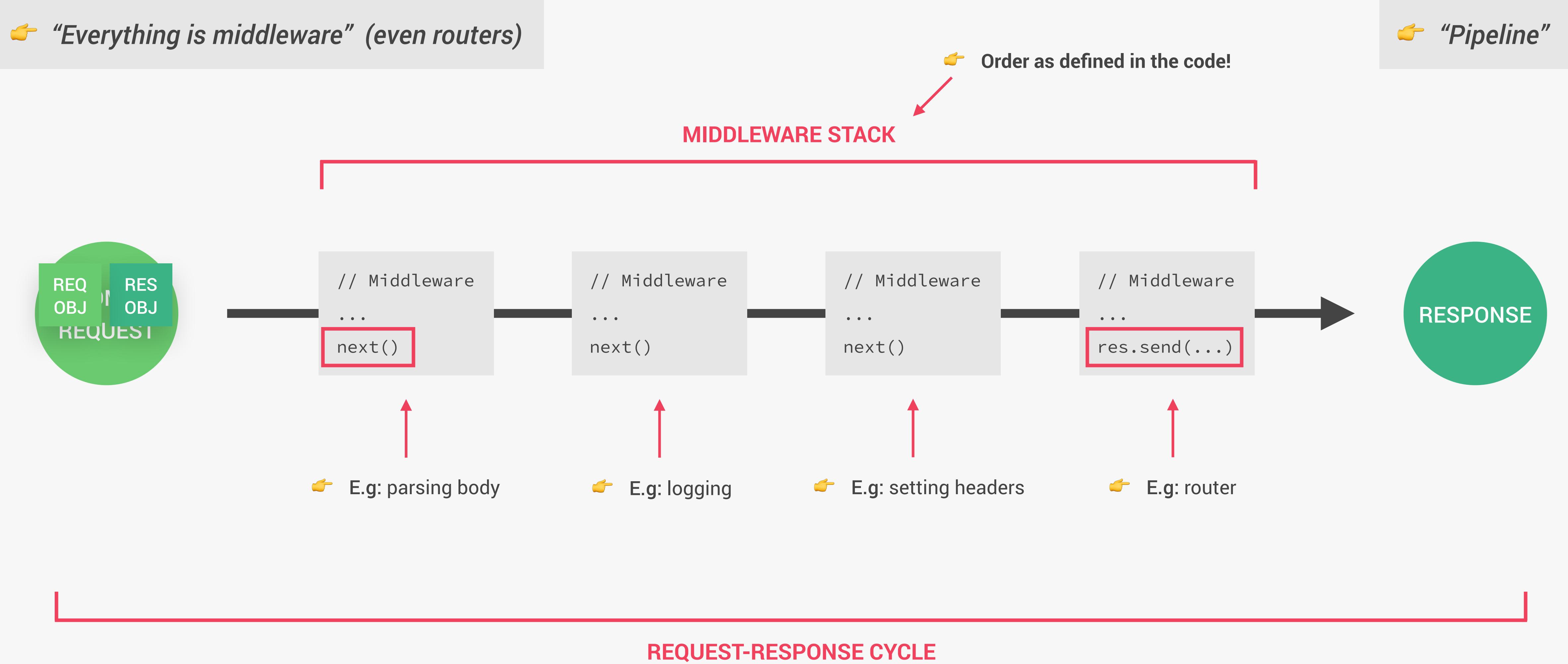
SECTION

EXPRESS: LET'S START BUILDING THE
NATOURS API!

LECTURE

MIDDLEWARE AND THE REQUEST-
RESPONSE CYCLE

THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE



SECTION 7 – INTRODUCTION TO MONGODB



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

INTRODUCTION TO MONGODB

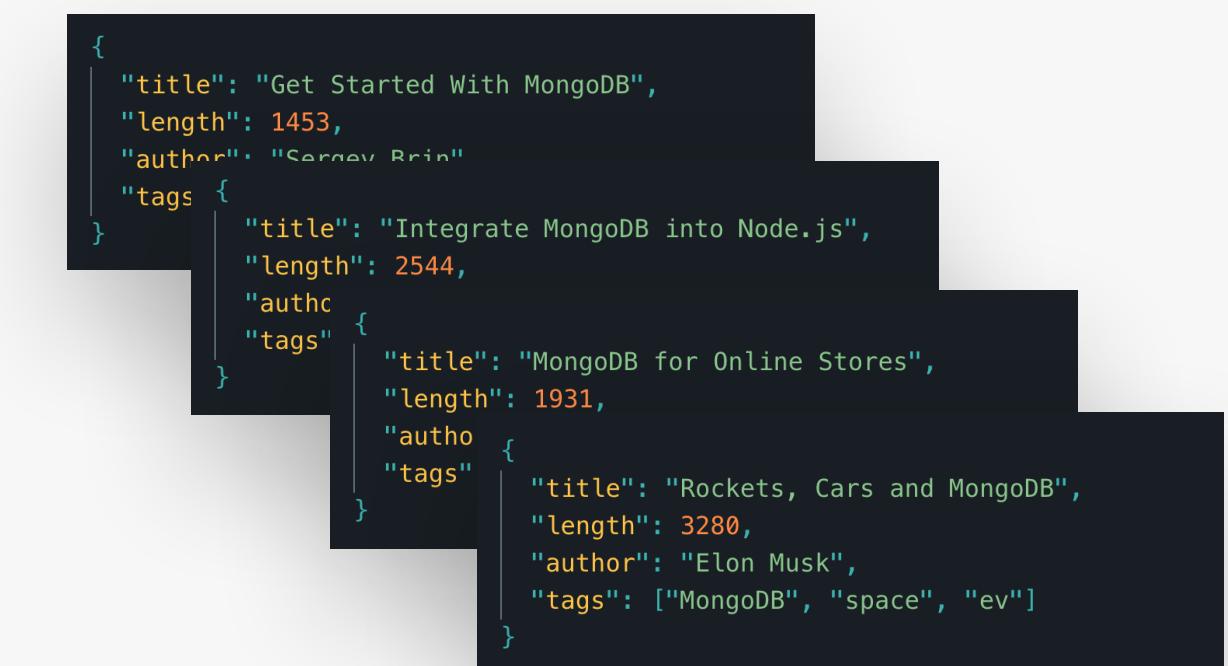
LECTURE

WHAT IS MONGODB?



@JONASSCHMEDTMAN

MONGODB: AN OVERVIEW



DATABASE

COLLECTIONS

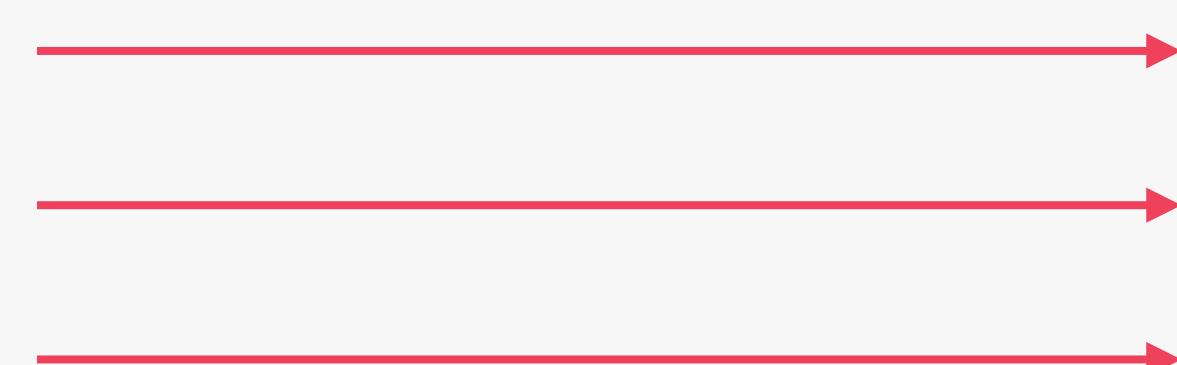
("Tables")

DOCUMENTS

("Rows")

