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

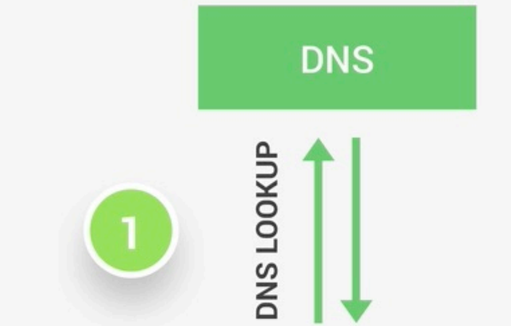
- API with database behind it (preferably NoSQL);
- Data streaming (think YouTube);
- Real-time chat application;
- Server-side web application.

DON'T USE

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

RAILS php

WHAT HAPPENS WHEN WE ACCESS A WEBPAGE




```
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)

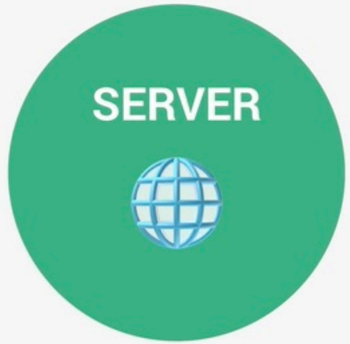
```
<BODY>
```

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

HTTP REQUEST 3

2 TCP/IP socket connection

HTTP RESPONSE 4



5

`index.html` is the first to be loaded

👇

Scanned for assets: JS, CSS, images

👇

Process is repeated for each file

```
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)

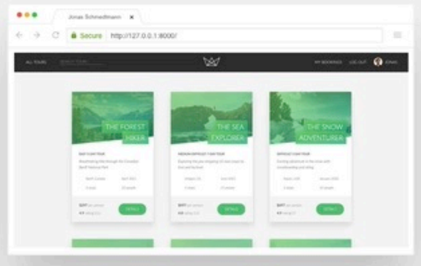
```
<BODY>
```

Response body (most responses)