👉 NoSQL

blog
users
reviews



post
user
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" }  
  ]  
}
```

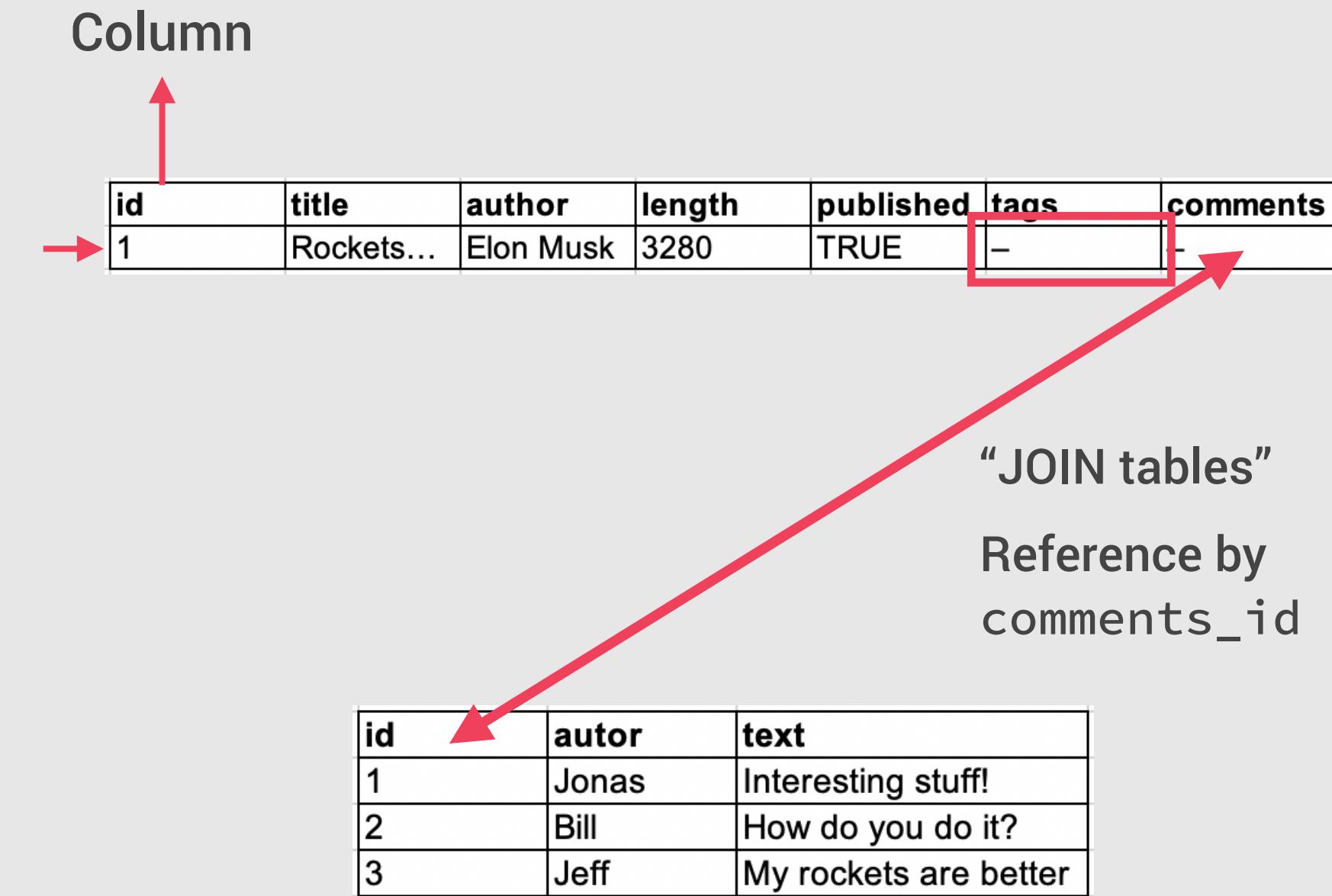
Unique ID

Fields

Embedded documents

Values (*typed*)

RELATIONAL DATABASE



- 👉 **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).

- 👉 **Data is always normalized**

SECTION 8 –

USING MONGODB WITH

MONGOOSE



JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

USING MONGODB WITH MONGOOSE

LECTURE

WHAT IS MONGOOSE?

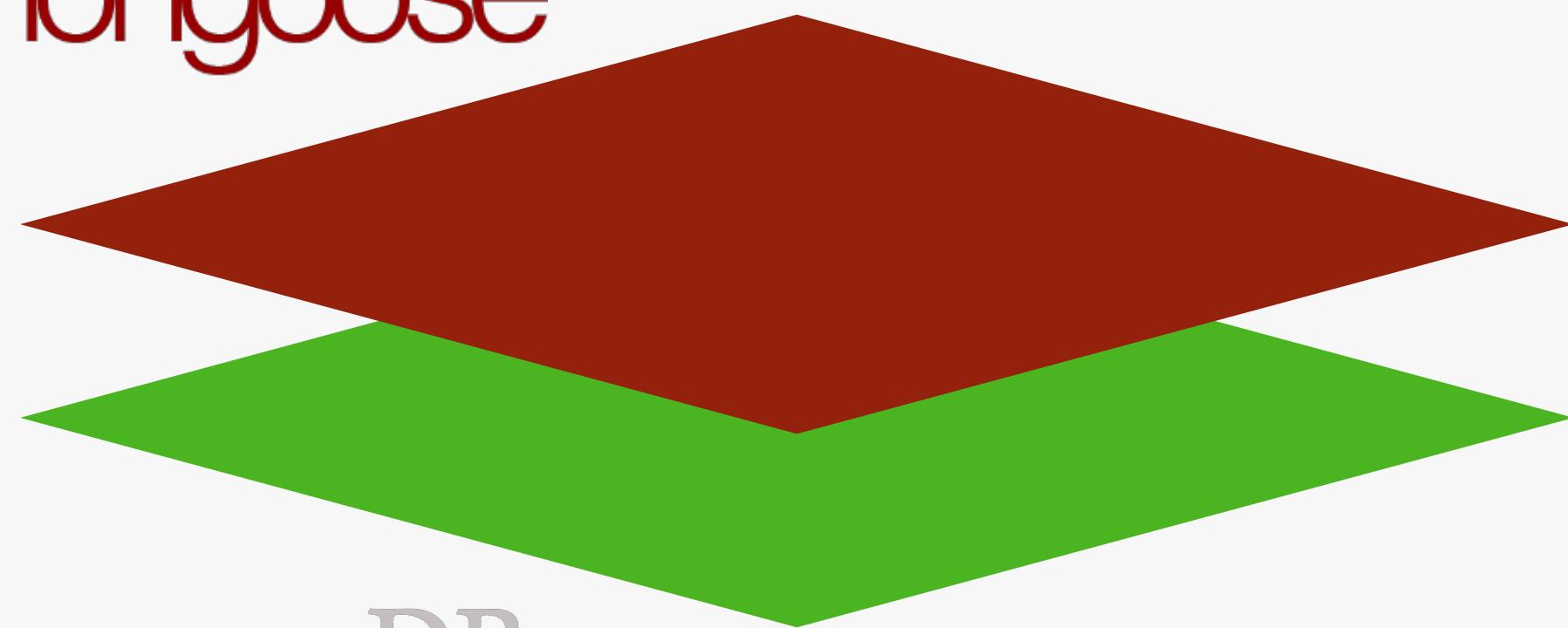


@JONASSCHMEDTMAN

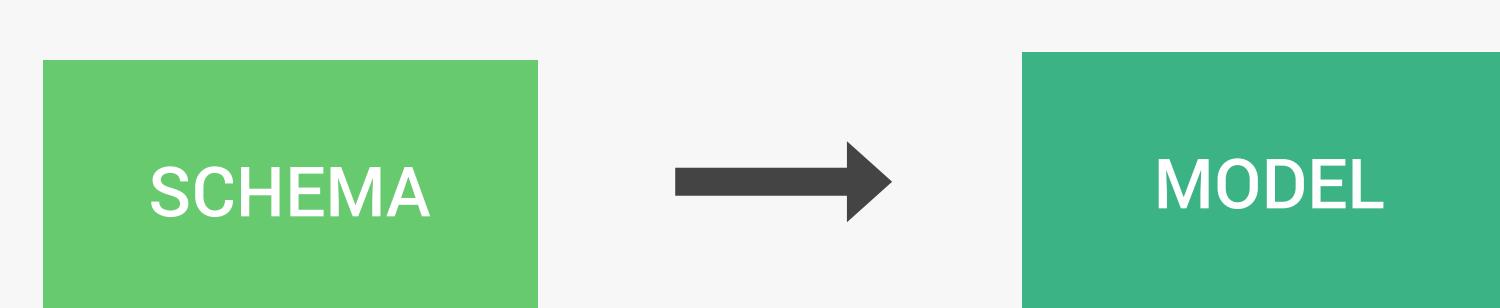
WHAT IS MONGOOSE, AND WHY USE IT?

mongoose

 mongoDB



- 👉 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.





JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

USING MONGODB WITH MONGOOSE

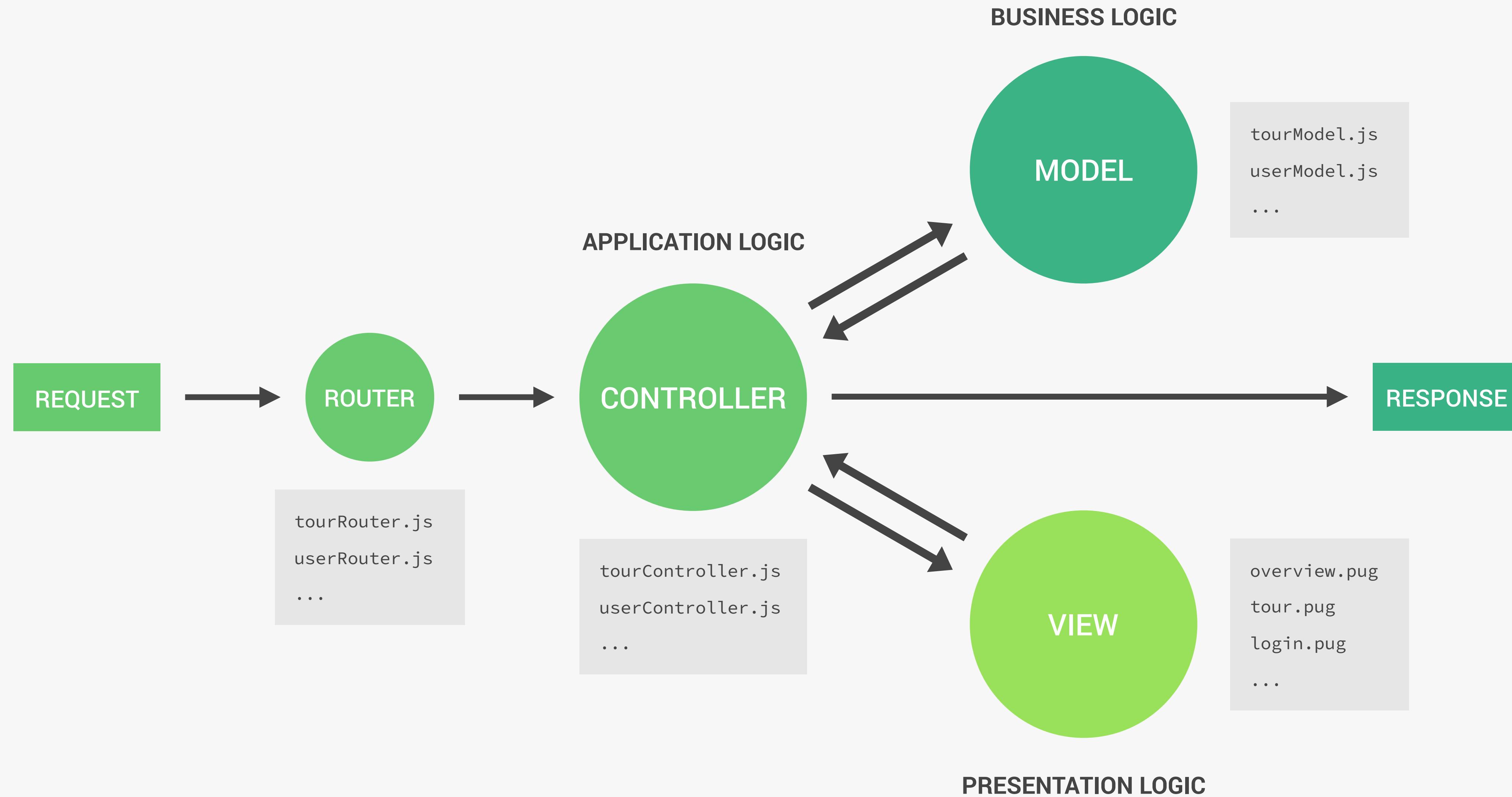
LECTURE

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

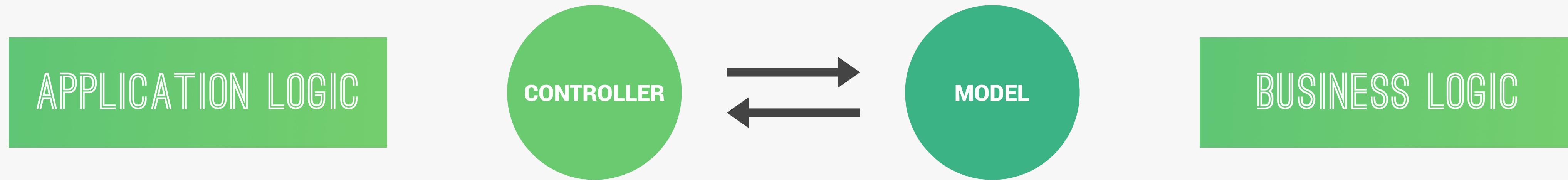


@JONASSCHMEDTMAN

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

SECTION

ERROR HANDLING WITH EXPRESS

LECTURE

AN OVERVIEW OF ERROR HANDLING



@JONASSCHMEDTMAN

ERROR HANDLING IN EXPRESS: AN OVERVIEW

OPERATIONAL ERRORS

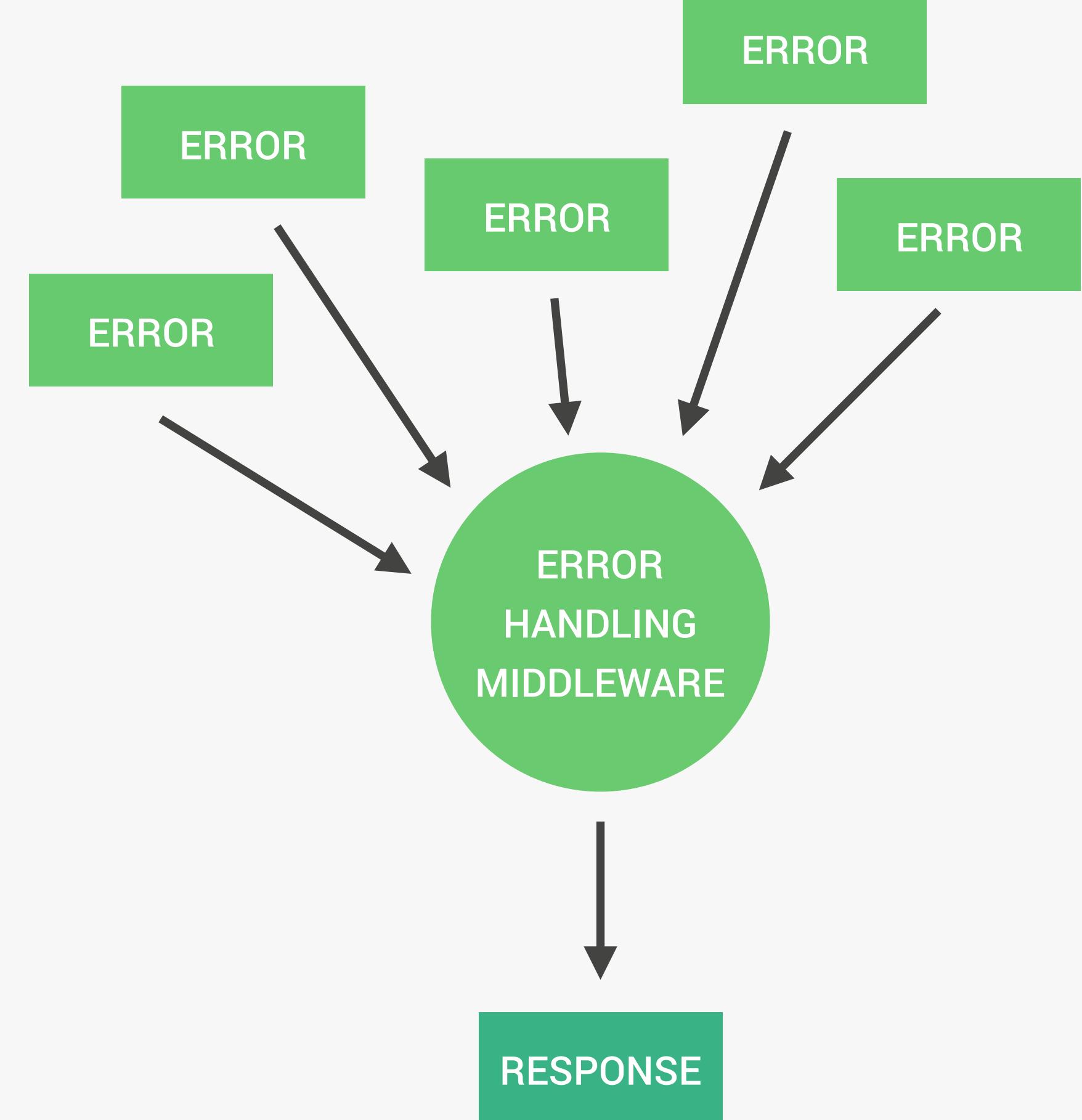
Problems that we can predict will happen at some point, so we just need to handle them in advance.

- 👉 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

SECTION

AUTHENTICATION, AUTHORIZATION AND
SECURITY

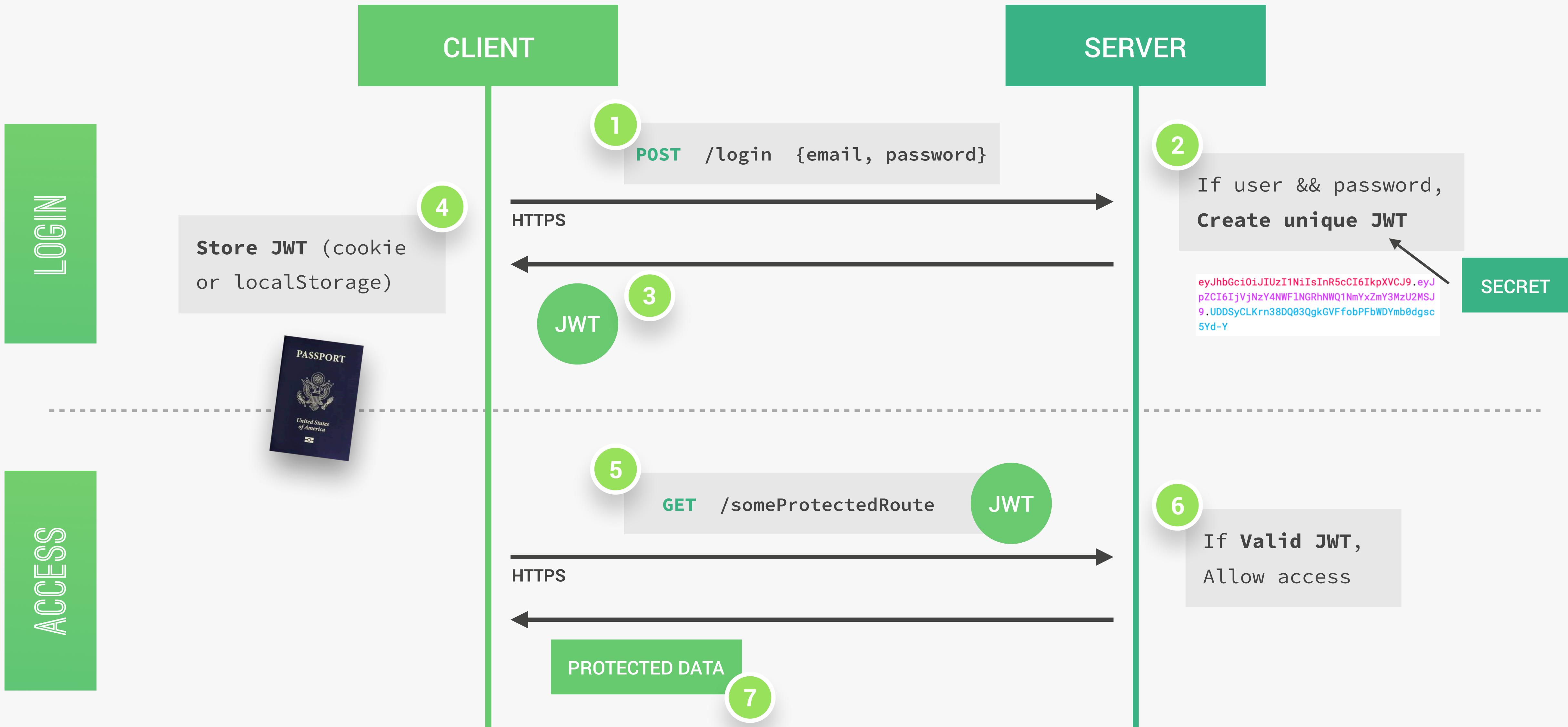
LECTURE

HOW AUTHENTICATION WITH JWT WORKS



@JONASSCHMEDTMAN

HOW JSON WEB TOKEN (JWT) AUTHENTICATION WORKS



WHAT A JWT LOOKS LIKE



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJ  
pZCI6IjVjNzY4NWF1NGRhNWQ1NmYxZmY3MzU2MSJ  
9.UDDSyCLKrn38DQ03QgkGVFfobPFbWDYmb0dgsc  
5Yd-Y
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE

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

PAYOUT: DATA

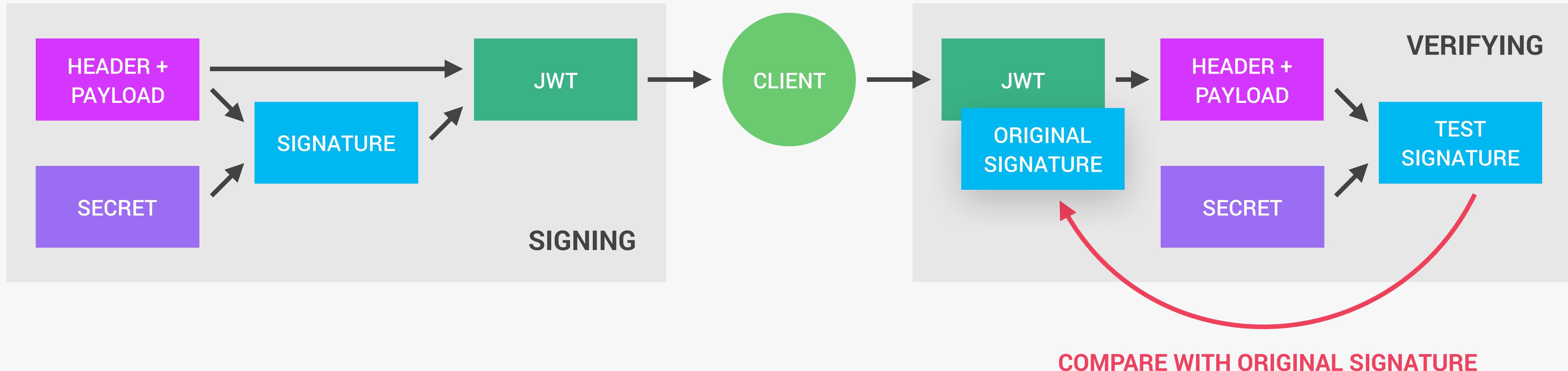
```
{  
  "id": "5c7685ae4da5d56f1ff73561"  
}
```

VERIFY SIGNATURE

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

SECRET

HOW SIGNING AND VERIFYING WORKS



Encoded PASTE A TOKEN HERE