FRONT-END AND BACK-END

FRONT-END

BACK-END



BROWSER

WEB SERVER

HTTP Server

App

DATABASE

Files

PostgreSQL

MySQL

php

FRONT-END STACK

HTML

CSS

JS

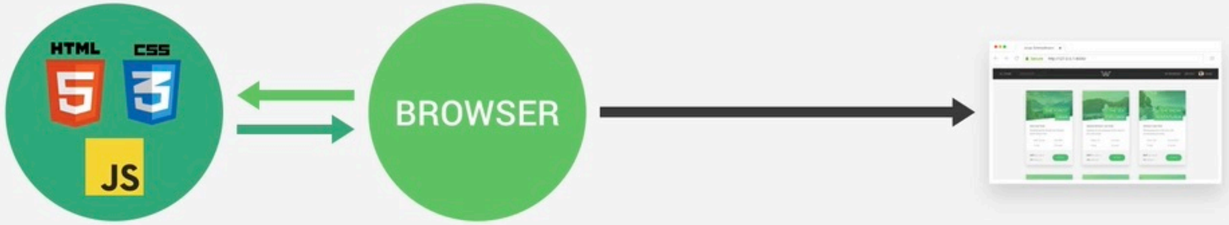
BACK-END STACK

node

mongoDB

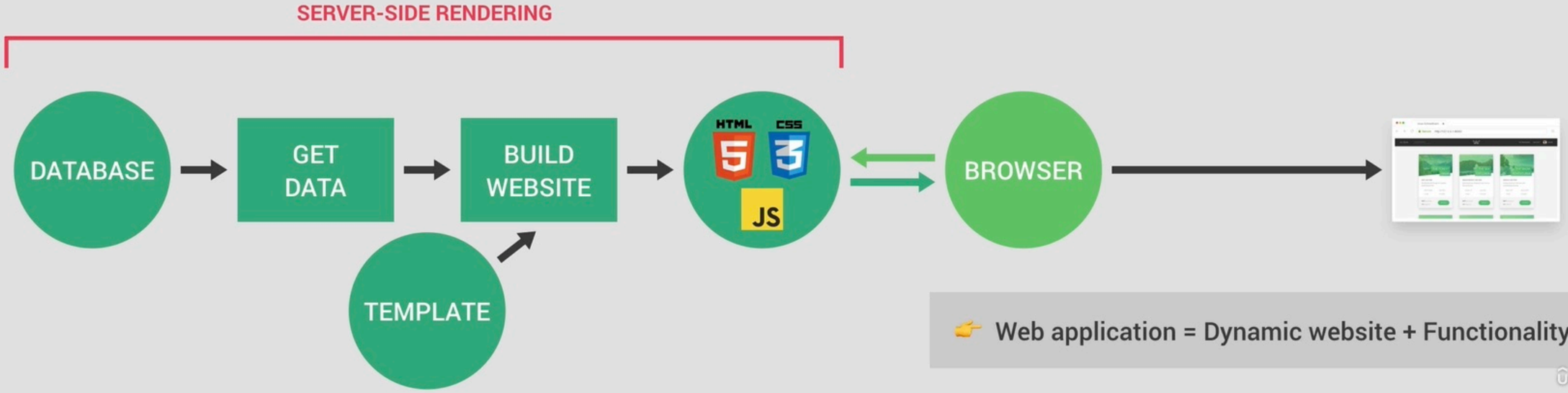
STATIC WEBSITES VS DYNAMIC WEBSITES

STATIC



👉 JavaScript ≠ Dynamic

DYNAMIC



👉 Web application = Dynamic website + Functionality

DYNAMIC WEBSITES VS API-POWERED WEBSITES

DYNAMIC

API

THIS COURSE 🚀😍

SERVER-SIDE RENDERED

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

TEMPLATE

CLIENT-SIDE RENDERED

BUILDING API

CONSUMING API

DATABASE → GET DATA → JSON

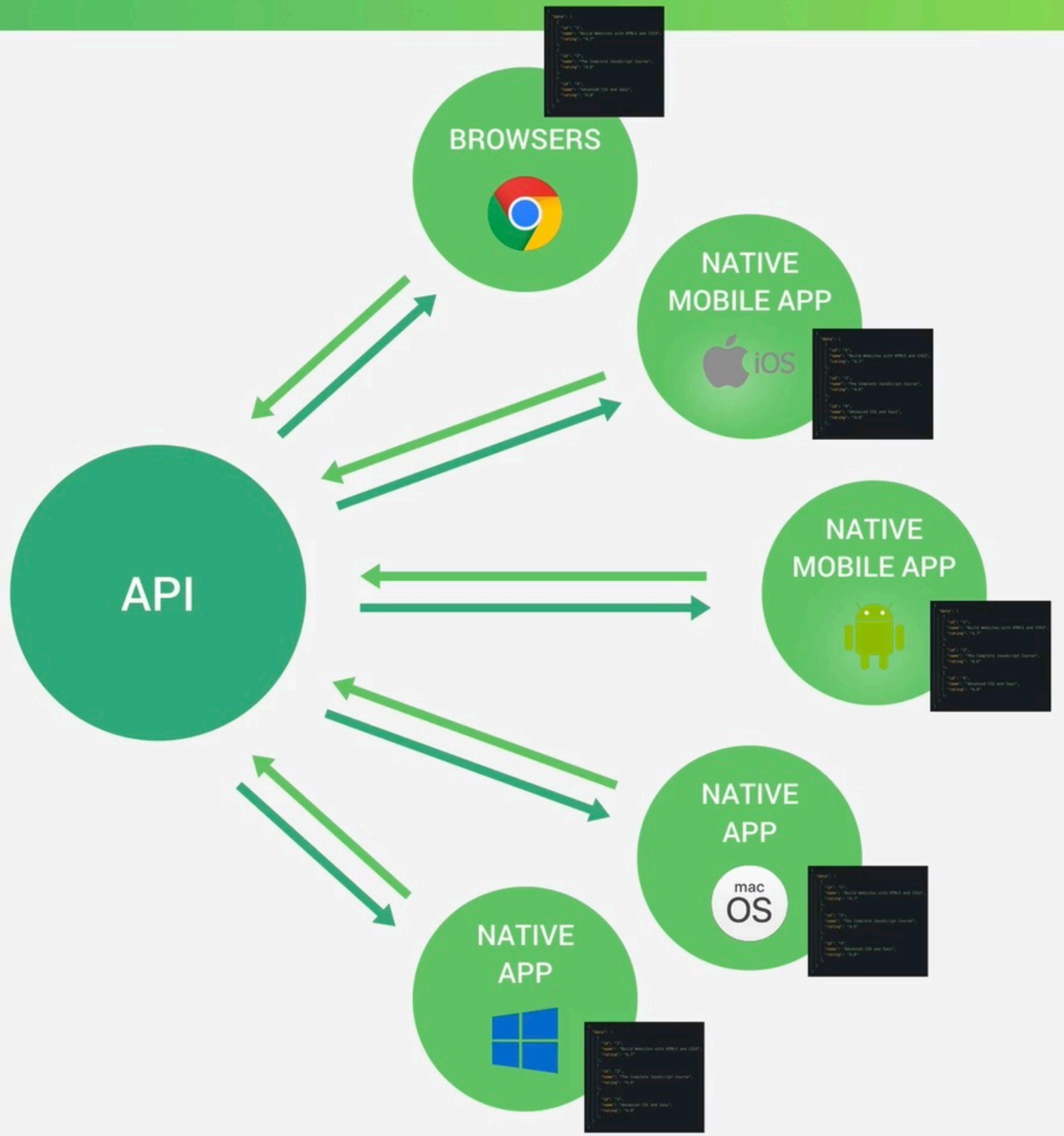
BROWSER → BUILD WEBSITE

TEMPLATE

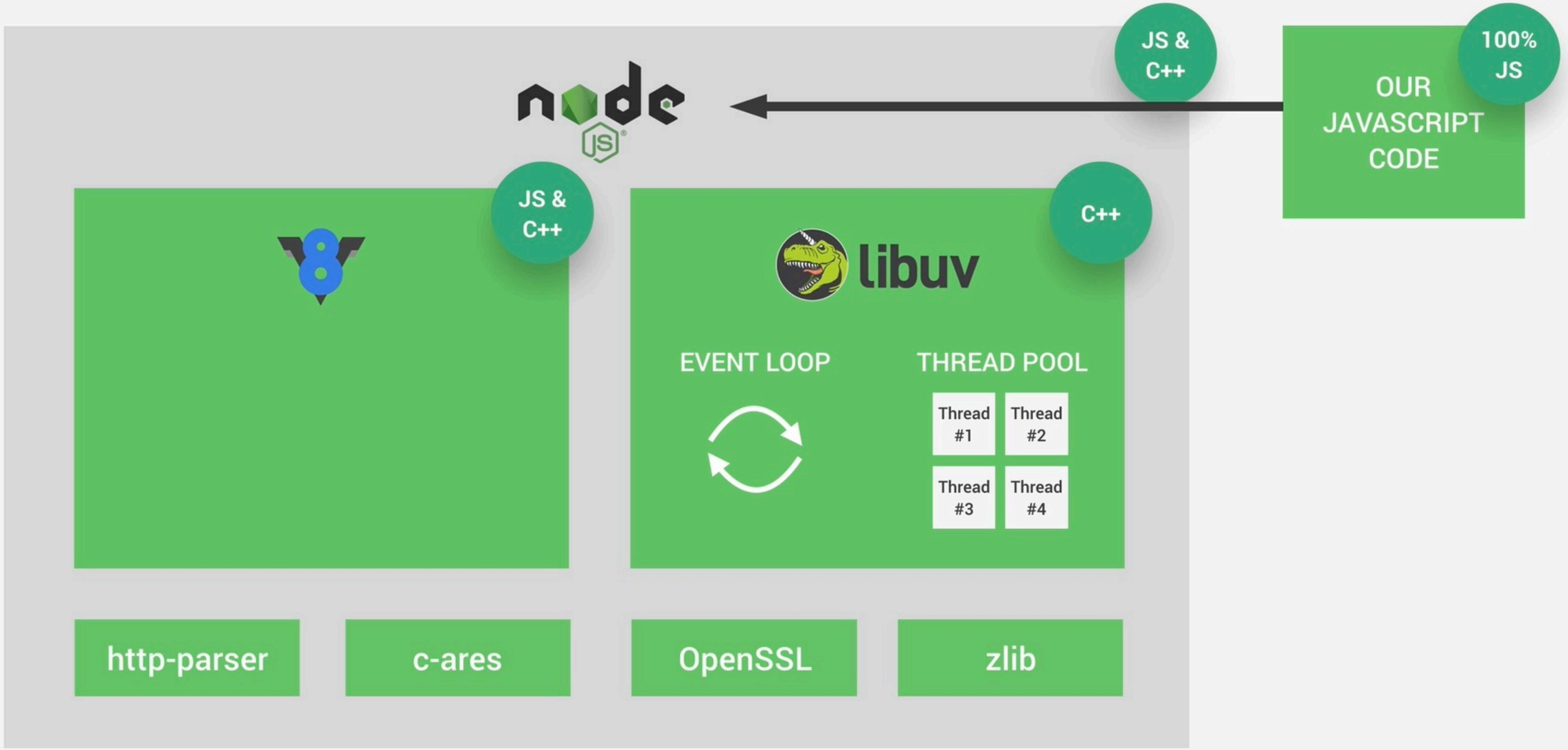
ONE API, MANY CONSUMERS

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

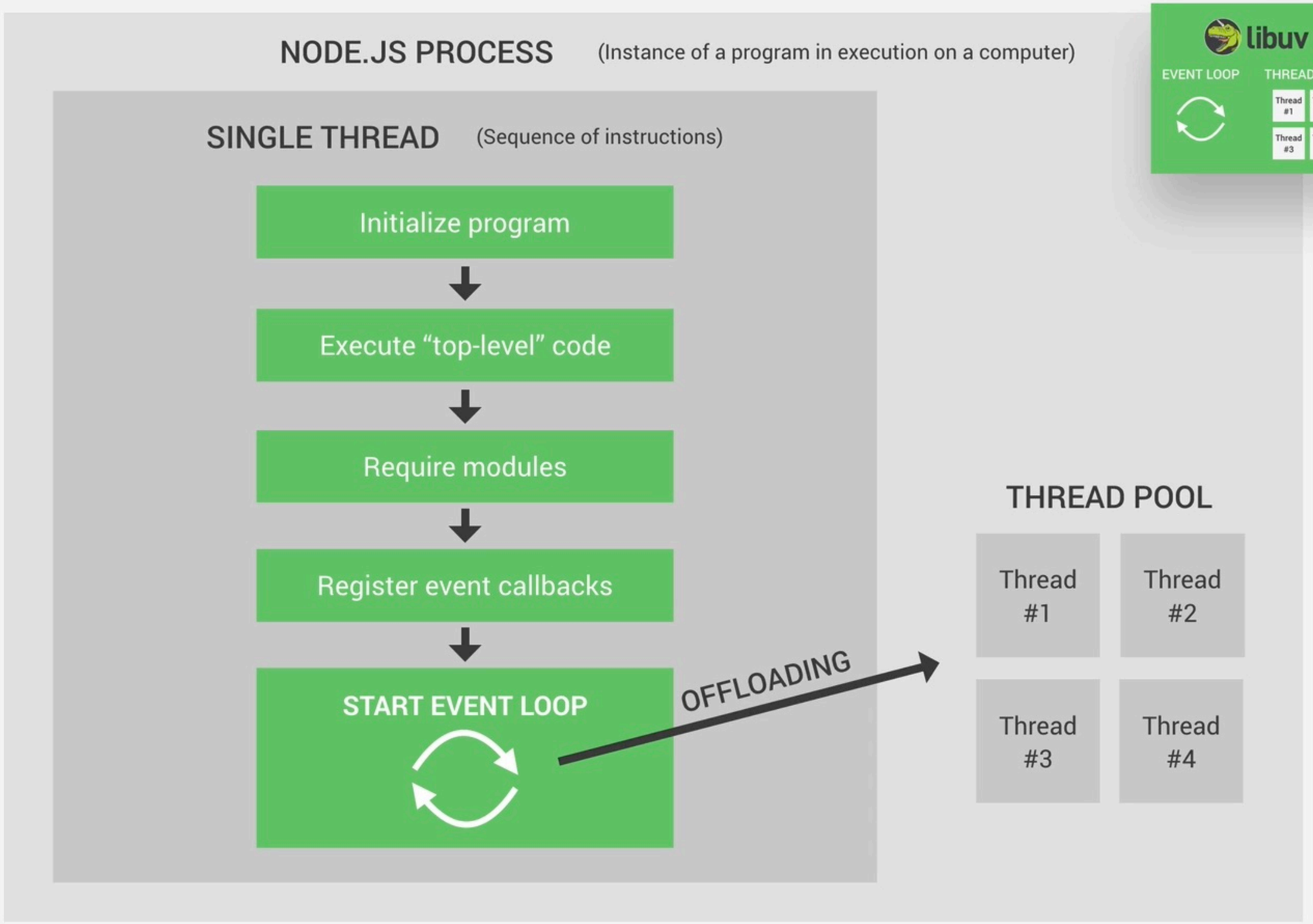

```
{
  "data": [
    {
      "id": "1",
      "name": "Build Websites with HTML5 and CSS3",
      "rating": "4.7"
    },
    {
      "id": "3",
      "name": "The Complete JavaScript Course",
      "rating": "4.6"
    },
    {
      "id": "4",
      "name": "Advanced CSS and Sass",
      "rating": "4.8"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

THE NODE.JS ARCHITECTURE BEHIND THE SCENES



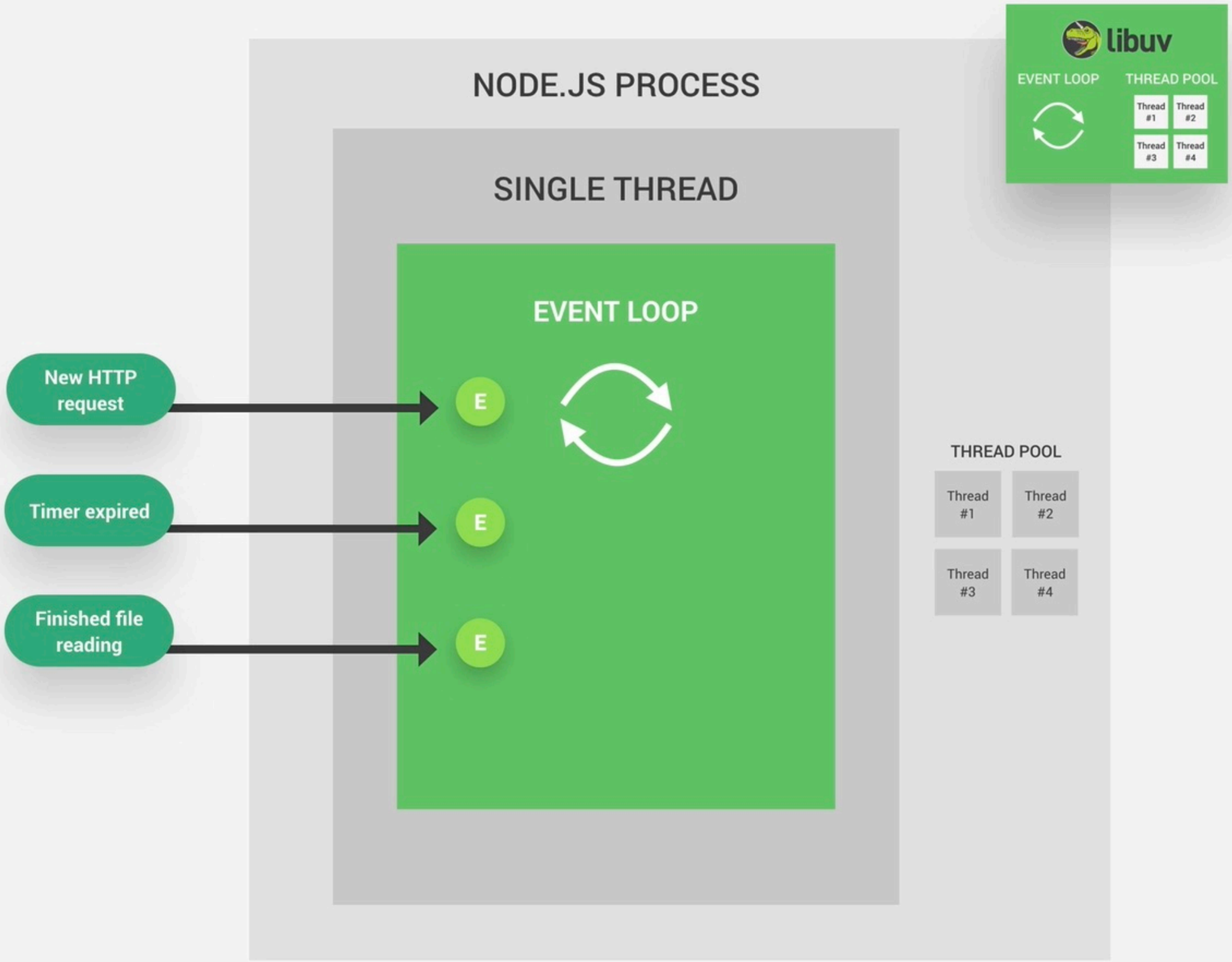
NODE PROCESS AND THREADS



THREAD POOL:

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

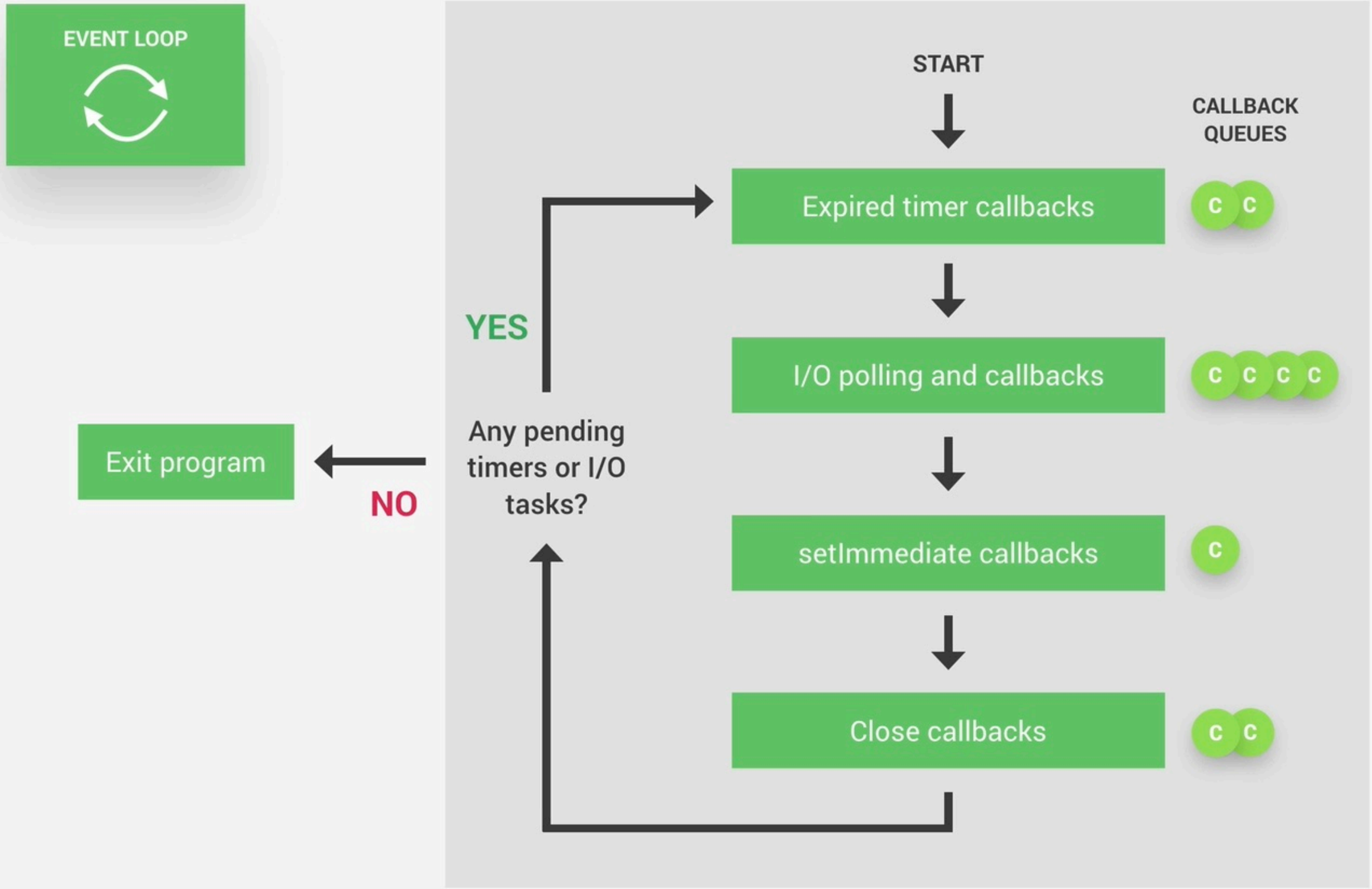
THE HEART OF NODE.JS: THE EVENT LOOP



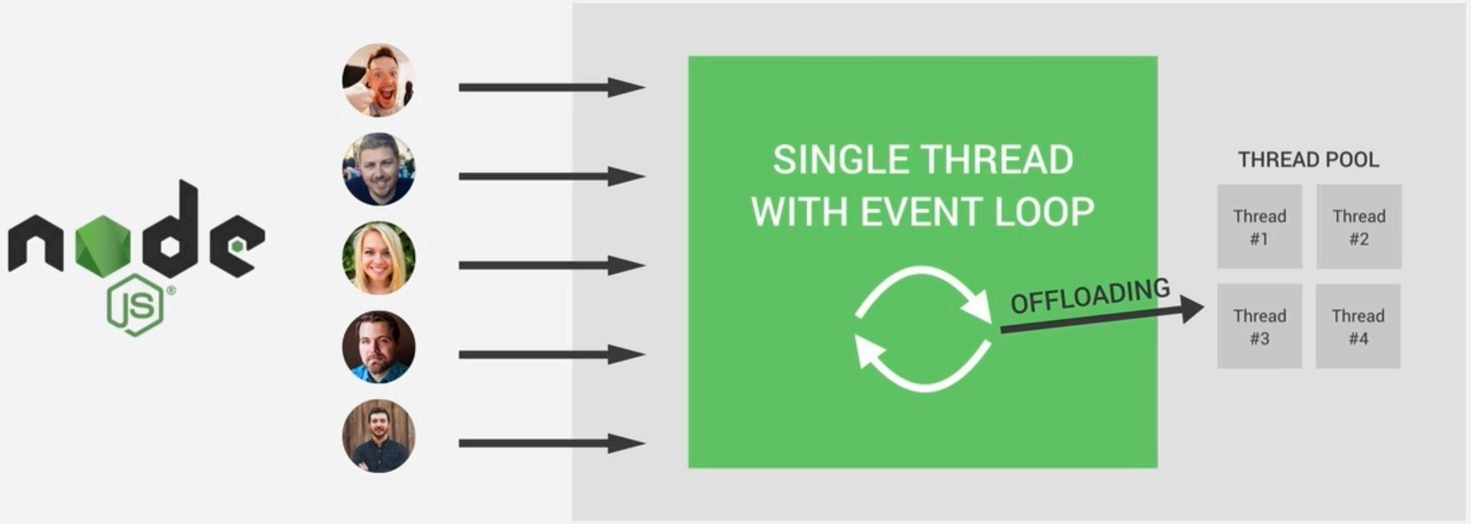
EVENT LOOP:

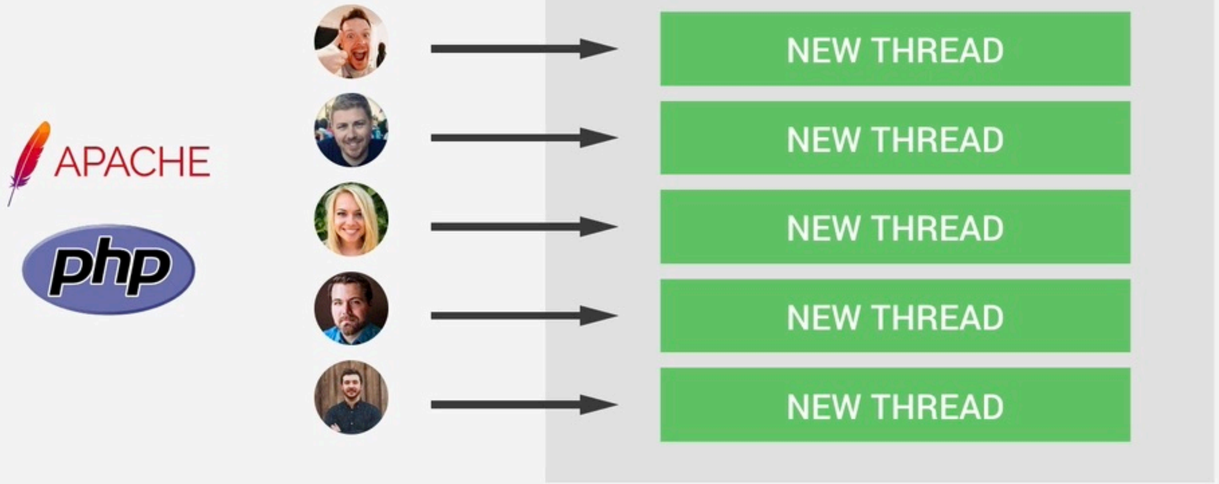
- All the application code that is inside **callback functions** (non-top-top-level code)
- Node.js is build around callback functions
- Event-driven architecture:
  - Events are emitted
  - Event loops picks them up
  - Callbacks are called
- Event loop does **orchestration**

THE EVENT LOOP IN DETAIL



SUMMARY OF THE EVENT LOOP: NODE VS. OTHERS

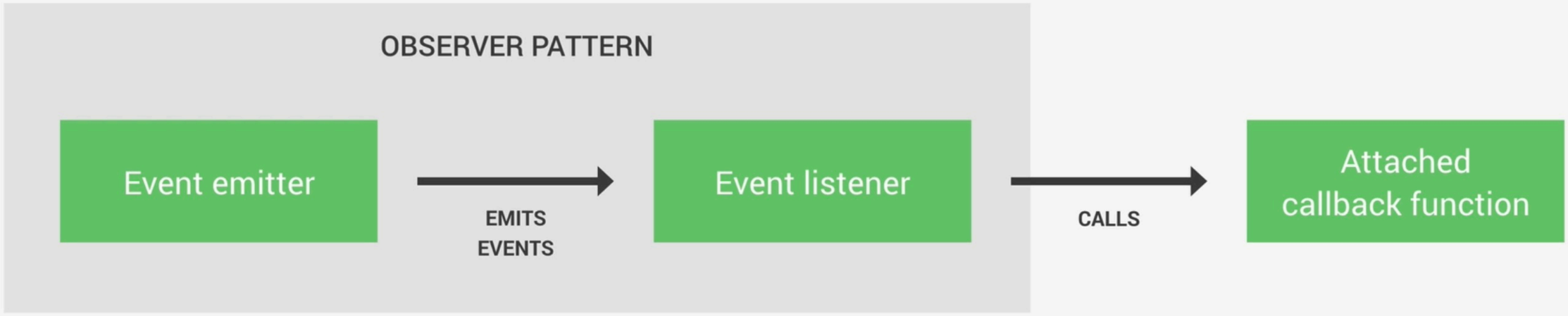




DON'T BLOCK!

- 👉 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



EMITTER

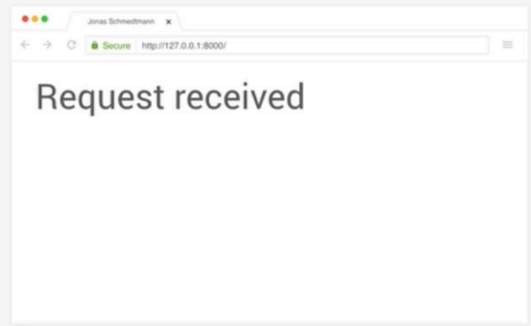
NEW REQUEST ON SERVER

127.0.0.1:8000

LISTENER

'request' event

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



👉 Instance of `EventEmitter` class

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		

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');
```