```
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJpZCI6IjVjNzY4NWF1NGRhNWQ1NmYxZmY3MzU2MSJ9.UDDsyCLKn38DQ030gKGVFfobPFbWDYmb0dgsc5Yd-Y
```

Decoded EDIT THE PAYLOAD AND SECRET

HEADER: ALGORITHM & TOKEN TYPE
{ "alg": "HS256", "typ": "JWT" }
PAYOUT: DATA
{ "id": "5c7685ae4da5d56f1ff73561" }
VERIFY SIGNATURE
HMACSHA256(base64UrlEncode(header) + "." + base64UrlEncode(payload), my-very-secret-secret)

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

SECTION

AUTHENTICATION, AUTHORIZATION AND
SECURITY

LECTURE

SECURITY BEST PRACTICES



@JONASSCHMEDTMAN

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

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

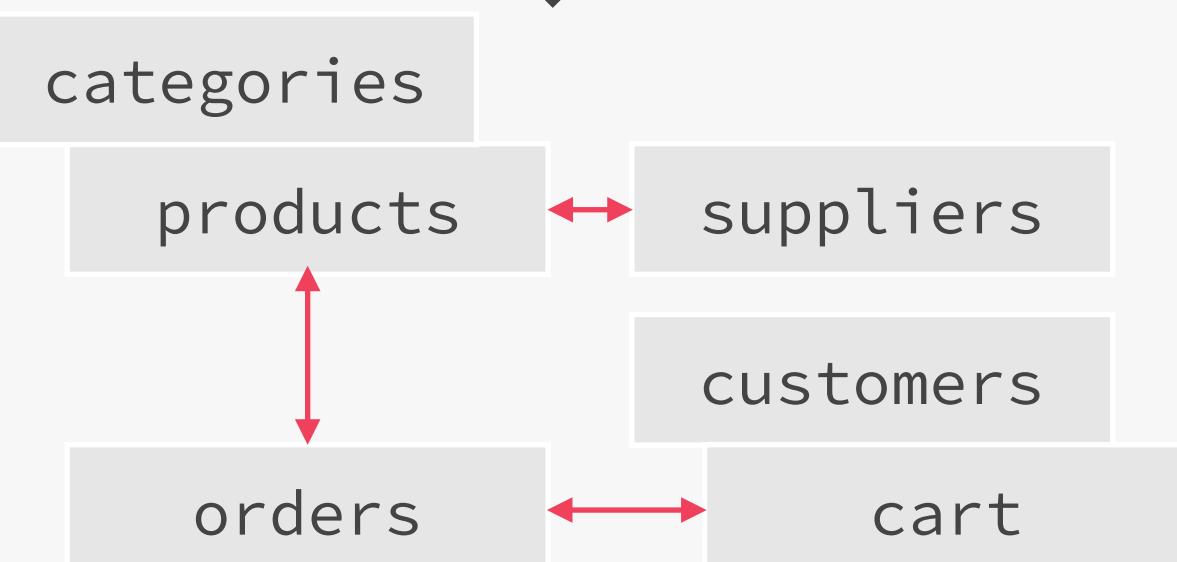
Unstructured data

Structured, logical data model

Example



Online shop



1

Different types of **relationships** between data