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

👉 **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...
});
```

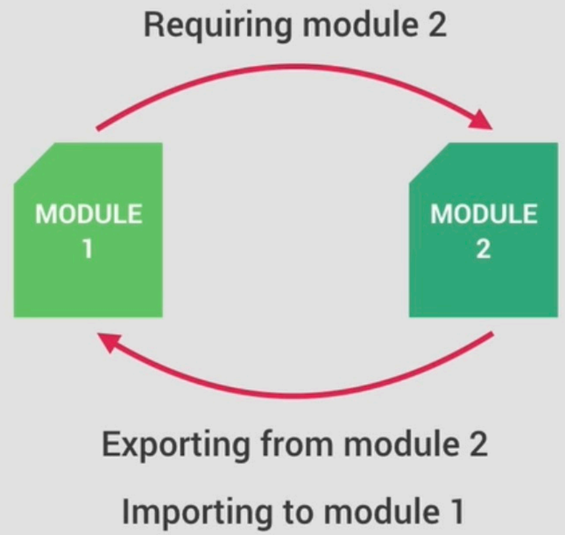
- 👉 **`require`**: function to require modules;
- 👉 **`module`:** reference to the current module;
- 👉 **`exports`**: a reference to `module.exports`, used to export object from a module;
- 👉 **`__filename`**: absolute path of the current module's file;
- 👉 **`__dirname`**: directory name of the current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

WHAT HAPPENS WHEN WE REQUIRE() A MODULE

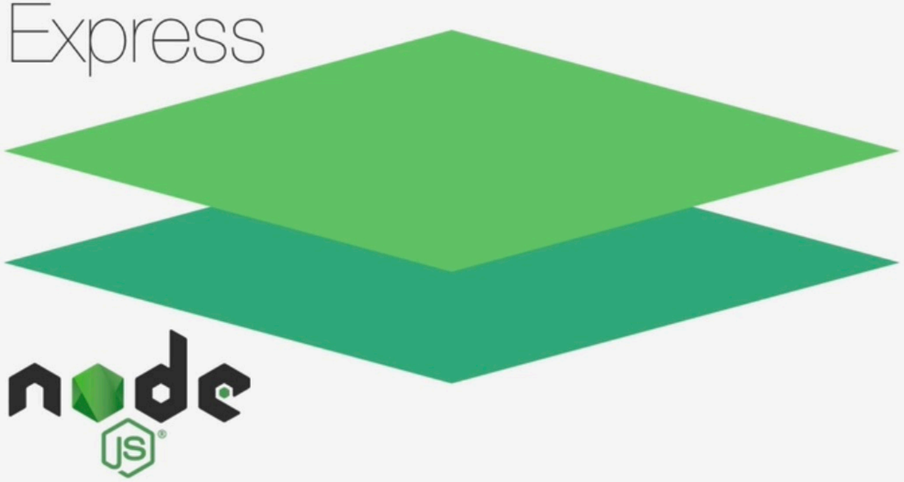
```
require('test-module');
```

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;



WHAT IS EXPRESS, AND WHY USE IT?



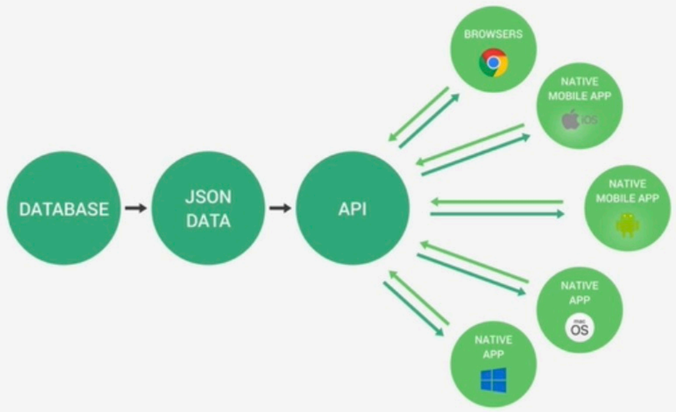
- 👉 Express is a minimal node.js framework, a higher level of abstraction;
- 👉 Express contains a very robust set of features: **complex routing, easier handling of requests and responses, middleware, server-side rendering**, etc.;
- 👉 Express allows for rapid development of node.js applications: *we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

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:

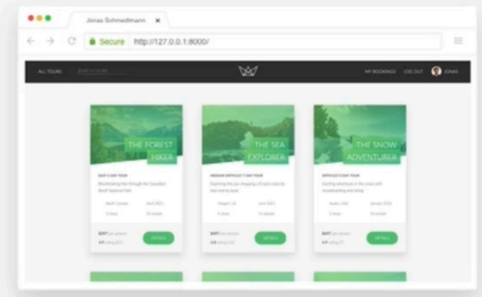
- 👉 Node.js’ `fs` or `http` APIs (“node APIs”);
- 👉 Browser’s DOM JavaScript API;
- 👉 With object-oriented programming, when exposing methods to the public, we’re creating an API;
- 👉 ...

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**

👉 **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`

`/updateTour`

BAD 👎

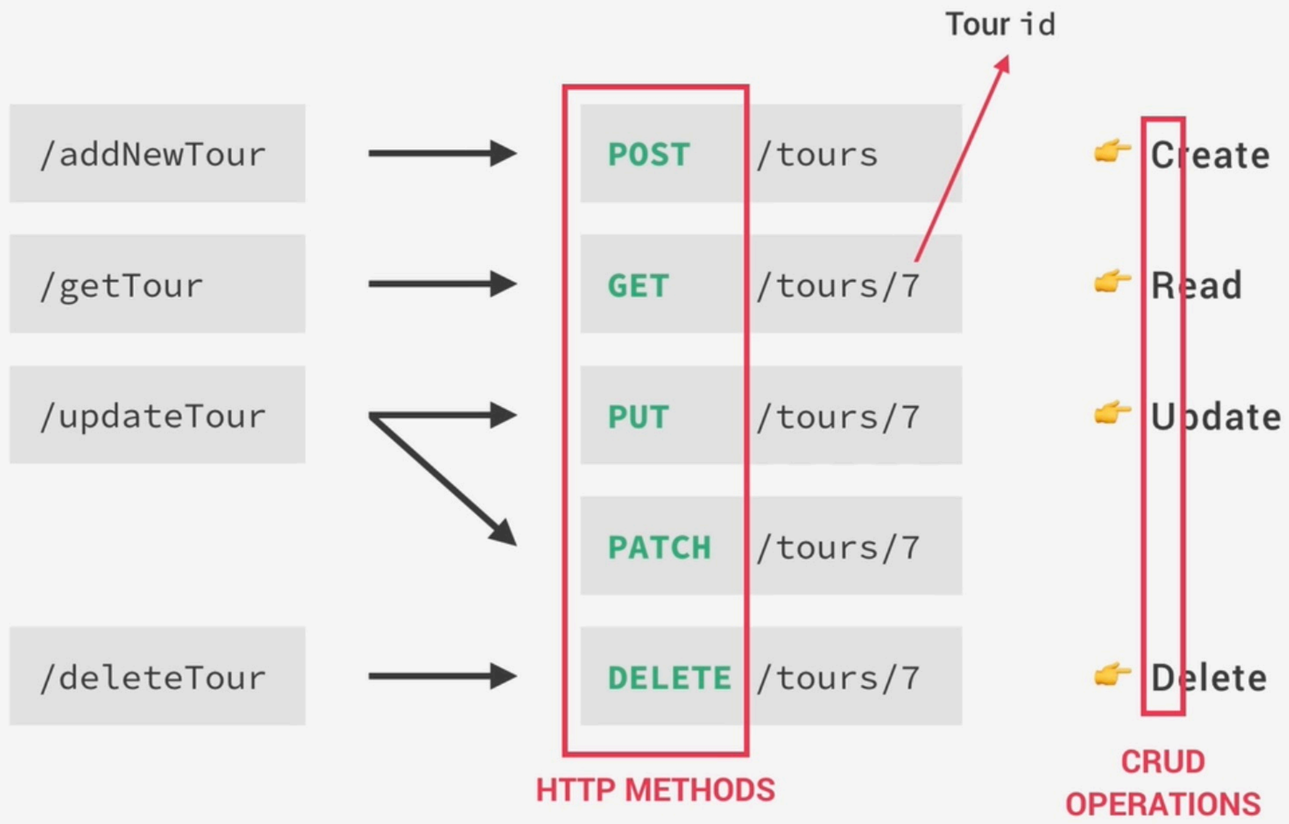
`/getToursByUser`

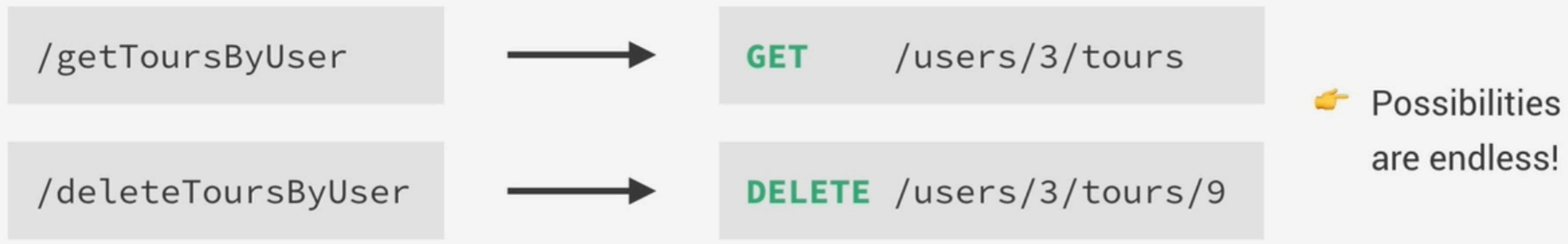
`/deleteToursByUser`

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

THE REST ARCHITECTURE

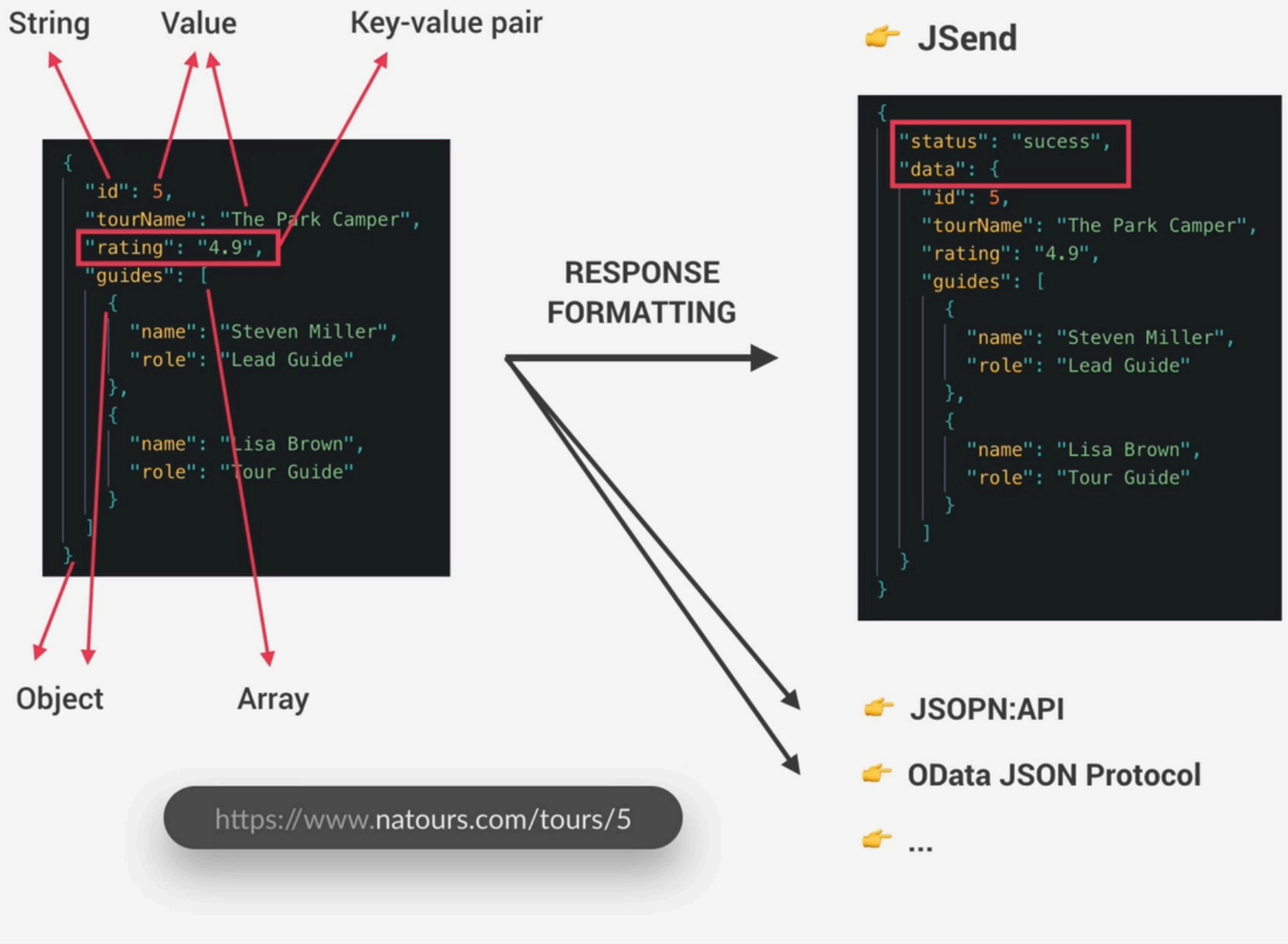
1. Separate API into logical **resources**
2. Expose structured, **resource-based URLs**
3. Use **HTTP methods** (verbs)
4. Send data as **JSON** (usually)
5. Be **stateless**





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**

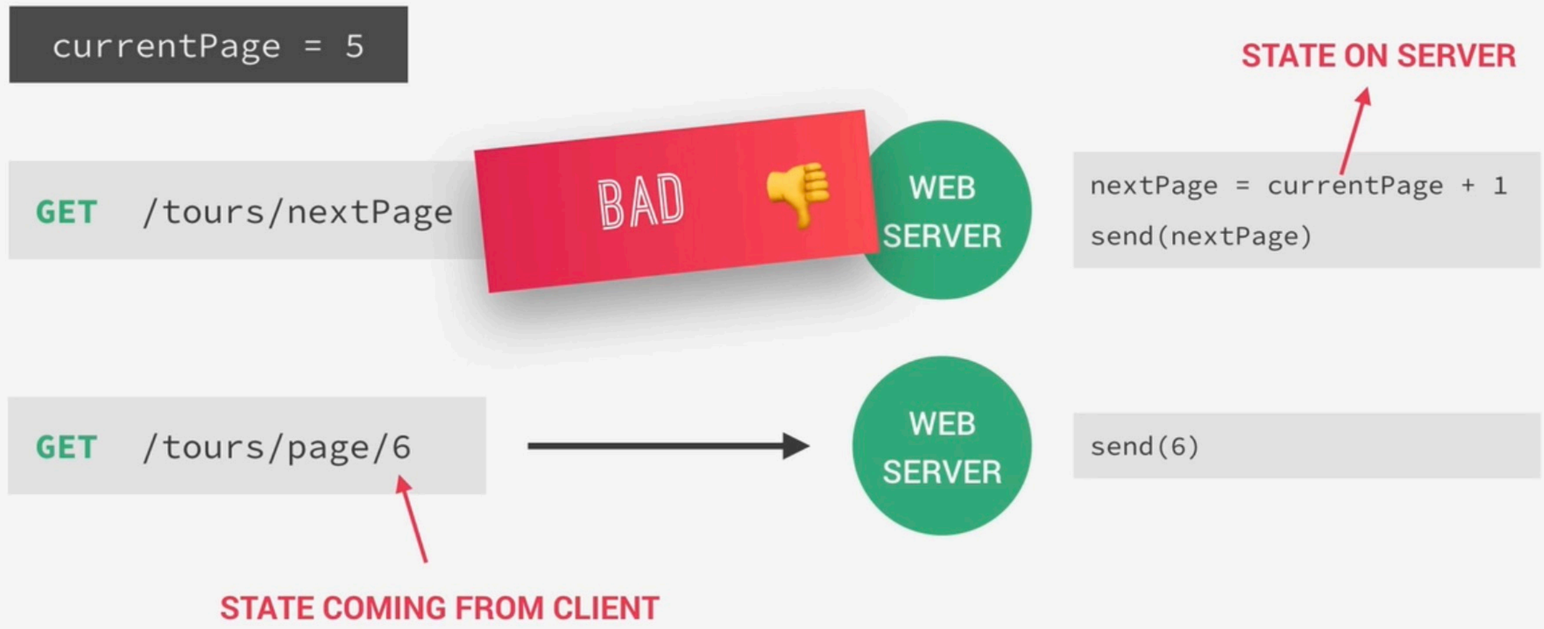


THE REST ARCHITECTURE

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.

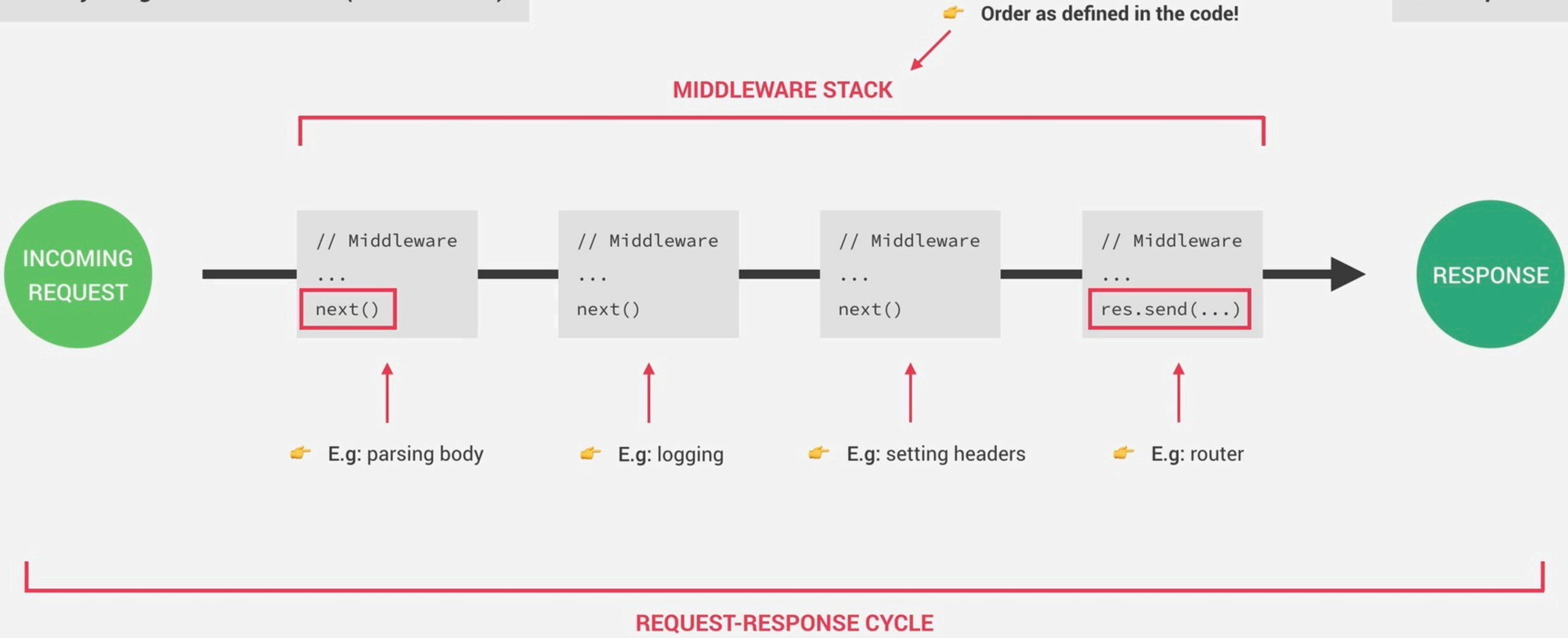
- **Examples of state:** `loggedIn` `currentPage`



THE ESSENCE OF EXPRESS DEVELOPMENT: THE REQUEST-RESPONSE CYCLE

👉 *"Everything is middleware" (even routers)*

👉 *"Pipeline"*



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"

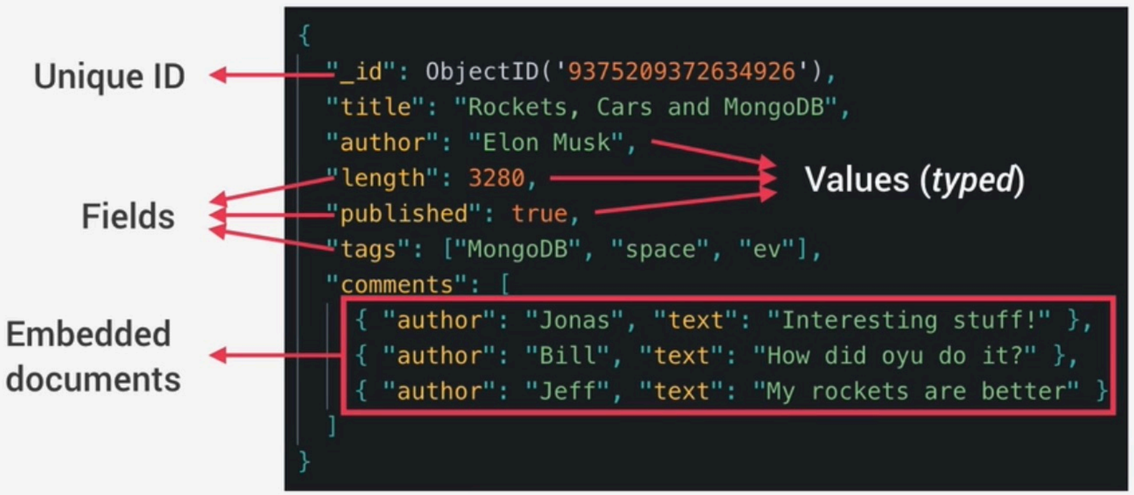
KEY MONGODB FEATURES:

- 👉 **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.



👉 **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

Column

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

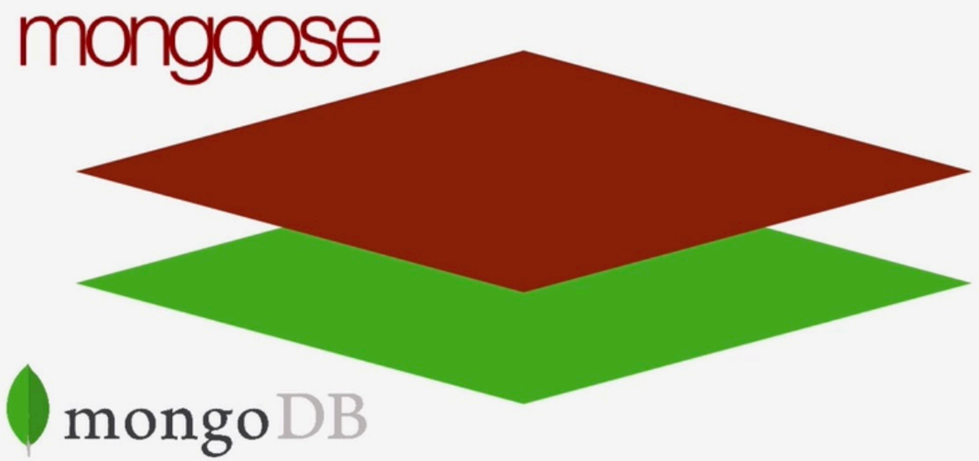
"JOIN tables"

Reference by `comments_id`