2

Referencing/normalization vs. embedding/denormalization

3

Embedding or referencing other documents?

4

Types of referencing

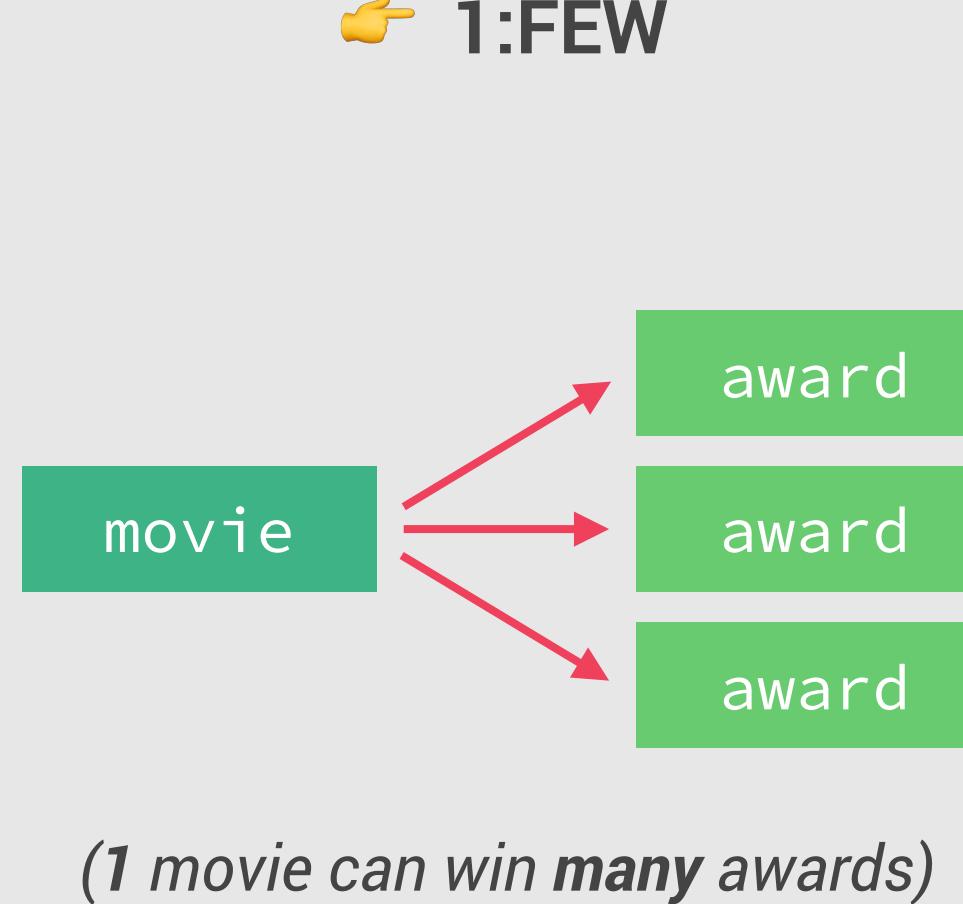
1. TYPES OF RELATIONSHIPS BETWEEN DATA

1:1



(1 movie can only have 1 name)

1:MANY

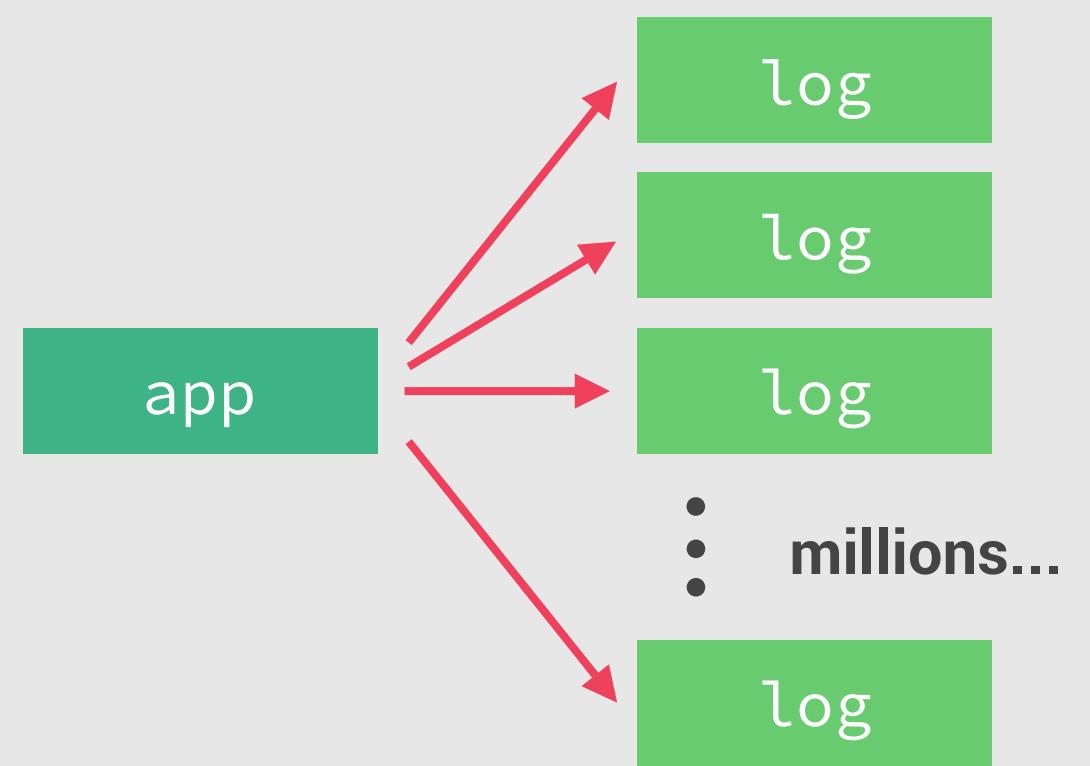
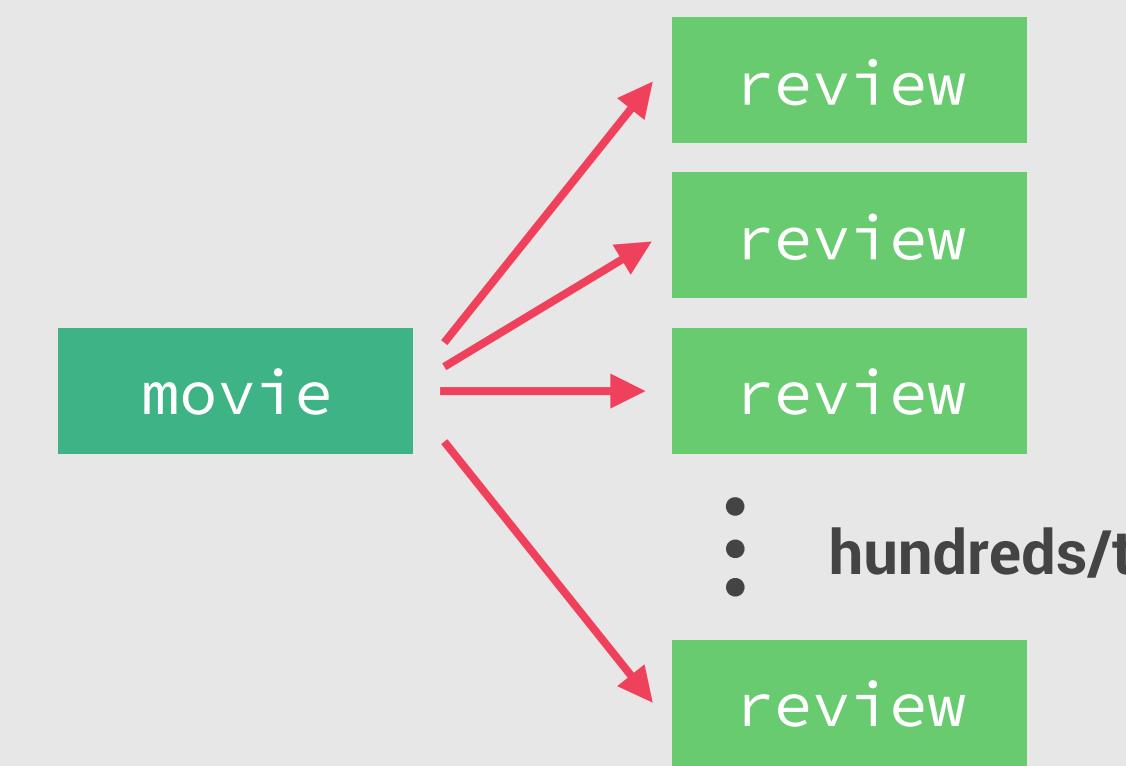


(1 movie can win many awards)

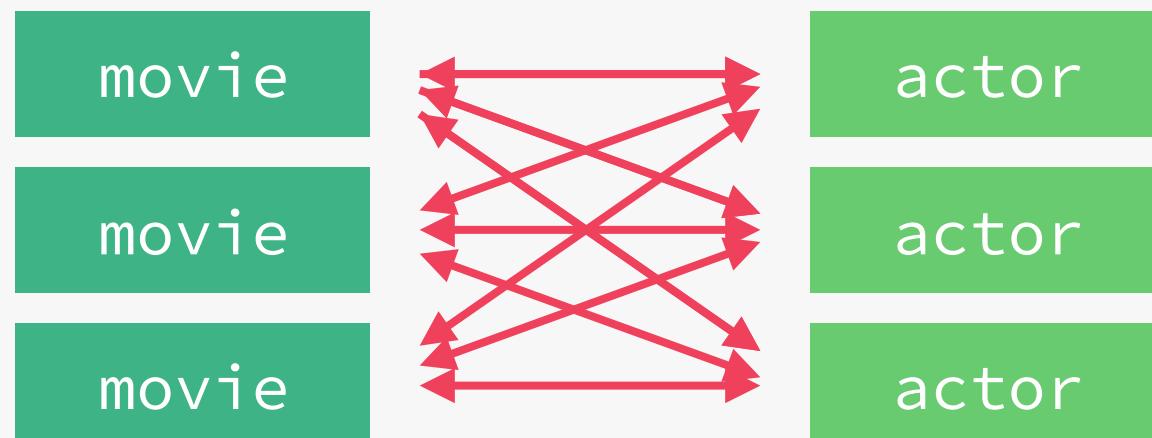
👉 1:FEW

👉 1:MANY

👉 1:TON

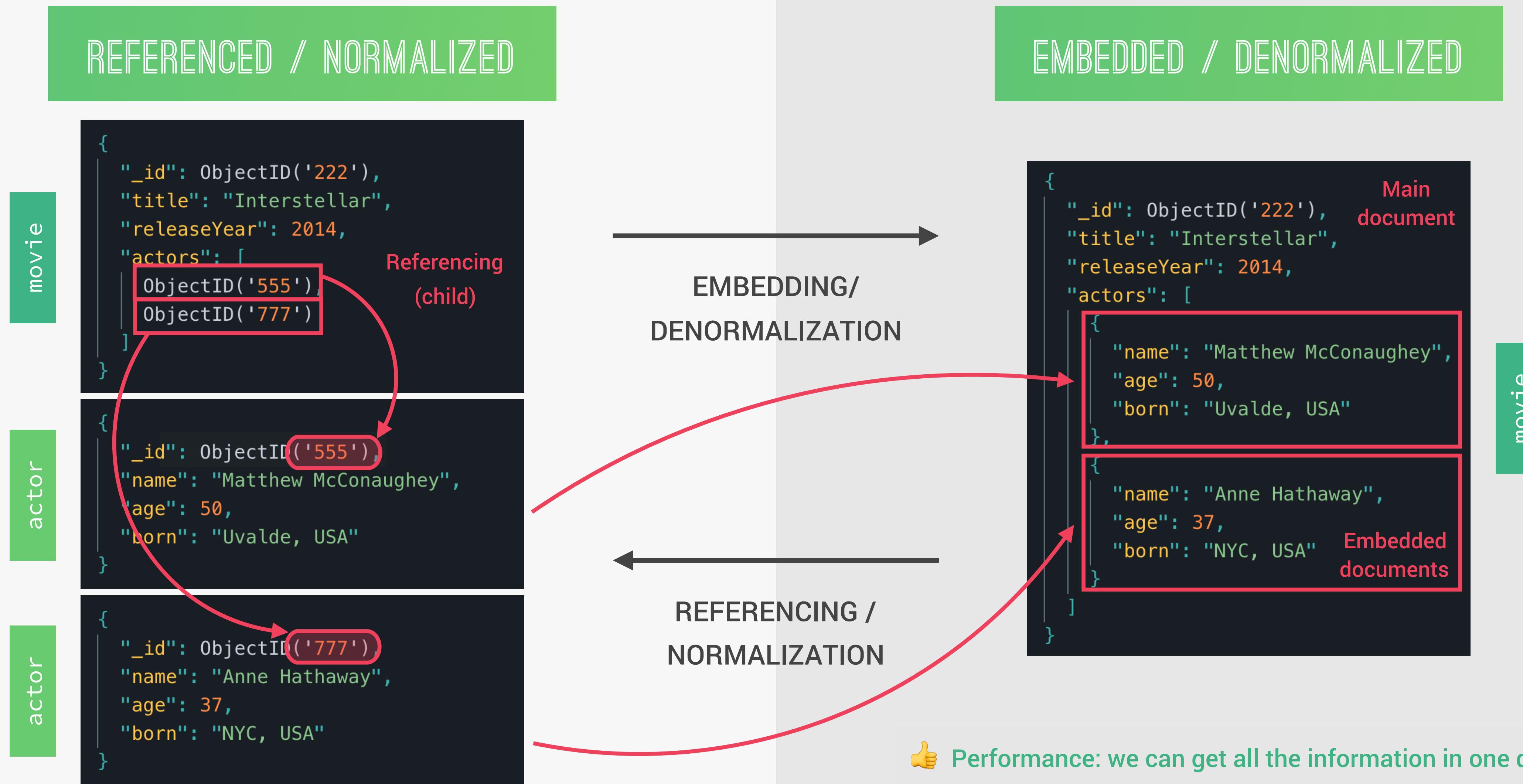


MANY:MANY



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

2. REFERENCING VS. EMBEDDING



👍 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

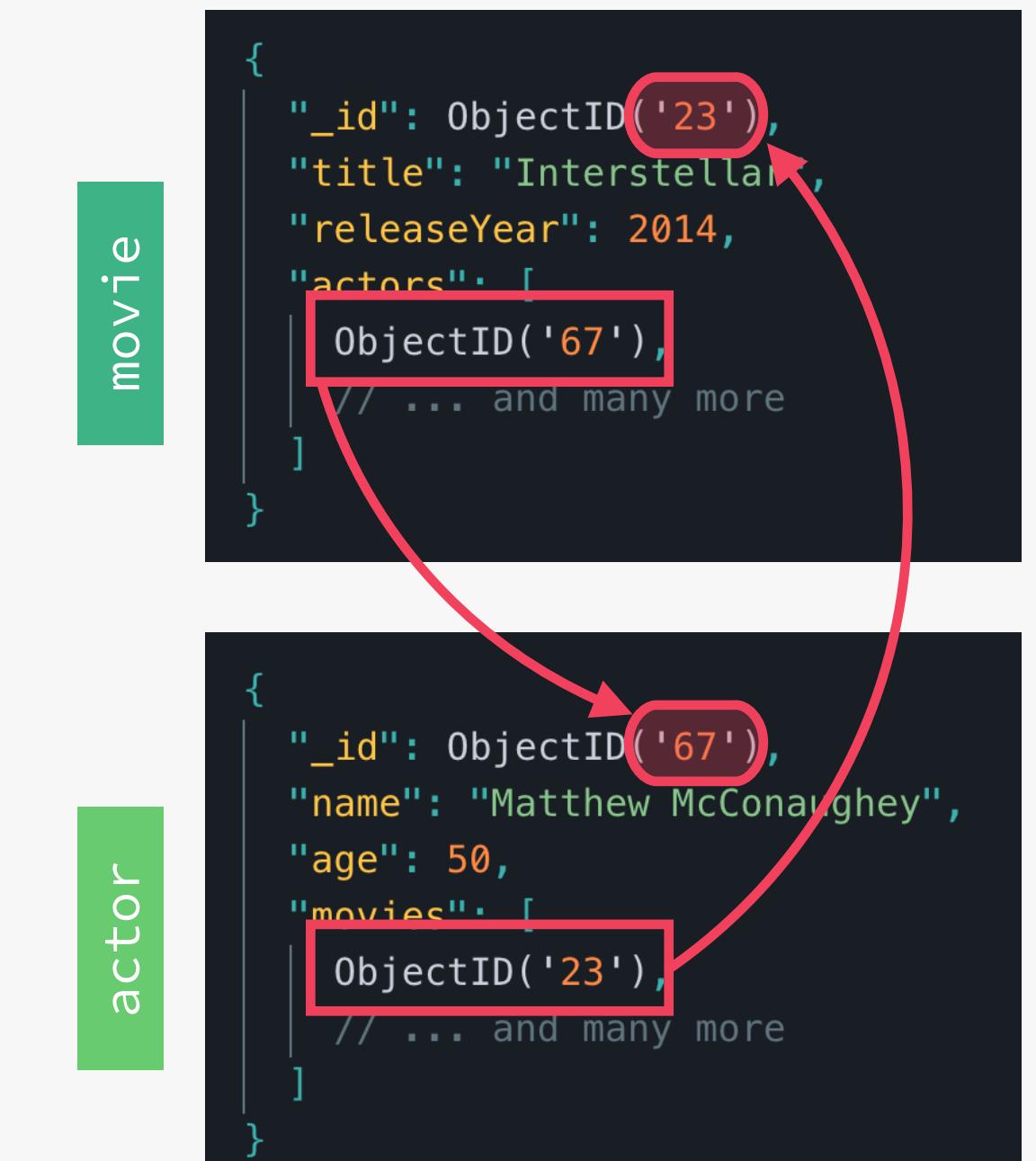
CHILD REFERENCING



PARENT REFERENCING



TWO-WAY REFERENCING

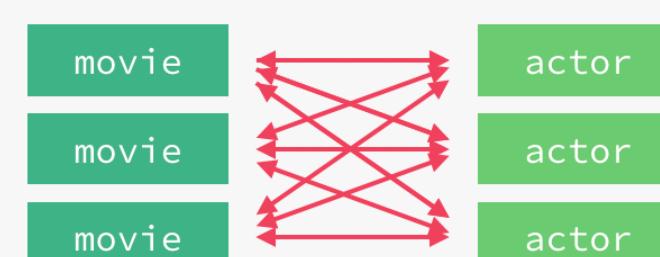


👉 1:FEW

👉 1:TON

👉 1:MANY