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

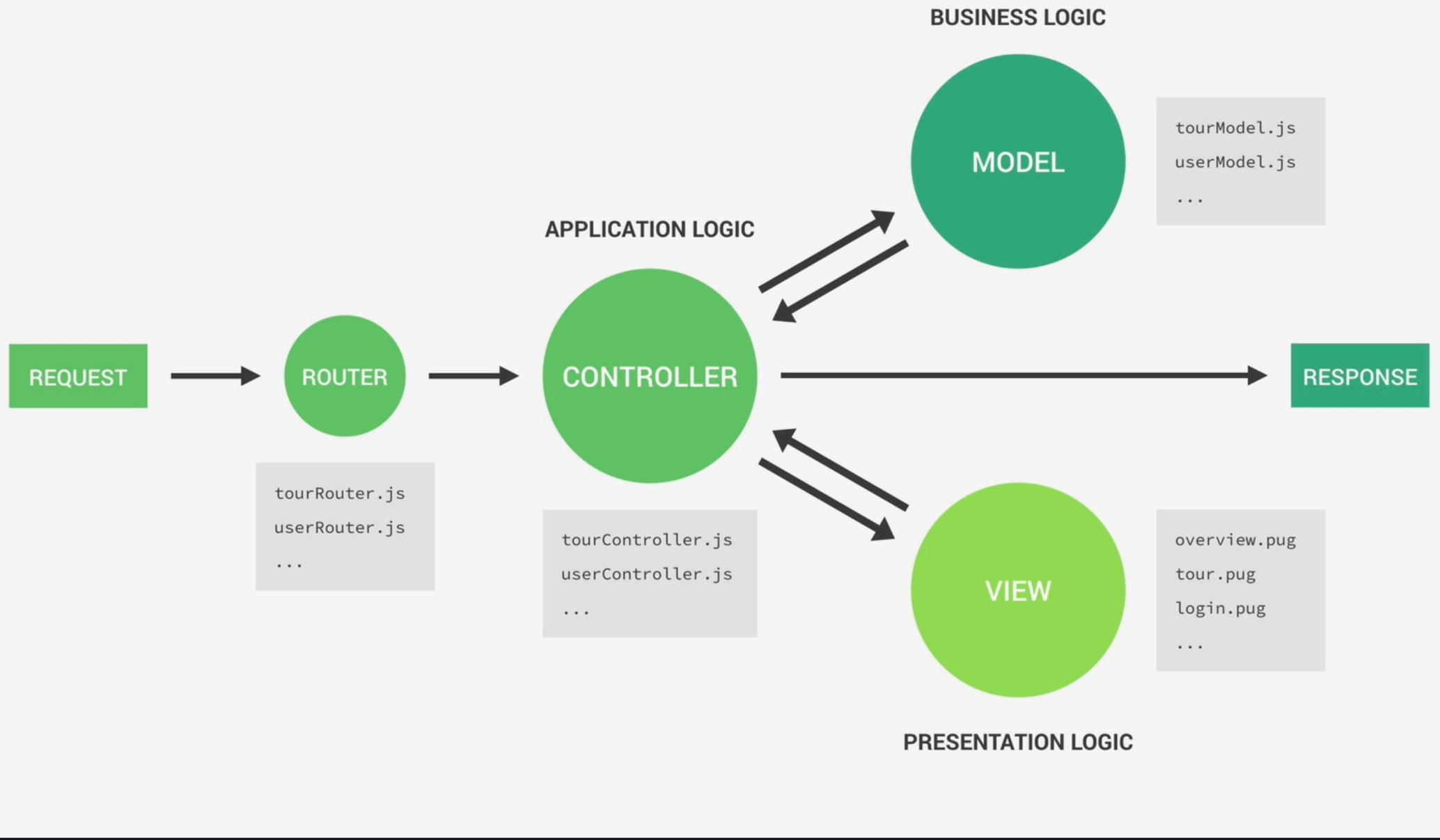
👉 **Data is always normalized**

WHAT IS MONGOOSE, AND WHY USE IT?

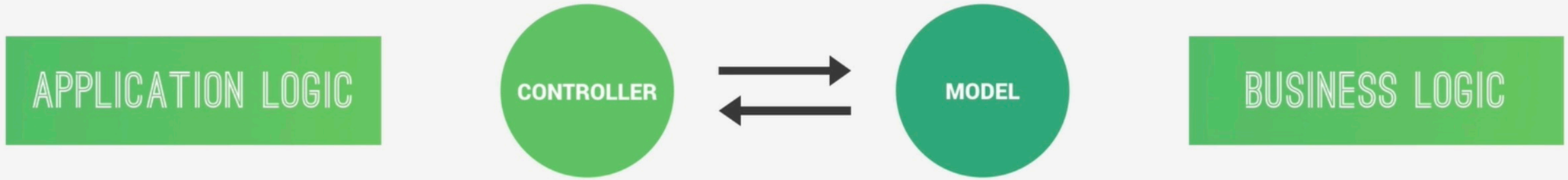


- 👉 Mongoose is an Object Data Modeling (ODM) library for MongoDB and Node.js, a higher level of abstraction;
- 👉 Mongoose allows for rapid and simple development of mongoDB database interactions;
- 👉 Features: schemas to model data and relationships, easy data validation, simple query API, middleware, 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.

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.

ERROR HANDLING IN EXPRESS: AN OVERVIEW

OPERATIONAL ERRORS

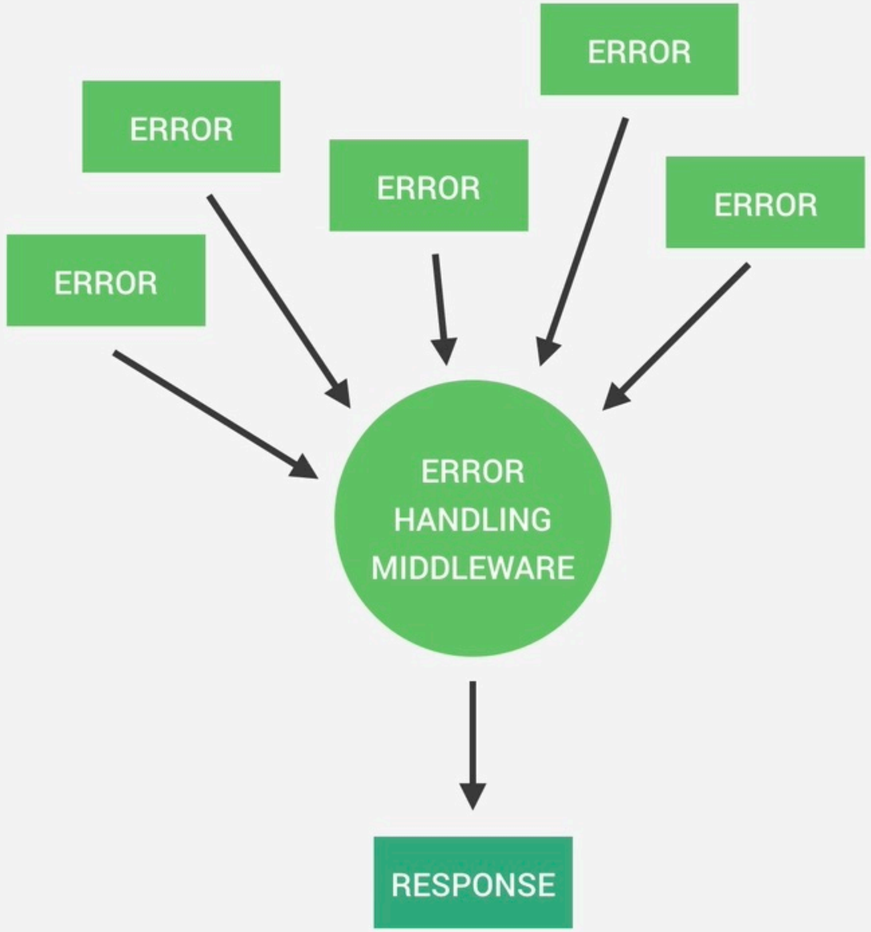
Problems that we can predict will happen at some point, so we just need to handle them in advance.

- 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...



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)

ACCESS

5 `GET /someProtectedRoute` JWT

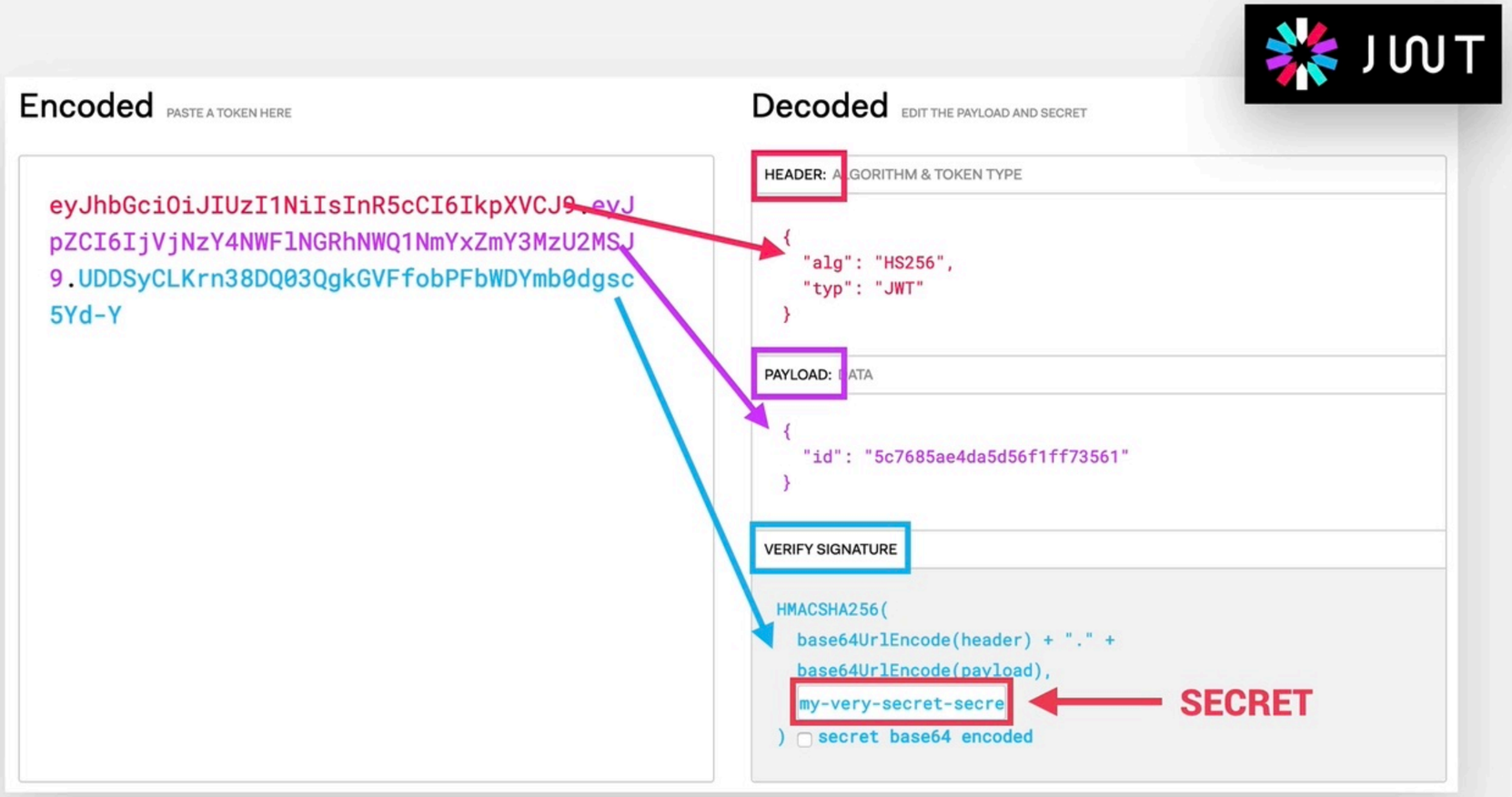
HTTPS

6 If **Valid JWT**, Allow access

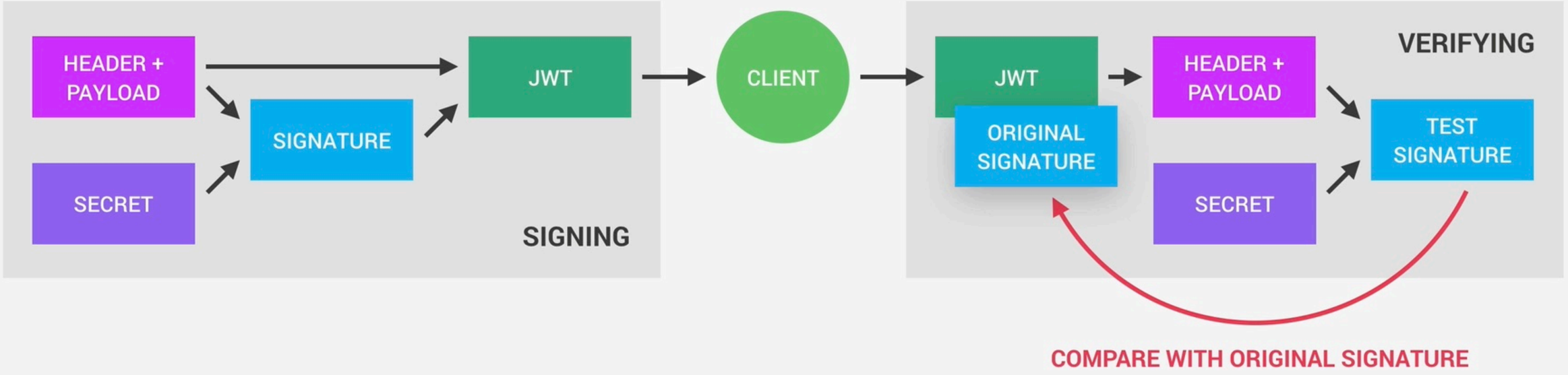
PROTECTED DATA

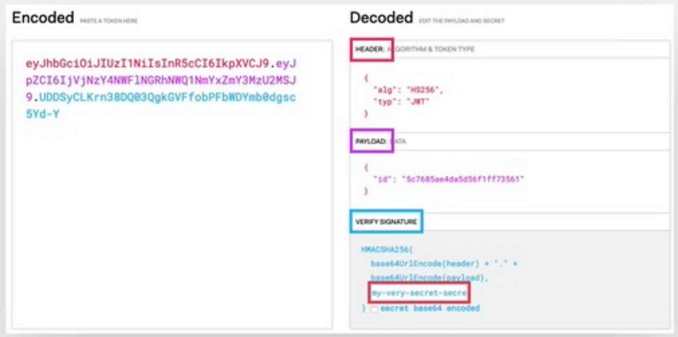
7

WHAT A JWT LOOKS LIKE



HOW SIGNING AND VERIFYING WORKS





`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!

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

"DATA... WHAT? 🤔"

DATA MODELLING

Real-world scenario

↓

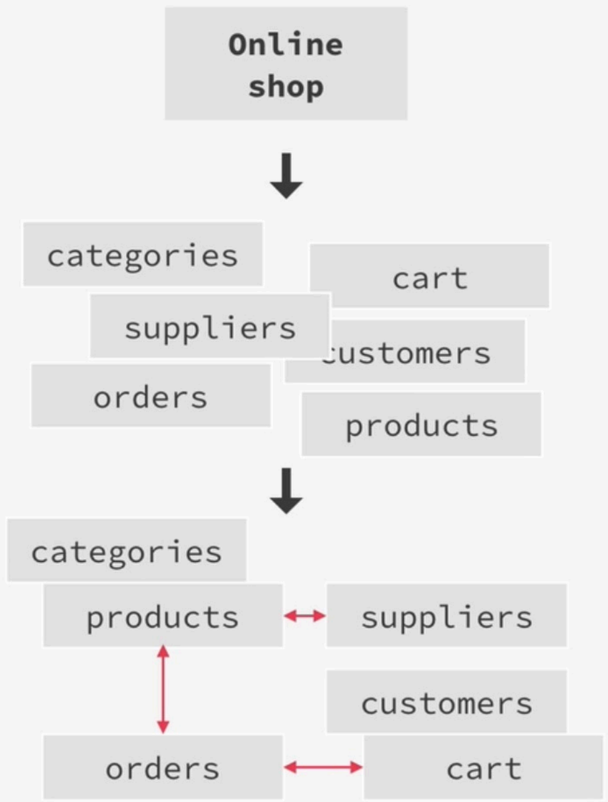
Unstructured data

↓

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

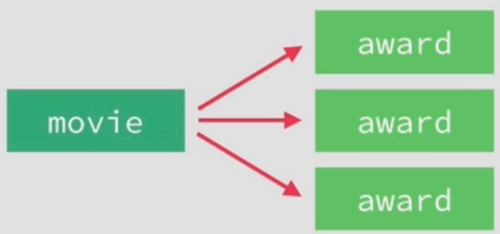
1. TYPES OF RELATIONSHIPS BETWEEN DATA

1:1

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

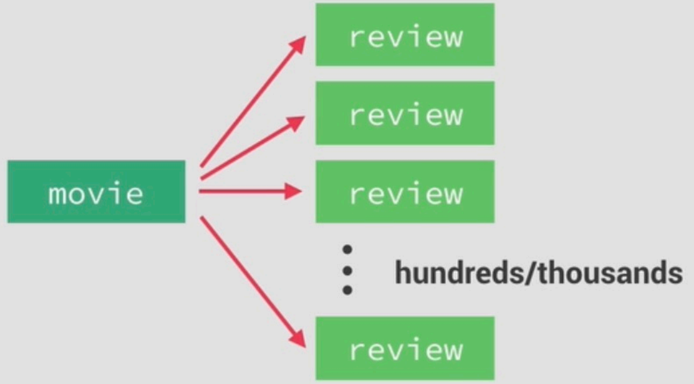
👉 1:FEW

1:MANY

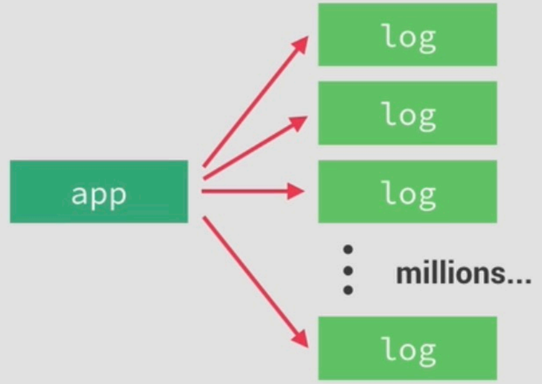


*(**1** movie can win **many** awards)*

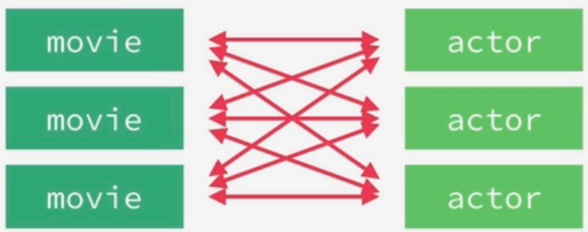
👉 1:MANY



👉 1:TON



MANY:MANY



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

2. REFERENCING VS. EMBEDDING

REFERENCED / NORMALIZED

movie

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

Referencing (child)

actor

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

actor

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

EMBEDDING/
DENORMALIZATION

REFERENCING /
NORMALIZATION

EMBEDDED / DENORMALIZED

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

Main document

Embedded documents

movie

👍 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

app

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

log

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

log

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

👉 1:FEW

PARENT REFERENCING

app

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

log

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

log