👉 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

SECTION

MODELLING DATA AND ADVANCED
MONGOOSE

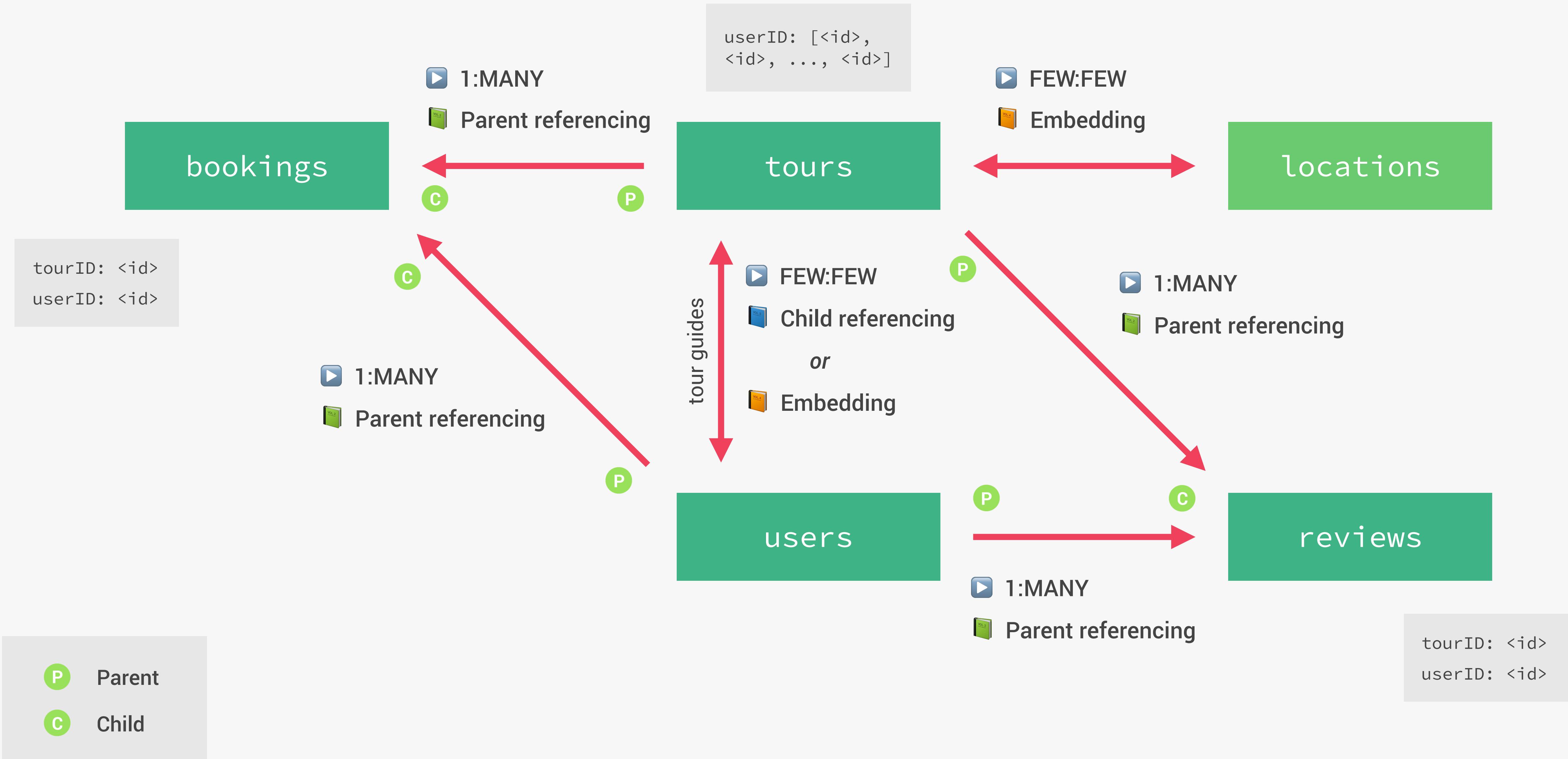
LECTURE

DESIGNING OUR DATA MODEL



@JONASSCHMEDTMAN

THE NATOURS DATA MODEL



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

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP



@JONASSCHMEDTMAN

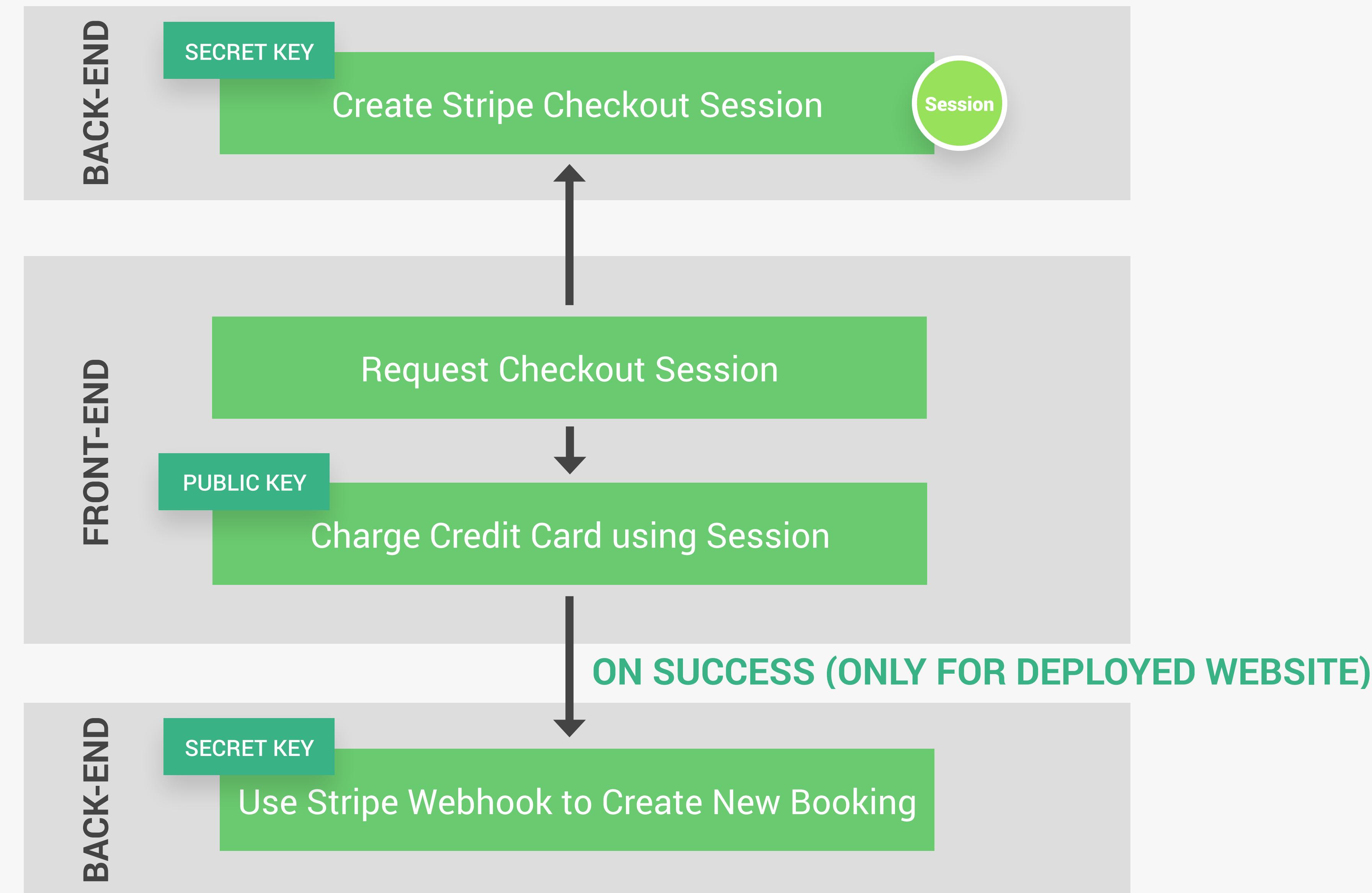
SECTION

ADVANCED FEATURES: PAYMENTS, EMAIL,
FILE UPLOADS

LECTURE

CREDIT CARD PAYMENTS WITH STRIPE

STRIPE WORKFLOW





JONAS.IO
SCHMEDTMANN

NODE.JS, EXPRESS & MONGODB

THE COMPLETE BOOTCAMP

SECTION

ADVANCED FEATURES: PAYMENTS, EMAIL,
FILE UPLOADS

LECTURE

FINAL CONSIDERATIONS



@JONASSCHMEDTMAN

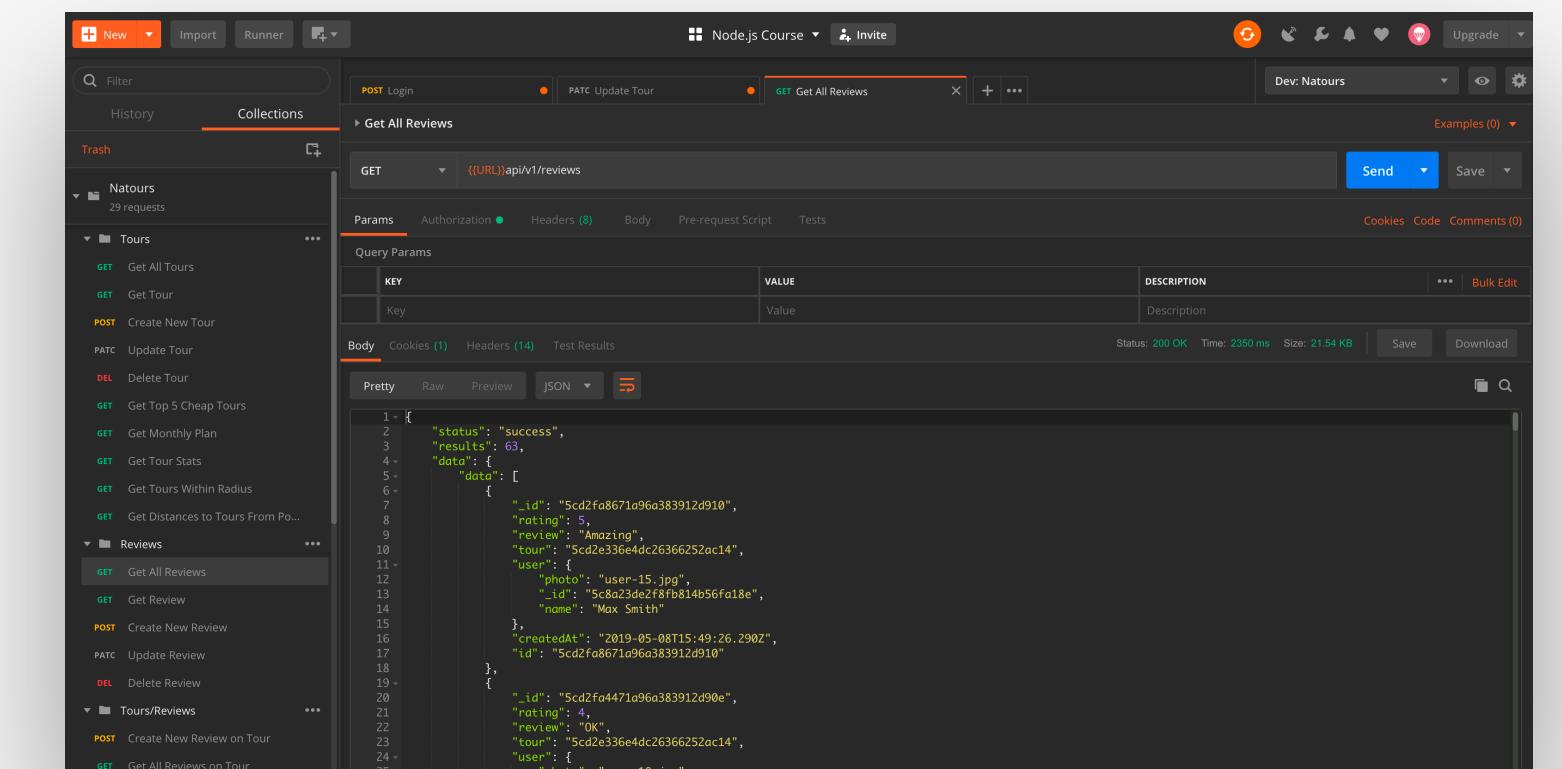
CHALLENGES (API) 😎

👉 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;

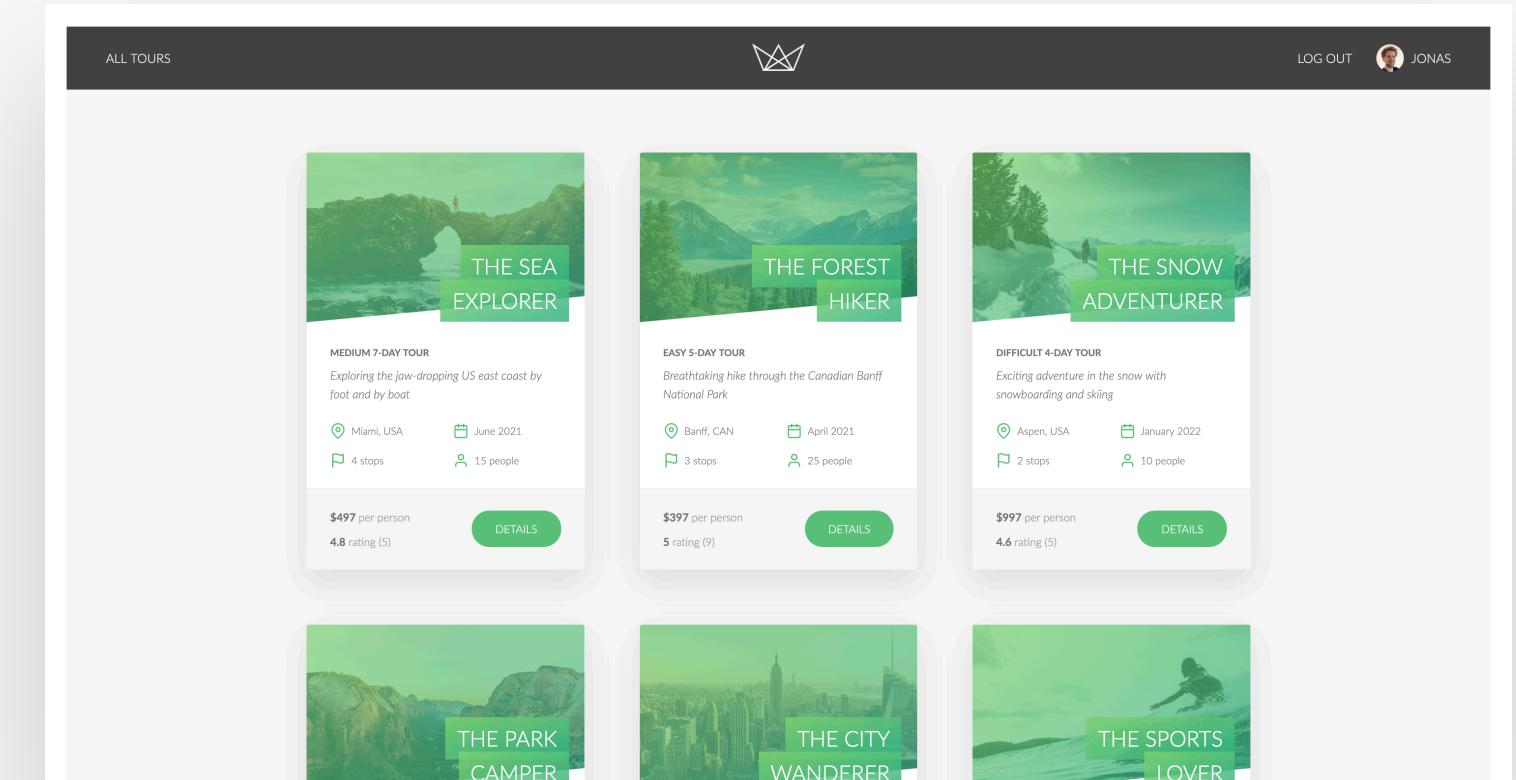
👉 Implement **advanced authentication features**: confirm user email, keep users logged in with refresh tokens, two-factor authentication, etc.



```
1: {
  "status": "success",
  "results": 63,
  "data": [
    {
      "date": [
        {
          "_id": "5cd2fa8671a96a383912d9f0",
          "rating": 5,
          "reviewing": "Anonymous",
          "tour": "5cd2e336e4dc26366252c14",
          "user": "5cd2e336e4dc26366252c14",
          "photo": "user-15.jpg",
          "id": "5cd2e336e4dc26366252c14",
          "name": "Max Smith"
        },
        {
          "createdat": "2019-05-08T15:49:26.290Z",
          "id": "5cd2fa8671a96a383912d9f0"
        }
      ],
      "participants": 15,
      "soldOut": false
    }
  ]
}
```

CHALLENGES (WEBSITE) 😎

- 👉 Implement a **sign up** form, 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