```
{
  "_id": ObjectID('28273927'),
  "app": ObjectID('23'),
  "type": "error",
  "timestamp": 1412844672
}
```

👉 1:MANY

👉 1:TON

TWO-WAY REFERENCING

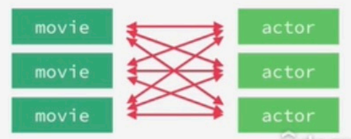
movie

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

actor

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

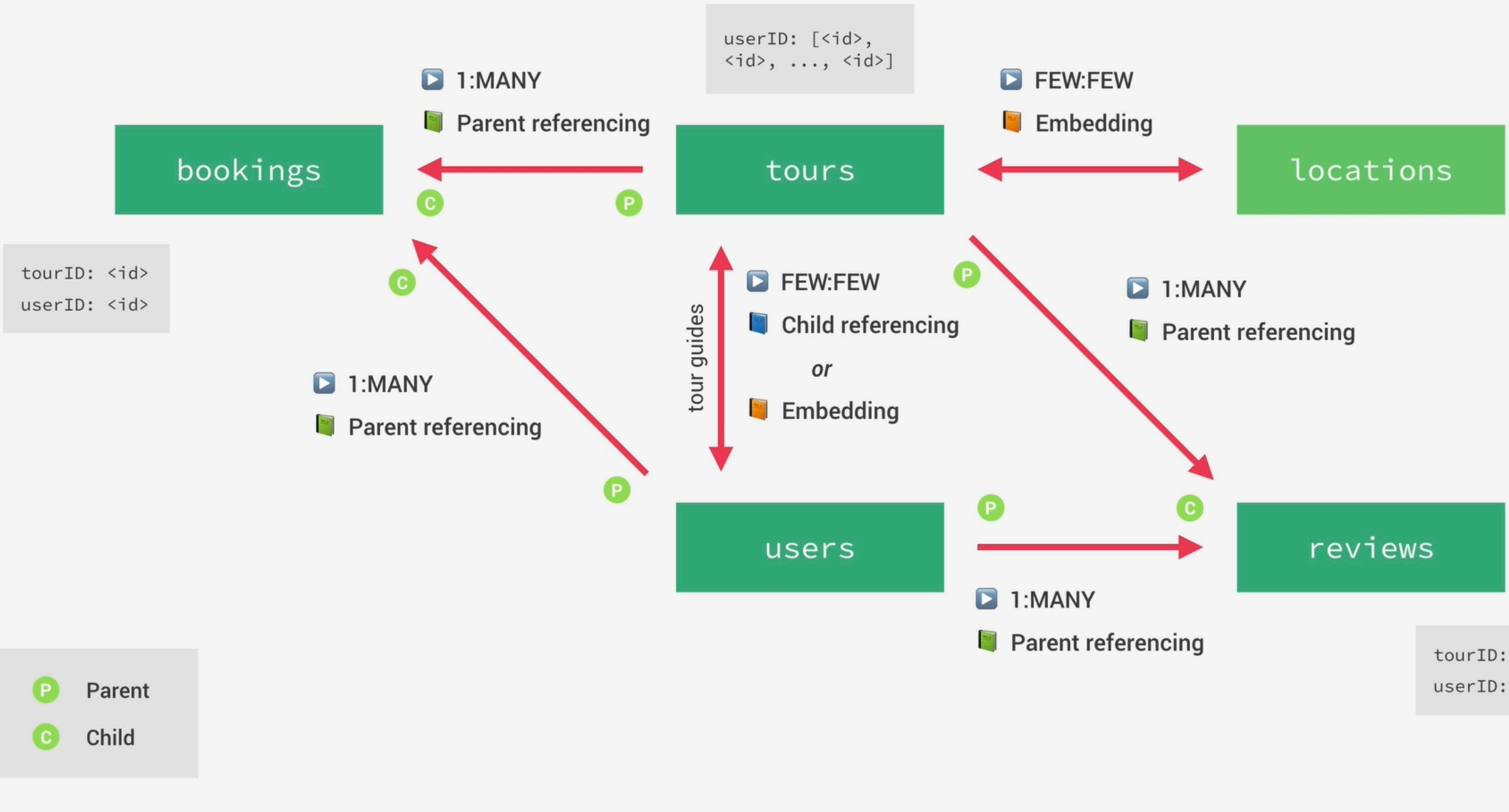
👉 MANY:MANY



SUMMARY 🥳

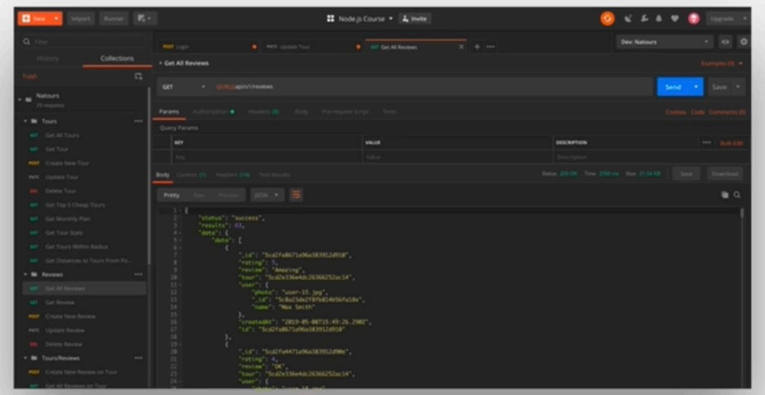
- 👉 The most important principle is: Structure your data to **match the ways that your application queries and updates data**;
- 👉 In other words: Identify the questions that arise from your **application's use cases** first, and then model your data so that the **questions can get answered** in the 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.

THE NATOURS DATA MODEL



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);

👉 I**mplement "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.